File 1 - __init__.py:

```
1: (0)              """
2: (0)              NetworkX
3: (0)              ========
4: (0)              NetworkX is a Python package for the creation, manipulation, and study of the
5: (0)              structure, dynamics, and functions of complex networks.
6: (0)              See https://networkx.org for complete documentation.
7: (0)              """
8: (0)              __version__ = "3.3"
9: (0)              from networkx.lazy_imports import _lazy_import
10: (0)             from networkx.exception import *
11: (0)             from networkx import utils
12: (0)             from networkx.utils import _clear_cache, _dispatchable, config
13: (0)             from networkx import classes
14: (0)             from networkx.classes import filters
15: (0)             from networkx.classes import *
16: (0)             from networkx import convert
17: (0)             from networkx.convert import *
18: (0)             from networkx import convert_matrix
19: (0)             from networkx.convert_matrix import *
20: (0)             from networkx import relabel
21: (0)             from networkx.relabel import *
22: (0)             from networkx import generators
23: (0)             from networkx.generators import *
24: (0)             from networkx import readwrite
25: (0)             from networkx.readwrite import *
26: (0)             from networkx import algorithms
27: (0)             from networkx.algorithms import *
28: (0)             from networkx import linalg
29: (0)             from networkx.linalg import *
30: (0)             from networkx import drawing
31: (0)             from networkx.drawing import *
```

File 2 - convert.py:

```
1: (0)              """Functions to convert NetworkX graphs to and from other formats.
2: (0)              The preferred way of converting data to a NetworkX graph is through the
3: (0)              graph constructor.  The constructor calls the to_networkx_graph() function
4: (0)              which attempts to guess the input type and convert it automatically.
5: (0)              Examples
6: (0)              --------
7: (0)              Create a graph with a single edge from a dictionary of dictionaries
8: (0)              >>> d = {0: {1: 1}}  # dict-of-dicts single edge (0,1)
9: (0)              >>> G = nx.Graph(d)
10: (0)             See Also
11: (0)             --------
12: (0)             nx_agraph, nx_pydot
13: (0)             """
14: (0)             import warnings
15: (0)             from collections.abc import Collection, Generator, Iterator
16: (0)             import networkx as nx
17: (0)             __all__ = [
18: (4)                 "to_networkx_graph",
19: (4)                 "from_dict_of_dicts",
20: (4)                 "to_dict_of_dicts",
21: (4)                 "from_dict_of_lists",
22: (4)                 "to_dict_of_lists",
23: (4)                 "from_edgelist",
24: (4)                 "to_edgelist",
25: (0)             ]
26: (0)             def to_networkx_graph(data, create_using=None, multigraph_input=False):
27: (4)                 """Make a NetworkX graph from a known data structure.
28: (4)                 The preferred way to call this is automatically
29: (4)                 from the class constructor
30: (4)                 >>> d = {0: {1: {"weight": 1}}}  # dict-of-dicts single edge (0,1)
31: (4)                 >>> G = nx.Graph(d)
```

```
 32: (4)                 instead of the equivalent
 33: (4)                 >>> G = nx.from_dict_of_dicts(d)
 34: (4)                 Parameters
 35: (4)                 ----------
 36: (4)                 data : object to be converted
 37: (8)                     Current known types are:
 38: (9)                      any NetworkX graph
 39: (9)                      dict-of-dicts
 40: (9)                      dict-of-lists
 41: (9)                      container (e.g. set, list, tuple) of edges
 42: (9)                      iterator (e.g. itertools.chain) that produces edges
 43: (9)                      generator of edges
 44: (9)                      Pandas DataFrame (row per edge)
 45: (9)                      2D numpy array
 46: (9)                      scipy sparse array
 47: (9)                      pygraphviz agraph
 48: (4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
 49: (8)                     Graph type to create. If graph instance, then cleared before
populated.
 50: (4)                 multigraph_input : bool (default False)
 51: (8)                     If True and  data is a dict_of_dicts,
 52: (8)                     try to create a multigraph assuming dict_of_dict_of_lists.
 53: (8)                     If data and create_using are both multigraphs then create
 54: (8)                     a multigraph from a multigraph.
 55: (4)                 """
 56: (4)                 if hasattr(data, "adj"):
 57: (8)                     try:
 58: (12)                        result = from_dict_of_dicts(
 59: (16)                            data.adj,
 60: (16)                            create_using=create_using,
 61: (16)                            multigraph_input=data.is_multigraph(),
 62: (12)                        )
 63: (12)                        result.graph.update(data.graph)
 64: (12)                        for n, dd in data.nodes.items():
 65: (16)                            result._node[n].update(dd)
 66: (12)                        return result
 67: (8)                     except Exception as err:
 68: (12)                        raise nx.NetworkXError("Input is not a correct NetworkX graph.")
from err
 69: (4)                 if hasattr(data, "is_strict"):
 70: (8)                     try:
 71: (12)                        return nx.nx_agraph.from_agraph(data, create_using=create_using)
 72: (8)                     except Exception as err:
 73: (12)                        raise nx.NetworkXError("Input is not a correct pygraphviz graph.")
from err
 74: (4)                 if isinstance(data, dict):
 75: (8)                     try:
 76: (12)                        return from_dict_of_dicts(
 77: (16)                            data, create_using=create_using,
multigraph_input=multigraph_input
 78: (12)                        )
 79: (8)                     except Exception as err1:
 80: (12)                        if multigraph_input is True:
 81: (16)                            raise nx.NetworkXError(
 82: (20)                                f"converting multigraph_input raised:\n{type(err1)}:
{err1}"
 83: (16)                            )
 84: (12)                        try:
 85: (16)                            return from_dict_of_lists(data, create_using=create_using)
 86: (12)                        except Exception as err2:
 87: (16)                            raise TypeError("Input is not known type.") from err2
 88: (4)                 try:
 89: (8)                     import pandas as pd
 90: (8)                     if isinstance(data, pd.DataFrame):
 91: (12)                        if data.shape[0] == data.shape[1]:
 92: (16)                            try:
 93: (20)                                return nx.from_pandas_adjacency(data,
create_using=create_using)
 94: (16)                            except Exception as err:
```

```
95: (20)                                msg = "Input is not a correct Pandas DataFrame adjacency
matrix."
96: (20)                                raise nx.NetworkXError(msg) from err
97: (12)                        else:
98: (16)                            try:
99: (20)                                return nx.from_pandas_edgelist(
100: (24)                                   data, edge_attr=True, create_using=create_using
101: (20)                               )
102: (16)                           except Exception as err:
103: (20)                               msg = "Input is not a correct Pandas DataFrame edge-list."
104: (20)                               raise nx.NetworkXError(msg) from err
105: (4)                except ImportError:
106: (8)                    warnings.warn("pandas not found, skipping conversion test.",
ImportWarning)
107: (4)                try:
108: (8)                    import numpy as np
109: (8)                    if isinstance(data, np.ndarray):
110: (12)                       try:
111: (16)                           return nx.from_numpy_array(data, create_using=create_using)
112: (12)                       except Exception as err:
113: (16)                           raise nx.NetworkXError(
114: (20)                               f"Failed to interpret array as an adjacency matrix."
115: (16)                           ) from err
116: (4)                except ImportError:
117: (8)                    warnings.warn("numpy not found, skipping conversion test.",
ImportWarning)
118: (4)                try:
119: (8)                    import scipy
120: (8)                    if hasattr(data, "format"):
121: (12)                       try:
122: (16)                           return nx.from_scipy_sparse_array(data,
create_using=create_using)
123: (12)                       except Exception as err:
124: (16)                           raise nx.NetworkXError(
125: (20)                               "Input is not a correct scipy sparse array type."
126: (16)                           ) from err
127: (4)                except ImportError:
128: (8)                    warnings.warn("scipy not found, skipping conversion test.",
ImportWarning)
129: (4)                if isinstance(data, Collection | Generator | Iterator):
130: (8)                    try:
131: (12)                       return from_edgelist(data, create_using=create_using)
132: (8)                    except Exception as err:
133: (12)                       raise nx.NetworkXError("Input is not a valid edge list") from err
134: (4)                raise nx.NetworkXError("Input is not a known data type for conversion.")
135: (0)            @nx._dispatchable
136: (0)            def to_dict_of_lists(G, nodelist=None):
137: (4)                """Returns adjacency representation of graph as a dictionary of lists.
138: (4)                Parameters
139: (4)                ----------
140: (4)                G : graph
141: (7)                   A NetworkX graph
142: (4)                nodelist : list
143: (7)                   Use only nodes specified in nodelist
144: (4)                Notes
145: (4)                -----
146: (4)                Completely ignores edge data for MultiGraph and MultiDiGraph.
147: (4)                """
148: (4)                if nodelist is None:
149: (8)                    nodelist = G
150: (4)                d = {}
151: (4)                for n in nodelist:
152: (8)                    d[n] = [nbr for nbr in G.neighbors(n) if nbr in nodelist]
153: (4)                return d
154: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
155: (0)            def from_dict_of_lists(d, create_using=None):
156: (4)                """Returns a graph from a dictionary of lists.
157: (4)                Parameters
158: (4)                ----------
```

```
159: (4)                d : dictionary of lists
160: (6)                  A dictionary of lists adjacency representation.
161: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
162: (8)                    Graph type to create. If graph instance, then cleared before
populated.
163: (4)                Examples
164: (4)                --------
165: (4)                >>> dol = {0: [1]}  # single edge (0,1)
166: (4)                >>> G = nx.from_dict_of_lists(dol)
167: (4)                or
168: (4)                >>> G = nx.Graph(dol)  # use Graph constructor
169: (4)                """
170: (4)                G = nx.empty_graph(0, create_using)
171: (4)                G.add_nodes_from(d)
172: (4)                if G.is_multigraph() and not G.is_directed():
173: (8)                    seen = {}
174: (8)                    for node, nbrlist in d.items():
175: (12)                       for nbr in nbrlist:
176: (16)                           if nbr not in seen:
177: (20)                               G.add_edge(node, nbr)
178: (12)                       seen[node] = 1  # don't allow reverse edge to show up
179: (4)                else:
180: (8)                    G.add_edges_from(
181: (12)                       ((node, nbr) for node, nbrlist in d.items() for nbr in nbrlist)
182: (8)                    )
183: (4)                return G
184: (0)            def to_dict_of_dicts(G, nodelist=None, edge_data=None):
185: (4)                """Returns adjacency representation of graph as a dictionary of
dictionaries.
186: (4)                Parameters
187: (4)                ----------
188: (4)                G : graph
189: (7)                   A NetworkX graph
190: (4)                nodelist : list
191: (7)                   Use only nodes specified in nodelist
192: (4)                edge_data : scalar, optional
193: (7)                   If provided, the value of the dictionary will be set to `edge_data` for
194: (7)                   all edges. Usual values could be `1` or `True`. If `edge_data` is
195: (7)                   `None` (the default), the edgedata in `G` is used, resulting in a
196: (7)                   dict-of-dict-of-dicts. If `G` is a MultiGraph, the result will be a
197: (7)                   dict-of-dict-of-dict-of-dicts. See Notes for an approach to customize
198: (7)                   handling edge data. `edge_data` should *not* be a container.
199: (4)                Returns
200: (4)                -------
201: (4)                dod : dict
202: (7)                   A nested dictionary representation of `G`. Note that the level of
203: (7)                   nesting depends on the type of `G` and the value of `edge_data`
204: (7)                   (see Examples).
205: (4)                See Also
206: (4)                --------
207: (4)                from_dict_of_dicts, to_dict_of_lists
208: (4)                Notes
209: (4)                -----
210: (4)                For a more custom approach to handling edge data, try::
211: (8)                    dod = {
212: (12)                       n: {nbr: custom(n, nbr, dd) for nbr, dd in nbrdict.items()}
213: (12)                       for n, nbrdict in G.adj.items()
214: (8)                    }
215: (4)                where `custom` returns the desired edge data for each edge between `n` and
216: (4)                `nbr`, given existing edge data `dd`.
217: (4)                Examples
218: (4)                --------
219: (4)                >>> G = nx.path_graph(3)
220: (4)                >>> nx.to_dict_of_dicts(G)
221: (4)                {0: {1: {}}, 1: {0: {}, 2: {}}, 2: {1: {}}}
222: (4)                Edge data is preserved by default (``edge_data=None``), resulting
223: (4)                in dict-of-dict-of-dicts where the innermost dictionary contains the
224: (4)                edge data:
225: (4)                >>> G = nx.Graph()
```

```
226: (4)                >>> G.add_edges_from(
227: (4)                ...     [
228: (4)                ...         (0, 1, {"weight": 1.0}),
229: (4)                ...         (1, 2, {"weight": 2.0}),
230: (4)                ...         (2, 0, {"weight": 1.0}),
231: (4)                ...     ]
232: (4)                ... )
233: (4)                >>> d = nx.to_dict_of_dicts(G)
234: (4)                >>> d  # doctest: +SKIP
235: (4)                {0: {1: {'weight': 1.0}, 2: {'weight': 1.0}},
236: (5)                 1: {0: {'weight': 1.0}, 2: {'weight': 2.0}},
237: (5)                 2: {1: {'weight': 2.0}, 0: {'weight': 1.0}}}
238: (4)                >>> d[1][2]["weight"]
239: (4)                2.0
240: (4)                If `edge_data` is not `None`, edge data in the original graph (if any) is
241: (4)                replaced:
242: (4)                >>> d = nx.to_dict_of_dicts(G, edge_data=1)
243: (4)                >>> d
244: (4)                {0: {1: 1, 2: 1}, 1: {0: 1, 2: 1}, 2: {1: 1, 0: 1}}
245: (4)                >>> d[1][2]
246: (4)                1
247: (4)                This also applies to MultiGraphs: edge data is preserved by default:
248: (4)                >>> G = nx.MultiGraph()
249: (4)                >>> G.add_edge(0, 1, key="a", weight=1.0)
250: (4)                'a'
251: (4)                >>> G.add_edge(0, 1, key="b", weight=5.0)
252: (4)                'b'
253: (4)                >>> d = nx.to_dict_of_dicts(G)
254: (4)                >>> d  # doctest: +SKIP
255: (4)                {0: {1: {'a': {'weight': 1.0}, 'b': {'weight': 5.0}}},
256: (5)                 1: {0: {'a': {'weight': 1.0}, 'b': {'weight': 5.0}}}}
257: (4)                >>> d[0][1]["b"]["weight"]
258: (4)                5.0
259: (4)                But multi edge data is lost if `edge_data` is not `None`:
260: (4)                >>> d = nx.to_dict_of_dicts(G, edge_data=10)
261: (4)                >>> d
262: (4)                {0: {1: 10}, 1: {0: 10}}
263: (4)                """
264: (4)                dod = {}
265: (4)                if nodelist is None:
266: (8)                    if edge_data is None:
267: (12)                       for u, nbrdict in G.adjacency():
268: (16)                           dod[u] = nbrdict.copy()
269: (8)                    else:  # edge_data is not None
270: (12)                       for u, nbrdict in G.adjacency():
271: (16)                           dod[u] = dod.fromkeys(nbrdict, edge_data)
272: (4)                else:  # nodelist is not None
273: (8)                    if edge_data is None:
274: (12)                       for u in nodelist:
275: (16)                           dod[u] = {}
276: (16)                           for v, data in ((v, data) for v, data in G[u].items() if v in
nodelist):
277: (20)                               dod[u][v] = data
278: (8)                    else:  # nodelist and edge_data are not None
279: (12)                       for u in nodelist:
280: (16)                           dod[u] = {}
281: (16)                           for v in (v for v in G[u] if v in nodelist):
282: (20)                               dod[u][v] = edge_data
283: (4)                return dod
284: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
285: (0)            def from_dict_of_dicts(d, create_using=None, multigraph_input=False):
286: (4)                """Returns a graph from a dictionary of dictionaries.
287: (4)                Parameters
288: (4)                ----------
289: (4)                d : dictionary of dictionaries
290: (6)                  A dictionary of dictionaries adjacency representation.
291: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
292: (8)                    Graph type to create. If graph instance, then cleared before
populated.
```

```
(4)                 multigraph_input : bool (default False)
(7)                    When True, the dict `d` is assumed
(7)                    to be a dict-of-dict-of-dict-of-dict structure keyed by
(7)                    node to neighbor to edge keys to edge data for multi-edges.
(7)                    Otherwise this routine assumes dict-of-dict-of-dict keyed by
(7)                    node to neighbor to edge data.
(4)                 Examples
(4)                 --------
(4)                 >>> dod = {0: {1: {"weight": 1}}}  # single edge (0,1)
(4)                 >>> G = nx.from_dict_of_dicts(dod)
(4)                 or
(4)                 >>> G = nx.Graph(dod)  # use Graph constructor
(4)                 """
(4)                 G = nx.empty_graph(0, create_using)
(4)                 G.add_nodes_from(d)
(4)                 if multigraph_input:
(8)                     if G.is_directed():
(12)                        if G.is_multigraph():
(16)                            G.add_edges_from(
(20)                                (u, v, key, data)
(20)                                for u, nbrs in d.items()
(20)                                for v, datadict in nbrs.items()
(20)                                for key, data in datadict.items()
(16)                            )
(12)                        else:
(16)                            G.add_edges_from(
(20)                                (u, v, data)
(20)                                for u, nbrs in d.items()
(20)                                for v, datadict in nbrs.items()
(20)                                for key, data in datadict.items()
(16)                            )
(8)                     else:  # Undirected
(12)                        if G.is_multigraph():
(16)                            seen = set()  # don't add both directions of undirected graph
(16)                            for u, nbrs in d.items():
(20)                                for v, datadict in nbrs.items():
(24)                                    if (u, v) not in seen:
(28)                                        G.add_edges_from(
(32)                                            (u, v, key, data) for key, data in
datadict.items()
(28)                                        )
(28)                                        seen.add((v, u))
(12)                        else:
(16)                            seen = set()  # don't add both directions of undirected graph
(16)                            for u, nbrs in d.items():
(20)                                for v, datadict in nbrs.items():
(24)                                    if (u, v) not in seen:
(28)                                        G.add_edges_from(
(32)                                            (u, v, data) for key, data in datadict.items()
(28)                                        )
(28)                                        seen.add((v, u))
(4)                 else:  # not a multigraph to multigraph transfer
(8)                     if G.is_multigraph() and not G.is_directed():
(12)                        seen = set()
(12)                        for u, nbrs in d.items():
(16)                            for v, data in nbrs.items():
(20)                                if (u, v) not in seen:
(24)                                    G.add_edge(u, v, key=0)
(24)                                    G[u][v][0].update(data)
(20)                                seen.add((v, u))
(8)                     else:
(12)                        G.add_edges_from(
(16)                            ((u, v, data) for u, nbrs in d.items() for v, data in
nbrs.items())
(12)                        )
(4)                 return G
(0)             @nx._dispatchable(preserve_edge_attrs=True)
(0)             def to_edgelist(G, nodelist=None):
(4)                 """Returns a list of edges in the graph.
```

```
360: (4)                Parameters
361: (4)                ----------
362: (4)                G : graph
363: (7)                   A NetworkX graph
364: (4)                nodelist : list
365: (7)                   Use only nodes specified in nodelist
366: (4)                """
367: (4)                if nodelist is None:
368: (8)                    return G.edges(data=True)
369: (4)                return G.edges(nodelist, data=True)
370: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
371: (0)            def from_edgelist(edgelist, create_using=None):
372: (4)                """Returns a graph from a list of edges.
373: (4)                Parameters
374: (4)                ----------
375: (4)                edgelist : list or iterator
376: (6)                  Edge tuples
377: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
378: (8)                    Graph type to create. If graph instance, then cleared before
populated.
379: (4)                Examples
380: (4)                --------
381: (4)                >>> edgelist = [(0, 1)]  # single edge (0,1)
382: (4)                >>> G = nx.from_edgelist(edgelist)
383: (4)                or
384: (4)                >>> G = nx.Graph(edgelist)  # use Graph constructor
385: (4)                """
386: (4)                G = nx.empty_graph(0, create_using)
387: (4)                G.add_edges_from(edgelist)
388: (4)                return G
```

File 3 - conftest.py:

```
1: (0)              """
2: (0)              Testing
3: (0)              =======
4: (0)              General guidelines for writing good tests:
5: (0)              - doctests always assume ``import networkx as nx`` so don't add that
6: (0)              - prefer pytest fixtures over classes with setup methods.
7: (0)              - use the ``@pytest.mark.parametrize``  decorator
8: (0)              - use ``pytest.importorskip`` for numpy, scipy, pandas, and matplotlib b/c of
PyPy.
9: (2)                and add the module to the relevant entries below.
10: (0)             """
11: (0)             import os
12: (0)             import sys
13: (0)             import warnings
14: (0)             from importlib.metadata import entry_points
15: (0)             import pytest
16: (0)             import networkx
17: (0)             def pytest_addoption(parser):
18: (4)                 parser.addoption(
19: (8)                     "--runslow", action="store_true", default=False, help="run slow tests"
20: (4)                 )
21: (4)                 parser.addoption(
22: (8)                     "--backend",
23: (8)                     action="store",
24: (8)                     default=None,
25: (8)                     help="Run tests with a backend by auto-converting nx graphs to backend
graphs",
26: (4)                 )
27: (4)                 parser.addoption(
28: (8)                     "--fallback-to-nx",
29: (8)                     action="store_true",
30: (8)                     default=False,
31: (8)                     help="Run nx function if a backend doesn't implement a dispatchable
function"
```

```
32: (8)                     " (use with --backend)",
33: (4)                 )
34: (0)             def pytest_configure(config):
35: (4)                 config.addinivalue_line("markers", "slow: mark test as slow to run")
36: (4)                 backend = config.getoption("--backend")
37: (4)                 if backend is None:
38: (8)                     backend = os.environ.get("NETWORKX_TEST_BACKEND")
39: (4)                 backends = entry_points(name="nx-loopback", group="networkx.backends")
40: (4)                 if backends:
41: (8)                     networkx.utils.backends.backends["nx-loopback"] = next(iter(backends))
42: (4)                 else:
43: (8)                     warnings.warn(
44: (12)                        "\n\n             WARNING: Mixed NetworkX configuration! \n\n"
45: (12)                        "        This environment has mixed configuration for networkx.\n"
46: (12)                        "        The test object nx-loopback is not configured
correctly.\n"
47: (12)                        "        You should not be seeing this message.\n"
48: (12)                        "        Try `pip install -e .`, or change your PYTHONPATH\n"
49: (12)                        "        Make sure python finds the networkx repo you are
testing\n\n"
50: (8)                     )
51: (4)                 if backend:
52: (8)                     networkx.config["backend_priority"] = [backend]
53: (8)                     fallback_to_nx = config.getoption("--fallback-to-nx")
54: (8)                     if not fallback_to_nx:
55: (12)                        fallback_to_nx = os.environ.get("NETWORKX_FALLBACK_TO_NX")
56: (8)                     networkx.utils.backends._dispatchable._fallback_to_nx =
bool(fallback_to_nx)
57: (0)             def pytest_collection_modifyitems(config, items):
58: (4)                 networkx.utils.backends._dispatchable._is_testing = True
59: (4)                 if backend_priority := networkx.config["backend_priority"]:
60: (8)                     backend = networkx.utils.backends.backends[backend_priority[0]].load()
61: (8)                     if hasattr(backend, "on_start_tests"):
62: (12)                        getattr(backend, "on_start_tests")(items)
63: (4)                 if config.getoption("--runslow"):
64: (8)                     return
65: (4)                 skip_slow = pytest.mark.skip(reason="need --runslow option to run")
66: (4)                 for item in items:
67: (8)                     if "slow" in item.keywords:
68: (12)                        item.add_marker(skip_slow)
69: (0)             @pytest.fixture(autouse=True)
70: (0)             def set_warnings():
71: (4)                 warnings.filterwarnings(
72: (8)                     "ignore",
73: (8)                     category=FutureWarning,
74: (8)                     message="\n\nsingle_target_shortest_path_length",
75: (4)                 )
76: (4)                 warnings.filterwarnings(
77: (8)                     "ignore",
78: (8)                     category=FutureWarning,
79: (8)                     message="\n\nshortest_path",
80: (4)                 )
81: (4)                 warnings.filterwarnings(
82: (8)                     "ignore", category=DeprecationWarning, message="\nforest_str is
deprecated"
83: (4)                 )
84: (4)                 warnings.filterwarnings(
85: (8)                     "ignore", category=DeprecationWarning, message="\n\nrandom_tree"
86: (4)                 )
87: (4)                 warnings.filterwarnings(
88: (8)                     "ignore", category=DeprecationWarning, message="Edmonds has been
deprecated"
89: (4)                 )
90: (4)                 warnings.filterwarnings(
91: (8)                     "ignore",
92: (8)                     category=DeprecationWarning,
93: (8)                     message="MultiDiGraph_EdgeKey has been deprecated",
94: (4)                 )
95: (4)                 warnings.filterwarnings(
```

```
96: (8)                     "ignore", category=DeprecationWarning, message="\n\nThe `normalized`"
97: (4)                 )
98: (4)                 warnings.filterwarnings(
99: (8)                     "ignore",
100: (8)                    category=DeprecationWarning,
101: (8)                    message="The function `join` is deprecated",
102: (4)                )
103: (4)                warnings.filterwarnings(
104: (8)                    "ignore",
105: (8)                    category=DeprecationWarning,
106: (8)                    message="\n\nstrongly_connected_components_recursive",
107: (4)                )
108: (4)                warnings.filterwarnings(
109: (8)                    "ignore", category=DeprecationWarning, message="\n\nall_triplets"
110: (4)                )
111: (4)                warnings.filterwarnings(
112: (8)                    "ignore", category=DeprecationWarning, message="\n\nrandom_triad"
113: (4)                )
114: (4)                warnings.filterwarnings(
115: (8)                    "ignore", category=DeprecationWarning, message="minimal_d_separator"
116: (4)                )
117: (4)                warnings.filterwarnings(
118: (8)                    "ignore", category=DeprecationWarning, message="d_separated"
119: (4)                )
120: (4)                warnings.filterwarnings("ignore", category=DeprecationWarning,
message="\n\nk_core")
121: (4)                warnings.filterwarnings(
122: (8)                    "ignore", category=DeprecationWarning, message="\n\nk_shell"
123: (4)                )
124: (4)                warnings.filterwarnings(
125: (8)                    "ignore", category=DeprecationWarning, message="\n\nk_crust"
126: (4)                )
127: (4)                warnings.filterwarnings(
128: (8)                    "ignore", category=DeprecationWarning, message="\n\nk_corona"
129: (4)                )
130: (4)                warnings.filterwarnings(
131: (8)                    "ignore", category=DeprecationWarning,
message="\n\ntotal_spanning_tree_weight"
132: (4)                )
133: (4)                warnings.filterwarnings(
134: (8)                    "ignore", category=DeprecationWarning, message=r"\n\nThe
'create=matrix'"
135: (4)                )
136: (0)            @pytest.fixture(autouse=True)
137: (0)            def add_nx(doctest_namespace):
138: (4)                doctest_namespace["nx"] = networkx
139: (0)            try:
140: (4)                import numpy
141: (4)                has_numpy = True
142: (0)            except ImportError:
143: (4)                has_numpy = False
144: (0)            try:
145: (4)                import scipy
146: (4)                has_scipy = True
147: (0)            except ImportError:
148: (4)                has_scipy = False
149: (0)            try:
150: (4)                import matplotlib
151: (4)                has_matplotlib = True
152: (0)            except ImportError:
153: (4)                has_matplotlib = False
154: (0)            try:
155: (4)                import pandas
156: (4)                has_pandas = True
157: (0)            except ImportError:
158: (4)                has_pandas = False
159: (0)            try:
160: (4)                import pygraphviz
161: (4)                has_pygraphviz = True
```

```
(0)  except ImportError:
(4)      has_pygraphviz = False
(0)  try:
(4)      import pydot
(4)      has_pydot = True
(0)  except ImportError:
(4)      has_pydot = False
(0)  try:
(4)      import sympy
(4)      has_sympy = True
(0)  except ImportError:
(4)      has_sympy = False
(0)  collect_ignore = []
(0)  needs_numpy = [
(4)      "algorithms/approximation/traveling_salesman.py",
(4)      "algorithms/centrality/current_flow_closeness.py",
(4)      "algorithms/node_classification.py",
(4)      "algorithms/non_randomness.py",
(4)      "algorithms/shortest_paths/dense.py",
(4)      "algorithms/tree/mst.py",
(4)      "generators/expanders.py",
(4)      "linalg/bethehessianmatrix.py",
(4)      "linalg/laplacianmatrix.py",
(4)      "utils/misc.py",
(4)      "algorithms/centrality/laplacian.py",
(0)  ]
(0)  needs_scipy = [
(4)      "algorithms/approximation/traveling_salesman.py",
(4)      "algorithms/assortativity/correlation.py",
(4)      "algorithms/assortativity/mixing.py",
(4)      "algorithms/assortativity/pairs.py",
(4)      "algorithms/bipartite/matrix.py",
(4)      "algorithms/bipartite/spectral.py",
(4)      "algorithms/centrality/current_flow_betweenness.py",
(4)      "algorithms/centrality/current_flow_betweenness_subset.py",
(4)      "algorithms/centrality/eigenvector.py",
(4)      "algorithms/centrality/katz.py",
(4)      "algorithms/centrality/laplacian.py",
(4)      "algorithms/centrality/second_order.py",
(4)      "algorithms/centrality/subgraph_alg.py",
(4)      "algorithms/communicability_alg.py",
(4)      "algorithms/community/divisive.py",
(4)      "algorithms/distance_measures.py",
(4)      "algorithms/link_analysis/hits_alg.py",
(4)      "algorithms/link_analysis/pagerank_alg.py",
(4)      "algorithms/node_classification.py",
(4)      "algorithms/similarity.py",
(4)      "algorithms/tree/mst.py",
(4)      "algorithms/walks.py",
(4)      "convert_matrix.py",
(4)      "drawing/layout.py",
(4)      "drawing/nx_pylab.py",
(4)      "generators/spectral_graph_forge.py",
(4)      "generators/expanders.py",
(4)      "linalg/algebraicconnectivity.py",
(4)      "linalg/attrmatrix.py",
(4)      "linalg/bethehessianmatrix.py",
(4)      "linalg/graphmatrix.py",
(4)      "linalg/laplacianmatrix.py",
(4)      "linalg/modularitymatrix.py",
(4)      "linalg/spectrum.py",
(4)      "utils/rcm.py",
(0)  ]
(0)  needs_matplotlib = ["drawing/nx_pylab.py"]
(0)  needs_pandas = ["convert_matrix.py"]
(0)  needs_pygraphviz = ["drawing/nx_agraph.py"]
(0)  needs_pydot = ["drawing/nx_pydot.py"]
(0)  needs_sympy = ["algorithms/polynomials.py"]
(0)  if not has_numpy:
```

```
231: (4)                collect_ignore += needs_numpy
232: (0)            if not has_scipy:
233: (4)                collect_ignore += needs_scipy
234: (0)            if not has_matplotlib:
235: (4)                collect_ignore += needs_matplotlib
236: (0)            if not has_pandas:
237: (4)                collect_ignore += needs_pandas
238: (0)            if not has_pygraphviz:
239: (4)                collect_ignore += needs_pygraphviz
240: (0)            if not has_pydot:
241: (4)                collect_ignore += needs_pydot
242: (0)            if not has_sympy:
243: (4)                collect_ignore += needs_sympy
```

File 4 - convert_matrix.py:

```
1: (0)              """Functions to convert NetworkX graphs to and from common data containers
2: (0)              like numpy arrays, scipy sparse arrays, and pandas DataFrames.
3: (0)              The preferred way of converting data to a NetworkX graph is through the
4: (0)              graph constructor.  The constructor calls the
`~networkx.convert.to_networkx_graph`
5: (0)              function which attempts to guess the input type and convert it automatically.
6: (0)              Examples
7: (0)              --------
8: (0)              Create a 10 node random graph from a numpy array
9: (0)              >>> import numpy as np
10: (0)             >>> rng = np.random.default_rng()
11: (0)             >>> a = rng.integers(low=0, high=2, size=(10, 10))
12: (0)             >>> DG = nx.from_numpy_array(a, create_using=nx.DiGraph)
13: (0)             or equivalently:
14: (0)             >>> DG = nx.DiGraph(a)
15: (0)             which calls `from_numpy_array` internally based on the type of ``a``.
16: (0)             See Also
17: (0)             --------
18: (0)             nx_agraph, nx_pydot
19: (0)             """
20: (0)             import itertools
21: (0)             from collections import defaultdict
22: (0)             import networkx as nx
23: (0)             from networkx.utils import not_implemented_for
24: (0)             __all__ = [
25: (4)                 "from_pandas_adjacency",
26: (4)                 "to_pandas_adjacency",
27: (4)                 "from_pandas_edgelist",
28: (4)                 "to_pandas_edgelist",
29: (4)                 "from_scipy_sparse_array",
30: (4)                 "to_scipy_sparse_array",
31: (4)                 "from_numpy_array",
32: (4)                 "to_numpy_array",
33: (0)             ]
34: (0)             @nx._dispatchable(edge_attrs="weight")
35: (0)             def to_pandas_adjacency(
36: (4)                 G,
37: (4)                 nodelist=None,
38: (4)                 dtype=None,
39: (4)                 order=None,
40: (4)                 multigraph_weight=sum,
41: (4)                 weight="weight",
42: (4)                 nonedge=0.0,
43: (0)             ):
44: (4)                 """Returns the graph adjacency matrix as a Pandas DataFrame.
45: (4)                 Parameters
46: (4)                 ----------
47: (4)                 G : graph
48: (8)                     The NetworkX graph used to construct the Pandas DataFrame.
49: (4)                 nodelist : list, optional
50: (7)                    The rows and columns are ordered according to the nodes in `nodelist`.
```

```
   If `nodelist` is None, then the ordering is produced by G.nodes().
multigraph_weight : {sum, min, max}, optional
    An operator that determines how weights in multigraphs are handled.
    The default is to sum the weights of the multiple edges.
weight : string or None, optional
    The edge attribute that holds the numerical value used for
    the edge weight.  If an edge does not have that attribute, then the
    value 1 is used instead.
nonedge : float, optional
    The matrix values corresponding to nonedges are typically set to zero.
    However, this could be undesirable if there are matrix values
    corresponding to actual edges that also have the value zero. If so,
    one might prefer nonedges to have some other value, such as nan.
Returns
-------
df : Pandas DataFrame
   Graph adjacency matrix
Notes
-----
For directed graphs, entry i,j corresponds to an edge from i to j.
The DataFrame entries are assigned to the weight edge attribute. When
an edge does not have a weight attribute, the value of the entry is set to
the number 1.  For multiple (parallel) edges, the values of the entries
are determined by the 'multigraph_weight' parameter.  The default is to
sum the weight attributes for each of the parallel edges.
When `nodelist` does not contain every node in `G`, the matrix is built
from the subgraph of `G` that is induced by the nodes in `nodelist`.
The convention used for self-loop edges in graphs is to assign the
diagonal matrix entry value to the weight attribute of the edge
(or the number 1 if the edge has no weight attribute).  If the
alternate convention of doubling the edge weight is desired the
resulting Pandas DataFrame can be modified as follows::
    >>> import pandas as pd
    >>> G = nx.Graph([(1, 1), (2, 2)])
    >>> df = nx.to_pandas_adjacency(G)
    >>> df
         1    2
    1  1.0  0.0
    2  0.0  1.0
    >>> diag_idx = list(range(len(df)))
    >>> df.iloc[diag_idx, diag_idx] *= 2
    >>> df
         1    2
    1  2.0  0.0
    2  0.0  2.0
Examples
--------
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0, 1, weight=2)
0
>>> G.add_edge(1, 0)
0
>>> G.add_edge(2, 2, weight=3)
0
>>> G.add_edge(2, 2)
1
>>> nx.to_pandas_adjacency(G, nodelist=[0, 1, 2], dtype=int)
   0  1  2
0  0  2  0
1  1  0  0
2  0  0  4
"""
import pandas as pd
M = to_numpy_array(
    G,
    nodelist=nodelist,
    dtype=dtype,
    order=order,
    multigraph_weight=multigraph_weight,
```

```
120: (8)                    weight=weight,
121: (8)                    nonedge=nonedge,
122: (4)                )
123: (4)                if nodelist is None:
124: (8)                    nodelist = list(G)
125: (4)                return pd.DataFrame(data=M, index=nodelist, columns=nodelist)
126: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
127: (0)            def from_pandas_adjacency(df, create_using=None):
128: (4)                r"""Returns a graph from Pandas DataFrame.
129: (4)                The Pandas DataFrame is interpreted as an adjacency matrix for the graph.
130: (4)                Parameters
131: (4)                ----------
132: (4)                df : Pandas DataFrame
133: (6)                  An adjacency matrix representation of a graph
134: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
135: (7)                   Graph type to create. If graph instance, then cleared before populated.
136: (4)                Notes
137: (4)                -----
138: (4)                For directed graphs, explicitly mention create_using=nx.DiGraph,
139: (4)                and entry i,j of df corresponds to an edge from i to j.
140: (4)                If `df` has a single data type for each entry it will be converted to an
141: (4)                appropriate Python data type.
142: (4)                If you have node attributes stored in a separate dataframe `df_nodes`,
143: (4)                you can load those attributes to the graph `G` using the following code:
144: (4)                ```
145: (4)                df_nodes = pd.DataFrame({"node_id": [1, 2, 3], "attribute1": ["A", "B",
"C"]})
146: (4)                G.add_nodes_from((n, dict(d)) for n, d in df_nodes.iterrows())
147: (4)                ```
148: (4)                If `df` has a user-specified compound data type the names
149: (4)                of the data fields will be used as attribute keys in the resulting
150: (4)                NetworkX graph.
151: (4)                See Also
152: (4)                --------
153: (4)                to_pandas_adjacency
154: (4)                Examples
155: (4)                --------
156: (4)                Simple integer weights on edges:
157: (4)                >>> import pandas as pd
158: (4)                >>> pd.options.display.max_columns = 20
159: (4)                >>> df = pd.DataFrame([[1, 1], [2, 1]])
160: (4)                >>> df
161: (7)                   0  1
162: (4)                0  1  1
163: (4)                1  2  1
164: (4)                >>> G = nx.from_pandas_adjacency(df)
165: (4)                >>> G.name = "Graph from pandas adjacency matrix"
166: (4)                >>> print(G)
167: (4)                Graph named 'Graph from pandas adjacency matrix' with 2 nodes and 3 edges
168: (4)                """
169: (4)                try:
170: (8)                    df = df[df.index]
171: (4)                except Exception as err:
172: (8)                    missing = list(set(df.index).difference(set(df.columns)))
173: (8)                    msg = f"{missing} not in columns"
174: (8)                    raise nx.NetworkXError("Columns must match Indices.", msg) from err
175: (4)                A = df.values
176: (4)                G = from_numpy_array(A, create_using=create_using)
177: (4)                nx.relabel.relabel_nodes(G, dict(enumerate(df.columns)), copy=False)
178: (4)                return G
179: (0)            @nx._dispatchable(preserve_edge_attrs=True)
180: (0)            def to_pandas_edgelist(
181: (4)                G,
182: (4)                source="source",
183: (4)                target="target",
184: (4)                nodelist=None,
185: (4)                dtype=None,
186: (4)                edge_key=None,
187: (0)            ):
```

```
(4)     """Returns the graph edge list as a Pandas DataFrame.
(4)     Parameters
(4)     ----------
(4)     G : graph
(8)         The NetworkX graph used to construct the Pandas DataFrame.
(4)     source : str or int, optional
(8)         A valid column name (string or integer) for the source nodes (for the
(8)         directed case).
(4)     target : str or int, optional
(8)         A valid column name (string or integer) for the target nodes (for the
(8)         directed case).
(4)     nodelist : list, optional
(7)        Use only nodes specified in nodelist
(4)     dtype : dtype, default None
(8)         Use to create the DataFrame. Data type to force.
(8)         Only a single dtype is allowed. If None, infer.
(4)     edge_key : str or int or None, optional (default=None)
(8)         A valid column name (string or integer) for the edge keys (for the
(8)         multigraph case). If None, edge keys are not stored in the DataFrame.
(4)     Returns
(4)     -------
(4)     df : Pandas DataFrame
(7)        Graph edge list
(4)     Examples
(4)     --------
(4)     >>> G = nx.Graph(
(4)     ...     [
(4)     ...         ("A", "B", {"cost": 1, "weight": 7}),
(4)     ...         ("C", "E", {"cost": 9, "weight": 10}),
(4)     ...     ]
(4)     ... )
(4)     >>> df = nx.to_pandas_edgelist(G, nodelist=["A", "C"])
(4)     >>> df[["source", "target", "cost", "weight"]]
(6)       source target  cost  weight
(4)     0      A      B     1       7
(4)     1      C      E     9      10
(4)     >>> G = nx.MultiGraph([("A", "B", {"cost": 1}), ("A", "B", {"cost": 9})])
(4)     >>> df = nx.to_pandas_edgelist(G, nodelist=["A", "C"], edge_key="ekey")
(4)     >>> df[["source", "target", "cost", "ekey"]]
(6)       source target  cost  ekey
(4)     0      A      B     1     0
(4)     1      A      B     9     1
(4)     """
(4)     import pandas as pd
(4)     if nodelist is None:
(8)         edgelist = G.edges(data=True)
(4)     else:
(8)         edgelist = G.edges(nodelist, data=True)
(4)     source_nodes = [s for s, _, _ in edgelist]
(4)     target_nodes = [t for _, t, _ in edgelist]
(4)     all_attrs = set().union(*(d.keys() for _, _, d in edgelist))
(4)     if source in all_attrs:
(8)         raise nx.NetworkXError(f"Source name {source!r} is an edge attr name")
(4)     if target in all_attrs:
(8)         raise nx.NetworkXError(f"Target name {target!r} is an edge attr name")
(4)     nan = float("nan")
(4)     edge_attr = {k: [d.get(k, nan) for _, _, d in edgelist] for k in
all_attrs}
(4)     if G.is_multigraph() and edge_key is not None:
(8)         if edge_key in all_attrs:
(12)            raise nx.NetworkXError(f"Edge key name {edge_key!r} is an edge
attr name")
(8)         edge_keys = [k for _, _, k in G.edges(keys=True)]
(8)         edgelistdict = {source: source_nodes, target: target_nodes, edge_key:
edge_keys}
(4)     else:
(8)         edgelistdict = {source: source_nodes, target: target_nodes}
(4)     edgelistdict.update(edge_attr)
(4)     return pd.DataFrame(edgelistdict, dtype=dtype)
```

```
(0)            @nx._dispatchable(graphs=None, returns_graph=True)
(0)            def from_pandas_edgelist(
(4)                df,
(4)                source="source",
(4)                target="target",
(4)                edge_attr=None,
(4)                create_using=None,
(4)                edge_key=None,
(0)            ):
(4)                """Returns a graph from Pandas DataFrame containing an edge list.
(4)                The Pandas DataFrame should contain at least two columns of node names and
(4)                zero or more columns of edge attributes. Each row will be processed as one
(4)                edge instance.
(4)                Note: This function iterates over DataFrame.values, which is not
(4)                guaranteed to retain the data type across columns in the row. This is only
(4)                a problem if your row is entirely numeric and a mix of ints and floats. In
(4)                that case, all values will be returned as floats. See the
(4)                DataFrame.iterrows documentation for an example.
(4)                Parameters
(4)                ----------
(4)                df : Pandas DataFrame
(8)                    An edge list representation of a graph
(4)                source : str or int
(8)                    A valid column name (string or integer) for the source nodes (for the
(8)                    directed case).
(4)                target : str or int
(8)                    A valid column name (string or integer) for the target nodes (for the
(8)                    directed case).
(4)                edge_attr : str or int, iterable, True, or None
(8)                    A valid column name (str or int) or iterable of column names that are
(8)                    used to retrieve items and add them to the graph as edge attributes.
(8)                    If `True`, all of the remaining columns will be added.
(8)                    If `None`, no edge attributes are added to the graph.
(4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
(8)                    Graph type to create. If graph instance, then cleared before
populated.
(4)                edge_key : str or None, optional (default=None)
(8)                    A valid column name for the edge keys (for a MultiGraph). The values
in
(8)                    this column are used for the edge keys when adding edges if
create_using
(8)                    is a multigraph.
(4)                If you have node attributes stored in a separate dataframe `df_nodes`,
(4)                you can load those attributes to the graph `G` using the following code:
(4)                ```
(4)                df_nodes = pd.DataFrame({"node_id": [1, 2, 3], "attribute1": ["A", "B",
"C"]})
(4)                G.add_nodes_from((n, dict(d)) for n, d in df_nodes.iterrows())
(4)                ```
(4)                See Also
(4)                --------
(4)                to_pandas_edgelist
(4)                Examples
(4)                --------
(4)                Simple integer weights on edges:
(4)                >>> import pandas as pd
(4)                >>> pd.options.display.max_columns = 20
(4)                >>> import numpy as np
(4)                >>> rng = np.random.RandomState(seed=5)
(4)                >>> ints = rng.randint(1, 11, size=(3, 2))
(4)                >>> a = ["A", "B", "C"]
(4)                >>> b = ["D", "A", "E"]
(4)                >>> df = pd.DataFrame(ints, columns=["weight", "cost"])
(4)                >>> df[0] = a
(4)                >>> df["b"] = b
(4)                >>> df[["weight", "cost", 0, "b"]]
(7)                   weight  cost  0  b
(4)                0       4     7  A  D
(4)                1       7     1  B  A
```

```
319: (4)                2      10     9  C  E
320: (4)                >>> G = nx.from_pandas_edgelist(df, 0, "b", ["weight", "cost"])
321: (4)                >>> G["E"]["C"]["weight"]
322: (4)                10
323: (4)                >>> G["E"]["C"]["cost"]
324: (4)                9
325: (4)                >>> edges = pd.DataFrame(
326: (4)                ...     {
327: (4)                ...         "source": [0, 1, 2],
328: (4)                ...         "target": [2, 2, 3],
329: (4)                ...         "weight": [3, 4, 5],
330: (4)                ...         "color": ["red", "blue", "blue"],
331: (4)                ...     }
332: (4)                ... )
333: (4)                >>> G = nx.from_pandas_edgelist(edges, edge_attr=True)
334: (4)                >>> G[0][2]["color"]
335: (4)                'red'
336: (4)                Build multigraph with custom keys:
337: (4)                >>> edges = pd.DataFrame(
338: (4)                ...     {
339: (4)                ...         "source": [0, 1, 2, 0],
340: (4)                ...         "target": [2, 2, 3, 2],
341: (4)                ...         "my_edge_key": ["A", "B", "C", "D"],
342: (4)                ...         "weight": [3, 4, 5, 6],
343: (4)                ...         "color": ["red", "blue", "blue", "blue"],
344: (4)                ...     }
345: (4)                ... )
346: (4)                >>> G = nx.from_pandas_edgelist(
347: (4)                ...     edges,
348: (4)                ...     edge_key="my_edge_key",
349: (4)                ...     edge_attr=["weight", "color"],
350: (4)                ...     create_using=nx.MultiGraph(),
351: (4)                ... )
352: (4)                >>> G[0][2]
353: (4)                AtlasView({'A': {'weight': 3, 'color': 'red'}, 'D': {'weight': 6, 'color':
'blue'}})
354: (4)                """
355: (4)                g = nx.empty_graph(0, create_using)
356: (4)                if edge_attr is None:
357: (8)                    g.add_edges_from(zip(df[source], df[target]))
358: (8)                    return g
359: (4)                reserved_columns = [source, target]
360: (4)                attr_col_headings = []
361: (4)                attribute_data = []
362: (4)                if edge_attr is True:
363: (8)                    attr_col_headings = [c for c in df.columns if c not in
reserved_columns]
364: (4)                elif isinstance(edge_attr, list | tuple):
365: (8)                    attr_col_headings = edge_attr
366: (4)                else:
367: (8)                    attr_col_headings = [edge_attr]
368: (4)                if len(attr_col_headings) == 0:
369: (8)                    raise nx.NetworkXError(
370: (12)                       f"Invalid edge_attr argument: No columns found with name:
{attr_col_headings}"
371: (8)                    )
372: (4)                try:
373: (8)                    attribute_data = zip(*[df[col] for col in attr_col_headings])
374: (4)                except (KeyError, TypeError) as err:
375: (8)                    msg = f"Invalid edge_attr argument: {edge_attr}"
376: (8)                    raise nx.NetworkXError(msg) from err
377: (4)                if g.is_multigraph():
378: (8)                    if edge_key is not None:
379: (12)                       try:
380: (16)                           multigraph_edge_keys = df[edge_key]
381: (16)                           attribute_data = zip(attribute_data, multigraph_edge_keys)
382: (12)                       except (KeyError, TypeError) as err:
383: (16)                           msg = f"Invalid edge_key argument: {edge_key}"
384: (16)                           raise nx.NetworkXError(msg) from err
```

```
385: (8)                    for s, t, attrs in zip(df[source], df[target], attribute_data):
386: (12)                       if edge_key is not None:
387: (16)                           attrs, multigraph_edge_key = attrs
388: (16)                           key = g.add_edge(s, t, key=multigraph_edge_key)
389: (12)                       else:
390: (16)                           key = g.add_edge(s, t)
391: (12)                       g[s][t][key].update(zip(attr_col_headings, attrs))
392: (4)                else:
393: (8)                    for s, t, attrs in zip(df[source], df[target], attribute_data):
394: (12)                       g.add_edge(s, t)
395: (12)                       g[s][t].update(zip(attr_col_headings, attrs))
396: (4)                return g
397: (0)            @nx._dispatchable(edge_attrs="weight")
398: (0)            def to_scipy_sparse_array(G, nodelist=None, dtype=None, weight="weight",
format="csr"):
399: (4)                """Returns the graph adjacency matrix as a SciPy sparse array.
400: (4)                Parameters
401: (4)                ----------
402: (4)                G : graph
403: (8)                    The NetworkX graph used to construct the sparse matrix.
404: (4)                nodelist : list, optional
405: (7)                   The rows and columns are ordered according to the nodes in `nodelist`.
406: (7)                   If `nodelist` is None, then the ordering is produced by G.nodes().
407: (4)                dtype : NumPy data-type, optional
408: (8)                    A valid NumPy dtype used to initialize the array. If None, then the
409: (8)                    NumPy default is used.
410: (4)                weight : string or None   optional (default='weight')
411: (8)                    The edge attribute that holds the numerical value used for
412: (8)                    the edge weight.  If None then all edge weights are 1.
413: (4)                format : str in {'bsr', 'csr', 'csc', 'coo', 'lil', 'dia', 'dok'}
414: (8)                    The type of the matrix to be returned (default 'csr').  For
415: (8)                    some algorithms different implementations of sparse matrices
416: (8)                    can perform better.  See [1]_ for details.
417: (4)                Returns
418: (4)                -------
419: (4)                A : SciPy sparse array
420: (7)                   Graph adjacency matrix.
421: (4)                Notes
422: (4)                -----
423: (4)                For directed graphs, matrix entry i,j corresponds to an edge from i to j.
424: (4)                The matrix entries are populated using the edge attribute held in
425: (4)                parameter weight. When an edge does not have that attribute, the
426: (4)                value of the entry is 1.
427: (4)                For multiple edges the matrix values are the sums of the edge weights.
428: (4)                When `nodelist` does not contain every node in `G`, the adjacency matrix
429: (4)                is built from the subgraph of `G` that is induced by the nodes in
430: (4)                `nodelist`.
431: (4)                The convention used for self-loop edges in graphs is to assign the
432: (4)                diagonal matrix entry value to the weight attribute of the edge
433: (4)                (or the number 1 if the edge has no weight attribute).  If the
434: (4)                alternate convention of doubling the edge weight is desired the
435: (4)                resulting SciPy sparse array can be modified as follows:
436: (4)                >>> G = nx.Graph([(1, 1)])
437: (4)                >>> A = nx.to_scipy_sparse_array(G)
438: (4)                >>> print(A.todense())
439: (4)                [[1]]
440: (4)                >>> A.setdiag(A.diagonal() * 2)
441: (4)                >>> print(A.toarray())
442: (4)                [[2]]
443: (4)                Examples
444: (4)                --------
445: (4)                >>> G = nx.MultiDiGraph()
446: (4)                >>> G.add_edge(0, 1, weight=2)
447: (4)                0
448: (4)                >>> G.add_edge(1, 0)
449: (4)                0
450: (4)                >>> G.add_edge(2, 2, weight=3)
451: (4)                0
452: (4)                >>> G.add_edge(2, 2)
```

```
453: (4)                1
454: (4)                >>> S = nx.to_scipy_sparse_array(G, nodelist=[0, 1, 2])
455: (4)                >>> print(S.toarray())
456: (4)                [[0 2 0]
457: (5)                 [1 0 0]
458: (5)                 [0 0 4]]
459: (4)                References
460: (4)                ----------
461: (4)                .. [1] Scipy Dev. References, "Sparse Matrices",
462: (7)                   https://docs.scipy.org/doc/scipy/reference/sparse.html
463: (4)                """
464: (4)                import scipy as sp
465: (4)                if len(G) == 0:
466: (8)                    raise nx.NetworkXError("Graph has no nodes or edges")
467: (4)                if nodelist is None:
468: (8)                    nodelist = list(G)
469: (8)                    nlen = len(G)
470: (4)                else:
471: (8)                    nlen = len(nodelist)
472: (8)                    if nlen == 0:
473: (12)                       raise nx.NetworkXError("nodelist has no nodes")
474: (8)                    nodeset = set(G.nbunch_iter(nodelist))
475: (8)                    if nlen != len(nodeset):
476: (12)                       for n in nodelist:
477: (16)                           if n not in G:
478: (20)                               raise nx.NetworkXError(f"Node {n} in nodelist is not in
G")
479: (12)                       raise nx.NetworkXError("nodelist contains duplicates.")
480: (8)                    if nlen < len(G):
481: (12)                       G = G.subgraph(nodelist)
482: (4)                index = dict(zip(nodelist, range(nlen)))
483: (4)                coefficients = zip(
484: (8)                    *((index[u], index[v], wt) for u, v, wt in G.edges(data=weight,
default=1))
485: (4)                )
486: (4)                try:
487: (8)                    row, col, data = coefficients
488: (4)                except ValueError:
489: (8)                    row, col, data = [], [], []
490: (4)                if G.is_directed():
491: (8)                    A = sp.sparse.coo_array((data, (row, col)), shape=(nlen, nlen),
dtype=dtype)
492: (4)                else:
493: (8)                    d = data + data
494: (8)                    r = row + col
495: (8)                    c = col + row
496: (8)                    selfloops = list(nx.selfloop_edges(G, data=weight, default=1))
497: (8)                    if selfloops:
498: (12)                       diag_index, diag_data = zip(*((index[u], -wt) for u, v, wt in
selfloops))
499: (12)                       d += diag_data
500: (12)                       r += diag_index
501: (12)                       c += diag_index
502: (8)                    A = sp.sparse.coo_array((d, (r, c)), shape=(nlen, nlen), dtype=dtype)
503: (4)                try:
504: (8)                    return A.asformat(format)
505: (4)                except ValueError as err:
506: (8)                    raise nx.NetworkXError(f"Unknown sparse matrix format: {format}") from
err
507: (0)            def _csr_gen_triples(A):
508: (4)                """Converts a SciPy sparse array in **Compressed Sparse Row** format to
509: (4)                an iterable of weighted edge triples.
510: (4)                """
511: (4)                nrows = A.shape[0]
512: (4)                indptr, dst_indices, data = A.indptr, A.indices, A.data
513: (4)                import numpy as np
514: (4)                src_indices = np.repeat(np.arange(nrows), np.diff(indptr))
515: (4)                return zip(src_indices.tolist(), dst_indices.tolist(), A.data.tolist())
516: (0)            def _csc_gen_triples(A):
```

```
(4)                 """Converts a SciPy sparse array in **Compressed Sparse Column** format to
(4)                 an iterable of weighted edge triples.
(4)                 """
(4)                 ncols = A.shape[1]
(4)                 indptr, src_indices, data = A.indptr, A.indices, A.data
(4)                 import numpy as np
(4)                 dst_indices = np.repeat(np.arange(ncols), np.diff(indptr))
(4)                 return zip(src_indices.tolist(), dst_indices.tolist(), A.data.tolist())
(0)             def _coo_gen_triples(A):
(4)                 """Converts a SciPy sparse array in **Coordinate** format to an iterable
(4)                 of weighted edge triples.
(4)                 """
(4)                 return zip(A.row.tolist(), A.col.tolist(), A.data.tolist())
(0)             def _dok_gen_triples(A):
(4)                 """Converts a SciPy sparse array in **Dictionary of Keys** format to an
(4)                 iterable of weighted edge triples.
(4)                 """
(4)                 for (r, c), v in A.items():
(8)                     yield int(r), int(c), v.item()
(0)             def _generate_weighted_edges(A):
(4)                 """Returns an iterable over (u, v, w) triples, where u and v are adjacent
(4)                 vertices and w is the weight of the edge joining u and v.
(4)                 `A` is a SciPy sparse array (in any format).
(4)                 """
(4)                 if A.format == "csr":
(8)                     return _csr_gen_triples(A)
(4)                 if A.format == "csc":
(8)                     return _csc_gen_triples(A)
(4)                 if A.format == "dok":
(8)                     return _dok_gen_triples(A)
(4)                 return _coo_gen_triples(A.tocoo())
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def from_scipy_sparse_array(
(4)                 A, parallel_edges=False, create_using=None, edge_attribute="weight"
(0)             ):
(4)                 """Creates a new graph from an adjacency matrix given as a SciPy sparse
(4)                 array.
(4)                 Parameters
(4)                 ----------
(4)                 A: scipy.sparse array
(6)                   An adjacency matrix representation of a graph
(4)                 parallel_edges : Boolean
(6)                   If this is True, `create_using` is a multigraph, and `A` is an
(6)                   integer matrix, then entry *(i, j)* in the matrix is interpreted as the
(6)                   number of parallel edges joining vertices *i* and *j* in the graph.
(6)                   If it is False, then the entries in the matrix are interpreted as
(6)                   the weight of a single edge joining the vertices.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 edge_attribute: string
(7)                    Name of edge attribute to store matrix numeric value. The data will
(7)                    have the same type as the matrix entry (int, float, (real,imag)).
(4)                 Notes
(4)                 -----
(4)                 For directed graphs, explicitly mention create_using=nx.DiGraph,
(4)                 and entry i,j of A corresponds to an edge from i to j.
(4)                 If `create_using` is :class:`networkx.MultiGraph` or
(4)                 :class:`networkx.MultiDiGraph`, `parallel_edges` is True, and the
(4)                 entries of `A` are of type :class:`int`, then this function returns a
(4)                 multigraph (constructed from `create_using`) with parallel edges.
(4)                 In this case, `edge_attribute` will be ignored.
(4)                 If `create_using` indicates an undirected multigraph, then only the edges
(4)                 indicated by the upper triangle of the matrix `A` will be added to the
(4)                 graph.
(4)                 Examples
(4)                 --------
(4)                 >>> import scipy as sp
(4)                 >>> A = sp.sparse.eye(2, 2, 1)
(4)                 >>> G = nx.from_scipy_sparse_array(A)
```

```
586: (4)                If `create_using` indicates a multigraph and the matrix has only integer
587: (4)                entries and `parallel_edges` is False, then the entries will be treated
588: (4)                as weights for edges joining the nodes (without creating parallel edges):
589: (4)                >>> A = sp.sparse.csr_array([[1, 1], [1, 2]])
590: (4)                >>> G = nx.from_scipy_sparse_array(A, create_using=nx.MultiGraph)
591: (4)                >>> G[1][1]
592: (4)                AtlasView({0: {'weight': 2}})
593: (4)                If `create_using` indicates a multigraph and the matrix has only integer
594: (4)                entries and `parallel_edges` is True, then the entries will be treated
595: (4)                as the number of parallel edges joining those two vertices:
596: (4)                >>> A = sp.sparse.csr_array([[1, 1], [1, 2]])
597: (4)                >>> G = nx.from_scipy_sparse_array(A, parallel_edges=True,
create_using=nx.MultiGraph)
598: (4)                >>> G[1][1]
599: (4)                AtlasView({0: {'weight': 1}, 1: {'weight': 1}})
600: (4)                """
601: (4)                G = nx.empty_graph(0, create_using)
602: (4)                n, m = A.shape
603: (4)                if n != m:
604: (8)                    raise nx.NetworkXError(f"Adjacency matrix not square: nx,ny=
{A.shape}")
605: (4)                G.add_nodes_from(range(n))
606: (4)                triples = _generate_weighted_edges(A)
607: (4)                if A.dtype.kind in ("i", "u") and G.is_multigraph() and parallel_edges:
608: (8)                    chain = itertools.chain.from_iterable
609: (8)                    triples = chain(((u, v, 1) for d in range(w)) for (u, v, w) in
triples)
610: (4)                if G.is_multigraph() and not G.is_directed():
611: (8)                    triples = ((u, v, d) for u, v, d in triples if u <= v)
612: (4)                G.add_weighted_edges_from(triples, weight=edge_attribute)
613: (4)                return G
614: (0)            @nx._dispatchable(edge_attrs="weight")  # edge attrs may also be obtained from
`dtype`
615: (0)            def to_numpy_array(
616: (4)                G,
617: (4)                nodelist=None,
618: (4)                dtype=None,
619: (4)                order=None,
620: (4)                multigraph_weight=sum,
621: (4)                weight="weight",
622: (4)                nonedge=0.0,
623: (0)            ):
624: (4)                """Returns the graph adjacency matrix as a NumPy array.
625: (4)                Parameters
626: (4)                ----------
627: (4)                G : graph
628: (8)                    The NetworkX graph used to construct the NumPy array.
629: (4)                nodelist : list, optional
630: (8)                    The rows and columns are ordered according to the nodes in `nodelist`.
631: (8)                    If `nodelist` is ``None``, then the ordering is produced by
``G.nodes()``.
632: (4)                dtype : NumPy data type, optional
633: (8)                    A NumPy data type used to initialize the array. If None, then the
NumPy
634: (8)                    default is used. The dtype can be structured if `weight=None`, in
which
635: (8)                    case the dtype field names are used to look up edge attributes. The
636: (8)                    result is a structured array where each named field in the dtype
637: (8)                    corresponds to the adjacency for that edge attribute. See examples for
638: (8)                    details.
639: (4)                order : {'C', 'F'}, optional
640: (8)                    Whether to store multidimensional data in C- or Fortran-contiguous
641: (8)                    (row- or column-wise) order in memory. If None, then the NumPy default
642: (8)                    is used.
643: (4)                multigraph_weight : callable, optional
644: (8)                    An function that determines how weights in multigraphs are handled.
645: (8)                    The function should accept a sequence of weights and return a single
646: (8)                    value. The default is to sum the weights of the multiple edges.
647: (4)                weight : string or None optional (default = 'weight')
```

```
(8)                 The edge attribute that holds the numerical value used for
(8)                 the edge weight. If an edge does not have that attribute, then the
(8)                 value 1 is used instead. `weight` must be ``None`` if a structured
(8)                 dtype is used.
(4)             nonedge : array_like (default = 0.0)
(8)                 The value used to represent non-edges in the adjacency matrix.
(8)                 The array values corresponding to nonedges are typically set to zero.
(8)                 However, this could be undesirable if there are array values
(8)                 corresponding to actual edges that also have the value zero. If so,
(8)                 one might prefer nonedges to have some other value, such as ``nan``.
(4)             Returns
(4)             -------
(4)             A : NumPy ndarray
(8)                 Graph adjacency matrix
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If `dtype` is a structured dtype and `G` is a multigraph
(4)             ValueError
(8)                 If `dtype` is a structured dtype and `weight` is not `None`
(4)             See Also
(4)             --------
(4)             from_numpy_array
(4)             Notes
(4)             -----
(4)             For directed graphs, entry ``i, j`` corresponds to an edge from ``i`` to
``j``.
(4)             Entries in the adjacency matrix are given by the `weight` edge attribute.
(4)             When an edge does not have a weight attribute, the value of the entry is
(4)             set to the number 1.  For multiple (parallel) edges, the values of the
(4)             entries are determined by the `multigraph_weight` parameter. The default
is
(4)             to sum the weight attributes for each of the parallel edges.
(4)             When `nodelist` does not contain every node in `G`, the adjacency matrix
is
(4)             built from the subgraph of `G` that is induced by the nodes in `nodelist`.
(4)             The convention used for self-loop edges in graphs is to assign the
(4)             diagonal array entry value to the weight attribute of the edge
(4)             (or the number 1 if the edge has no weight attribute). If the
(4)             alternate convention of doubling the edge weight is desired the
(4)             resulting NumPy array can be modified as follows:
(4)             >>> import numpy as np
(4)             >>> G = nx.Graph([(1, 1)])
(4)             >>> A = nx.to_numpy_array(G)
(4)             >>> A
(4)             array([[1.]])
(4)             >>> A[np.diag_indices_from(A)] *= 2
(4)             >>> A
(4)             array([[2.]])
(4)             Examples
(4)             --------
(4)             >>> G = nx.MultiDiGraph()
(4)             >>> G.add_edge(0, 1, weight=2)
(4)             0
(4)             >>> G.add_edge(1, 0)
(4)             0
(4)             >>> G.add_edge(2, 2, weight=3)
(4)             0
(4)             >>> G.add_edge(2, 2)
(4)             1
(4)             >>> nx.to_numpy_array(G, nodelist=[0, 1, 2])
(4)             array([[0., 2., 0.],
(11)                   [1., 0., 0.],
(11)                   [0., 0., 4.]])
(4)             When `nodelist` argument is used, nodes of `G` which do not appear in the
`nodelist`
(4)             and their edges are not included in the adjacency matrix. Here is an
example:
(4)             >>> G = nx.Graph()
```

```
(4)                 >>> G.add_edge(3, 1)
(4)                 >>> G.add_edge(2, 0)
(4)                 >>> G.add_edge(2, 1)
(4)                 >>> G.add_edge(3, 0)
(4)                 >>> nx.to_numpy_array(G, nodelist=[1, 2, 3])
(4)                 array([[0., 1., 1.],
(11)                       [1., 0., 0.],
(11)                       [1., 0., 0.]])
(4)                 This function can also be used to create adjacency matrices for multiple
(4)                 edge attributes with structured dtypes:
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edge(0, 1, weight=10)
(4)                 >>> G.add_edge(1, 2, cost=5)
(4)                 >>> G.add_edge(2, 3, weight=3, cost=-4.0)
(4)                 >>> dtype = np.dtype([("weight", int), ("cost", float)])
(4)                 >>> A = nx.to_numpy_array(G, dtype=dtype, weight=None)
(4)                 >>> A["weight"]
(4)                 array([[ 0, 10,  0,  0],
(11)                       [10,  0,  1,  0],
(11)                       [ 0,  1,  0,  3],
(11)                       [ 0,  0,  3,  0]])
(4)                 >>> A["cost"]
(4)                 array([[ 0.,  1.,  0.,  0.],
(11)                       [ 1.,  0.,  5.,  0.],
(11)                       [ 0.,  5.,  0., -4.],
(11)                       [ 0.,  0., -4.,  0.]])
(4)                 As stated above, the argument "nonedge" is useful especially when there
are
(4)                 actually edges with weight 0 in the graph. Setting a nonedge value
different than 0,
(4)                 makes it much clearer to differentiate such 0-weighted edges and actual
nonedge values.
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edge(3, 1, weight=2)
(4)                 >>> G.add_edge(2, 0, weight=0)
(4)                 >>> G.add_edge(2, 1, weight=0)
(4)                 >>> G.add_edge(3, 0, weight=1)
(4)                 >>> nx.to_numpy_array(G, nonedge=-1.0)
(4)                 array([[-1.,  2., -1.,  1.],
(11)                       [ 2., -1.,  0., -1.],
(11)                       [-1.,  0., -1.,  0.],
(11)                       [ 1., -1.,  0., -1.]])
(4)                 """
(4)                 import numpy as np
(4)                 if nodelist is None:
(8)                     nodelist = list(G)
(4)                 nlen = len(nodelist)
(4)                 nodeset = set(nodelist)
(4)                 if nodeset - set(G):
(8)                     raise nx.NetworkXError(f"Nodes {nodeset - set(G)} in nodelist is not
in G")
(4)                 if len(nodeset) < nlen:
(8)                     raise nx.NetworkXError("nodelist contains duplicates.")
(4)                 A = np.full((nlen, nlen), fill_value=nonedge, dtype=dtype, order=order)
(4)                 if nlen == 0 or G.number_of_edges() == 0:
(8)                     return A
(4)                 edge_attrs = None  # Only single edge attribute by default
(4)                 if A.dtype.names:
(8)                     if weight is None:
(12)                        edge_attrs = dtype.names
(8)                     else:
(12)                        raise ValueError(
(16)                            "Specifying `weight` not supported for structured dtypes\n."
(16)                            "To create adjacency matrices from structured dtypes, use
`weight=None`."
(12)                        )
(4)                 idx = dict(zip(nodelist, range(nlen)))
(4)                 if len(nodelist) < len(G):
(8)                     G = G.subgraph(nodelist).copy()
```

```
(4)                 if G.is_multigraph():
(8)                     if edge_attrs:
(12)                        raise nx.NetworkXError(
(16)                            "Structured arrays are not supported for MultiGraphs"
(12)                        )
(8)                     d = defaultdict(list)
(8)                     for u, v, wt in G.edges(data=weight, default=1.0):
(12)                        d[(idx[u], idx[v])].append(wt)
(8)                     i, j = np.array(list(d.keys())).T  # indices
(8)                     wts = [multigraph_weight(ws) for ws in d.values()]  # reduced weights
(4)                 else:
(8)                     i, j, wts = [], [], []
(8)                     if edge_attrs:
(12)                        for u, v, data in G.edges(data=True):
(16)                            i.append(idx[u])
(16)                            j.append(idx[v])
(16)                            wts.append(data)
(12)                        for attr in edge_attrs:
(16)                            attr_data = [wt.get(attr, 1.0) for wt in wts]
(16)                            A[attr][i, j] = attr_data
(16)                            if not G.is_directed():
(20)                                A[attr][j, i] = attr_data
(12)                        return A
(8)                     for u, v, wt in G.edges(data=weight, default=1.0):
(12)                        i.append(idx[u])
(12)                        j.append(idx[v])
(12)                        wts.append(wt)
(4)                 A[i, j] = wts
(4)                 if not G.is_directed():
(8)                     A[j, i] = wts
(4)                 return A
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def from_numpy_array(A, parallel_edges=False, create_using=None,
edge_attr="weight"):
(4)                 """Returns a graph from a 2D NumPy array.
(4)                 The 2D NumPy array is interpreted as an adjacency matrix for the graph.
(4)                 Parameters
(4)                 ----------
(4)                 A : a 2D numpy.ndarray
(8)                     An adjacency matrix representation of a graph
(4)                 parallel_edges : Boolean
(8)                     If this is True, `create_using` is a multigraph, and `A` is an
(8)                     integer array, then entry *(i, j)* in the array is interpreted as the
(8)                     number of parallel edges joining vertices *i* and *j* in the graph.
(8)                     If it is False, then the entries in the array are interpreted as
(8)                     the weight of a single edge joining the vertices.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 edge_attr : String, optional (default="weight")
(8)                     The attribute to which the array values are assigned on each edge. If
(8)                     it is None, edge attributes will not be assigned.
(4)                 Notes
(4)                 -----
(4)                 For directed graphs, explicitly mention create_using=nx.DiGraph,
(4)                 and entry i,j of A corresponds to an edge from i to j.
(4)                 If `create_using` is :class:`networkx.MultiGraph` or
(4)                 :class:`networkx.MultiDiGraph`, `parallel_edges` is True, and the
(4)                 entries of `A` are of type :class:`int`, then this function returns a
(4)                 multigraph (of the same type as `create_using`) with parallel edges.
(4)                 If `create_using` indicates an undirected multigraph, then only the edges
(4)                 indicated by the upper triangle of the array `A` will be added to the
(4)                 graph.
(4)                 If `edge_attr` is Falsy (False or None), edge attributes will not be
(4)                 assigned, and the array data will be treated like a binary mask of
(4)                 edge presence or absence. Otherwise, the attributes will be assigned
(4)                 as follows:
(4)                 If the NumPy array has a single data type for each array entry it
(4)                 will be converted to an appropriate Python data type.
(4)                 If the NumPy array has a user-specified compound data type the names
```

```
    of the data fields will be used as attribute keys in the resulting
    NetworkX graph.
    See Also
    --------
    to_numpy_array
    Examples
    --------
    Simple integer weights on edges:
    >>> import numpy as np
    >>> A = np.array([[1, 1], [2, 1]])
    >>> G = nx.from_numpy_array(A)
    >>> G.edges(data=True)
    EdgeDataView([(0, 0, {'weight': 1}), (0, 1, {'weight': 2}), (1, 1, {'weight': 1})])
    If `create_using` indicates a multigraph and the array has only integer
    entries and `parallel_edges` is False, then the entries will be treated
    as weights for edges joining the nodes (without creating parallel edges):
    >>> A = np.array([[1, 1], [1, 2]])
    >>> G = nx.from_numpy_array(A, create_using=nx.MultiGraph)
    >>> G[1][1]
    AtlasView({0: {'weight': 2}})
    If `create_using` indicates a multigraph and the array has only integer
    entries and `parallel_edges` is True, then the entries will be treated
    as the number of parallel edges joining those two vertices:
    >>> A = np.array([[1, 1], [1, 2]])
    >>> temp = nx.MultiGraph()
    >>> G = nx.from_numpy_array(A, parallel_edges=True, create_using=temp)
    >>> G[1][1]
    AtlasView({0: {'weight': 1}, 1: {'weight': 1}})
    User defined compound data type on edges:
    >>> dt = [("weight", float), ("cost", int)]
    >>> A = np.array([[(1.0, 2)]], dtype=dt)
    >>> G = nx.from_numpy_array(A)
    >>> G.edges()
    EdgeView([(0, 0)])
    >>> G[0][0]["cost"]
    2
    >>> G[0][0]["weight"]
    1.0
    """
    kind_to_python_type = {
        "f": float,
        "i": int,
        "u": int,
        "b": bool,
        "c": complex,
        "S": str,
        "U": str,
        "V": "void",
    }
    G = nx.empty_graph(0, create_using)
    if A.ndim != 2:
        raise nx.NetworkXError(f"Input array must be 2D, not {A.ndim}")
    n, m = A.shape
    if n != m:
        raise nx.NetworkXError(f"Adjacency matrix not square: nx,ny={A.shape}")
    dt = A.dtype
    try:
        python_type = kind_to_python_type[dt.kind]
    except Exception as err:
        raise TypeError(f"Unknown numpy data type: {dt}") from err
    G.add_nodes_from(range(n))
    edges = ((int(e[0]), int(e[1])) for e in zip(*A.nonzero()))
    if python_type == "void":
        fields = sorted(
            (offset, dtype, name) for name, (dtype, offset) in A.dtype.fields.items()
        )
```

```
910: (8)                    triples = (
911: (12)                       (
912: (16)                           u,
913: (16)                           v,
914: (16)                           {}
915: (16)                           if edge_attr in [False, None]
916: (16)                           else {
917: (20)                               name: kind_to_python_type[dtype.kind](val)
918: (20)                               for (_, dtype, name), val in zip(fields, A[u, v])
919: (16)                           },
920: (12)                       )
921: (12)                       for u, v in edges
922: (8)                    )
923: (4)                elif python_type is int and G.is_multigraph() and parallel_edges:
924: (8)                    chain = itertools.chain.from_iterable
925: (8)                    if edge_attr in [False, None]:
926: (12)                       triples = chain(((u, v, {}) for d in range(A[u, v])) for (u, v) in
edges)
927: (8)                    else:
928: (12)                       triples = chain(
929: (16)                           ((u, v, {edge_attr: 1}) for d in range(A[u, v])) for (u, v) in
edges
930: (12)                       )
931: (4)                else:  # basic data type
932: (8)                    if edge_attr in [False, None]:
933: (12)                       triples = ((u, v, {}) for u, v in edges)
934: (8)                    else:
935: (12)                       triples = ((u, v, {edge_attr: python_type(A[u, v])}) for u, v in
edges)
936: (4)                if G.is_multigraph() and not G.is_directed():
937: (8)                    triples = ((u, v, d) for u, v, d in triples if u <= v)
938: (4)                G.add_edges_from(triples)
939: (4)                return G
```

File 5 - exception.py:

```
1: (0)              """
2: (0)              **********
3: (0)              Exceptions
4: (0)              **********
5: (0)              Base exceptions and errors for NetworkX.
6: (0)              """
7: (0)              __all__ = [
8: (4)                  "HasACycle",
9: (4)                  "NodeNotFound",
10: (4)                 "PowerIterationFailedConvergence",
11: (4)                 "ExceededMaxIterations",
12: (4)                 "AmbiguousSolution",
13: (4)                 "NetworkXAlgorithmError",
14: (4)                 "NetworkXException",
15: (4)                 "NetworkXError",
16: (4)                 "NetworkXNoCycle",
17: (4)                 "NetworkXNoPath",
18: (4)                 "NetworkXNotImplemented",
19: (4)                 "NetworkXPointlessConcept",
20: (4)                 "NetworkXUnbounded",
21: (4)                 "NetworkXUnfeasible",
22: (0)             ]
23: (0)             class NetworkXException(Exception):
24: (4)                 """Base class for exceptions in NetworkX."""
25: (0)             class NetworkXError(NetworkXException):
26: (4)                 """Exception for a serious error in NetworkX"""
27: (0)             class NetworkXPointlessConcept(NetworkXException):
28: (4)                 """Raised when a null graph is provided as input to an algorithm
29: (4)                 that cannot use it.
30: (4)                 The null graph is sometimes considered a pointless concept [1]_,
31: (4)                 thus the name of the exception.
```

```
32: (4)                 References
33: (4)                 ----------
34: (4)                 .. [1] Harary, F. and Read, R. "Is the Null Graph a Pointless
35: (7)                    Concept?"  In Graphs and Combinatorics Conference, George
36: (7)                    Washington University.  New York: Springer-Verlag, 1973.
37: (4)                 """
38: (0)             class NetworkXAlgorithmError(NetworkXException):
39: (4)                 """Exception for unexpected termination of algorithms."""
40: (0)             class NetworkXUnfeasible(NetworkXAlgorithmError):
41: (4)                 """Exception raised by algorithms trying to solve a problem
42: (4)                 instance that has no feasible solution."""
43: (0)             class NetworkXNoPath(NetworkXUnfeasible):
44: (4)                 """Exception for algorithms that should return a path when running
45: (4)                 on graphs where such a path does not exist."""
46: (0)             class NetworkXNoCycle(NetworkXUnfeasible):
47: (4)                 """Exception for algorithms that should return a cycle when running
48: (4)                 on graphs where such a cycle does not exist."""
49: (0)             class HasACycle(NetworkXException):
50: (4)                 """Raised if a graph has a cycle when an algorithm expects that it
51: (4)                 will have no cycles.
52: (4)                 """
53: (0)             class NetworkXUnbounded(NetworkXAlgorithmError):
54: (4)                 """Exception raised by algorithms trying to solve a maximization
55: (4)                 or a minimization problem instance that is unbounded."""
56: (0)             class NetworkXNotImplemented(NetworkXException):
57: (4)                 """Exception raised by algorithms not implemented for a type of graph."""
58: (0)             class NodeNotFound(NetworkXException):
59: (4)                 """Exception raised if requested node is not present in the graph"""
60: (0)             class AmbiguousSolution(NetworkXException):
61: (4)                 """Raised if more than one valid solution exists for an intermediary step
62: (4)                 of an algorithm.
63: (4)                 In the face of ambiguity, refuse the temptation to guess.
64: (4)                 This may occur, for example, when trying to determine the
65: (4)                 bipartite node sets in a disconnected bipartite graph when
66: (4)                 computing bipartite matchings.
67: (4)                 """
68: (0)             class ExceededMaxIterations(NetworkXException):
69: (4)                 """Raised if a loop iterates too many times without breaking.
70: (4)                 This may occur, for example, in an algorithm that computes
71: (4)                 progressively better approximations to a value but exceeds an
72: (4)                 iteration bound specified by the user.
73: (4)                 """
74: (0)             class PowerIterationFailedConvergence(ExceededMaxIterations):
75: (4)                 """Raised when the power iteration method fails to converge within a
76: (4)                 specified iteration limit.
77: (4)                 `num_iterations` is the number of iterations that have been
78: (4)                 completed when this exception was raised.
79: (4)                 """
80: (4)                 def __init__(self, num_iterations, *args, **kw):
81: (8)                     msg = f"power iteration failed to converge within {num_iterations}
iterations"
82: (8)                     exception_message = msg
83: (8)                     superinit = super().__init__
84: (8)                     superinit(self, exception_message, *args, **kw)
```

File 6 - lazy_imports.py:

```
1: (0)              import importlib
2: (0)              import importlib.util
3: (0)              import inspect
4: (0)              import os
5: (0)              import sys
6: (0)              import types
7: (0)              __all__ = ["attach", "_lazy_import"]
8: (0)              def attach(module_name, submodules=None, submod_attrs=None):
9: (4)                  """Attach lazily loaded submodules, and functions or other attributes.
10: (4)                 Typically, modules import submodules and attributes as follows::
```

```
 11: (6)                    import mysubmodule
 12: (6)                    import anothersubmodule
 13: (6)                    from .foo import someattr
 14: (4)                  The idea of  this function is to replace the `__init__.py`
 15: (4)                  module's `__getattr__`, `__dir__`, and `__all__` attributes such that
 16: (4)                  all imports work exactly the way they normally would, except that the
 17: (4)                  actual import is delayed until the resulting module object is first used.
 18: (4)                  The typical way to call this function, replacing the above imports, is::
 19: (6)                    __getattr__, __lazy_dir__, __all__ = lazy.attach(
 20: (10)                       __name__, ["mysubmodule", "anothersubmodule"], {"foo": "someattr"}
 21: (6)                    )
 22: (4)                  This functionality requires Python 3.7 or higher.
 23: (4)                  Parameters
 24: (4)                  ----------
 25: (4)                  module_name : str
 26: (8)                      Typically use __name__.
 27: (4)                  submodules : set
 28: (8)                      List of submodules to lazily import.
 29: (4)                  submod_attrs : dict
 30: (8)                      Dictionary of submodule -> list of attributes / functions.
 31: (8)                      These attributes are imported as they are used.
 32: (4)                  Returns
 33: (4)                  -------
 34: (4)                  __getattr__, __dir__, __all__
 35: (4)                  """
 36: (4)                  if submod_attrs is None:
 37: (8)                      submod_attrs = {}
 38: (4)                  if submodules is None:
 39: (8)                      submodules = set()
 40: (4)                  else:
 41: (8)                      submodules = set(submodules)
 42: (4)                  attr_to_modules = {
 43: (8)                      attr: mod for mod, attrs in submod_attrs.items() for attr in attrs
 44: (4)                  }
 45: (4)                  __all__ = list(submodules | attr_to_modules.keys())
 46: (4)                  def __getattr__(name):
 47: (8)                      if name in submodules:
 48: (12)                         return importlib.import_module(f"{module_name}.{name}")
 49: (8)                      elif name in attr_to_modules:
 50: (12)                         submod = importlib.import_module(f"{module_name}.
{attr_to_modules[name]}")
 51: (12)                         return getattr(submod, name)
 52: (8)                      else:
 53: (12)                         raise AttributeError(f"No {module_name} attribute {name}")
 54: (4)                  def __dir__():
 55: (8)                      return __all__
 56: (4)                  if os.environ.get("EAGER_IMPORT", ""):
 57: (8)                      for attr in set(attr_to_modules.keys()) | submodules:
 58: (12)                         __getattr__(attr)
 59: (4)                  return __getattr__, __dir__, list(__all__)
 60: (0)              class DelayedImportErrorModule(types.ModuleType):
 61: (4)                  def __init__(self, frame_data, *args, **kwargs):
 62: (8)                      self.__frame_data = frame_data
 63: (8)                      super().__init__(*args, **kwargs)
 64: (4)                  def __getattr__(self, x):
 65: (8)                      if x in ("__class__", "__file__", "__frame_data"):
 66: (12)                         super().__getattr__(x)
 67: (8)                      else:
 68: (12)                         fd = self.__frame_data
 69: (12)                         raise ModuleNotFoundError(
 70: (16)                             f"No module named '{fd['spec']}'\n\n"
 71: (16)                             "This error is lazily reported, having originally occurred
in\n"
 72: (16)                             f'  File {fd["filename"]}, line {fd["lineno"]}, in
{fd["function"]}\n\n'
 73: (16)                             f'----> {"".join(fd["code_context"] or "").strip()}'
 74: (12)                         )
 75: (0)              def _lazy_import(fullname):
 76: (4)                  """Return a lazily imported proxy for a module or library.
```

```
 77: (4)                 Warning
 78: (4)                 -------
 79: (4)                 Importing using this function can currently cause trouble
 80: (4)                 when the user tries to import from a subpackage of a module before
 81: (4)                 the package is fully imported. In particular, this idiom may not work:
 82: (6)                   np = lazy_import("numpy")
 83: (6)                   from numpy.lib import recfunctions
 84: (4)                 This is due to a difference in the way Python's LazyLoader handles
 85: (4)                 subpackage imports compared to the normal import process. Hopefully
 86: (4)                 we will get Python's LazyLoader to fix this, or find a workaround.
 87: (4)                 In the meantime, this is a potential problem.
 88: (4)                 The workaround is to import numpy before importing from the subpackage.
 89: (4)                 Notes
 90: (4)                 -----
 91: (4)                 We often see the following pattern::
 92: (6)                   def myfunc():
 93: (10)                      import scipy as sp
 94: (10)                      sp.argmin(...)
 95: (10)                      ....
 96: (4)                 This is to prevent a library, in this case `scipy`, from being
 97: (4)                 imported at function definition time, since that can be slow.
 98: (4)                 This function provides a proxy module that, upon access, imports
 99: (4)                 the actual module.  So the idiom equivalent to the above example is::
 100: (6)                  sp = lazy.load("scipy")
 101: (6)                  def myfunc():
 102: (10)                     sp.argmin(...)
 103: (10)                     ....
 104: (4)                The initial import time is fast because the actual import is delayed
 105: (4)                until the first attribute is requested. The overall import time may
 106: (4)                decrease as well for users that don't make use of large portions
 107: (4)                of the library.
 108: (4)                Parameters
 109: (4)                ----------
 110: (4)                fullname : str
 111: (8)                    The full name of the package or subpackage to import.  For example::
 112: (10)                     sp = lazy.load("scipy")  # import scipy as sp
 113: (10)                     spla = lazy.load("scipy.linalg")  # import scipy.linalg as spla
 114: (4)                Returns
 115: (4)                -------
 116: (4)                pm : importlib.util._LazyModule
 117: (8)                    Proxy module. Can be used like any regularly imported module.
 118: (8)                    Actual loading of the module occurs upon first attribute request.
 119: (4)                """
 120: (4)                try:
 121: (8)                    return sys.modules[fullname]
 122: (4)                except:
 123: (8)                    pass
 124: (4)                spec = importlib.util.find_spec(fullname)
 125: (4)                if spec is None:
 126: (8)                    try:
 127: (12)                       parent = inspect.stack()[1]
 128: (12)                       frame_data = {
 129: (16)                           "spec": fullname,
 130: (16)                           "filename": parent.filename,
 131: (16)                           "lineno": parent.lineno,
 132: (16)                           "function": parent.function,
 133: (16)                           "code_context": parent.code_context,
 134: (12)                       }
 135: (12)                       return DelayedImportErrorModule(frame_data,
"DelayedImportErrorModule")
 136: (8)                    finally:
 137: (12)                       del parent
 138: (4)                module = importlib.util.module_from_spec(spec)
 139: (4)                sys.modules[fullname] = module
 140: (4)                loader = importlib.util.LazyLoader(spec.loader)
 141: (4)                loader.exec_module(module)
 142: (4)                return module

----------------------------------------
```

File 7 - relabel.py:

```
1: (0)              import networkx as nx
2: (0)              __all__ = ["convert_node_labels_to_integers", "relabel_nodes"]
3: (0)              @nx._dispatchable(
4: (4)                  preserve_all_attrs=True, mutates_input={"not copy": 2}, returns_graph=True
5: (0)              )
6: (0)              def relabel_nodes(G, mapping, copy=True):
7: (4)                  """Relabel the nodes of the graph G according to a given mapping.
8: (4)                  The original node ordering may not be preserved if `copy` is `False` and
the
9: (4)                  mapping includes overlap between old and new labels.
10: (4)                 Parameters
11: (4)                 ----------
12: (4)                 G : graph
13: (7)                    A NetworkX graph
14: (4)                 mapping : dictionary
15: (7)                    A dictionary with the old labels as keys and new labels as values.
16: (7)                    A partial mapping is allowed. Mapping 2 nodes to a single node is
allowed.
17: (7)                    Any non-node keys in the mapping are ignored.
18: (4)                 copy : bool (optional, default=True)
19: (7)                    If True return a copy, or if False relabel the nodes in place.
20: (4)                 Examples
21: (4)                 --------
22: (4)                 To create a new graph with nodes relabeled according to a given
23: (4)                 dictionary:
24: (4)                 >>> G = nx.path_graph(3)
25: (4)                 >>> sorted(G)
26: (4)                 [0, 1, 2]
27: (4)                 >>> mapping = {0: "a", 1: "b", 2: "c"}
28: (4)                 >>> H = nx.relabel_nodes(G, mapping)
29: (4)                 >>> sorted(H)
30: (4)                 ['a', 'b', 'c']
31: (4)                 Nodes can be relabeled with any hashable object, including numbers
32: (4)                 and strings:
33: (4)                 >>> import string
34: (4)                 >>> G = nx.path_graph(26)  # nodes are integers 0 through 25
35: (4)                 >>> sorted(G)[:3]
36: (4)                 [0, 1, 2]
37: (4)                 >>> mapping = dict(zip(G, string.ascii_lowercase))
38: (4)                 >>> G = nx.relabel_nodes(G, mapping)  # nodes are characters a through z
39: (4)                 >>> sorted(G)[:3]
40: (4)                 ['a', 'b', 'c']
41: (4)                 >>> mapping = dict(zip(G, range(1, 27)))
42: (4)                 >>> G = nx.relabel_nodes(G, mapping)  # nodes are integers 1 through 26
43: (4)                 >>> sorted(G)[:3]
44: (4)                 [1, 2, 3]
45: (4)                 To perform a partial in-place relabeling, provide a dictionary
46: (4)                 mapping only a subset of the nodes, and set the `copy` keyword
47: (4)                 argument to False:
48: (4)                 >>> G = nx.path_graph(3)  # nodes 0-1-2
49: (4)                 >>> mapping = {0: "a", 1: "b"}  # 0->'a' and 1->'b'
50: (4)                 >>> G = nx.relabel_nodes(G, mapping, copy=False)
51: (4)                 >>> sorted(G, key=str)
52: (4)                 [2, 'a', 'b']
53: (4)                 A mapping can also be given as a function:
54: (4)                 >>> G = nx.path_graph(3)
55: (4)                 >>> H = nx.relabel_nodes(G, lambda x: x**2)
56: (4)                 >>> list(H)
57: (4)                 [0, 1, 4]
58: (4)                 In a multigraph, relabeling two or more nodes to the same new node
59: (4)                 will retain all edges, but may change the edge keys in the process:
60: (4)                 >>> G = nx.MultiGraph()
61: (4)                 >>> G.add_edge(0, 1, value="a")  # returns the key for this edge
62: (4)                 0
63: (4)                 >>> G.add_edge(0, 2, value="b")
64: (4)                 0
```

```
65: (4)                 >>> G.add_edge(0, 3, value="c")
66: (4)                 0
67: (4)                 >>> mapping = {1: 4, 2: 4, 3: 4}
68: (4)                 >>> H = nx.relabel_nodes(G, mapping, copy=True)
69: (4)                 >>> print(H[0])
70: (4)                 {4: {0: {'value': 'a'}, 1: {'value': 'b'}, 2: {'value': 'c'}}}
71: (4)                 This works for in-place relabeling too:
72: (4)                 >>> G = nx.relabel_nodes(G, mapping, copy=False)
73: (4)                 >>> print(G[0])
74: (4)                 {4: {0: {'value': 'a'}, 1: {'value': 'b'}, 2: {'value': 'c'}}}
75: (4)                 Notes
76: (4)                 -----
77: (4)                 Only the nodes specified in the mapping will be relabeled.
78: (4)                 Any non-node keys in the mapping are ignored.
79: (4)                 The keyword setting copy=False modifies the graph in place.
80: (4)                 Relabel_nodes avoids naming collisions by building a
81: (4)                 directed graph from ``mapping`` which specifies the order of
82: (4)                 relabelings. Naming collisions, such as a->b, b->c, are ordered
83: (4)                 such that "b" gets renamed to "c" before "a" gets renamed "b".
84: (4)                 In cases of circular mappings (e.g. a->b, b->a), modifying the
85: (4)                 graph is not possible in-place and an exception is raised.
86: (4)                 In that case, use copy=True.
87: (4)                 If a relabel operation on a multigraph would cause two or more
88: (4)                 edges to have the same source, target and key, the second edge must
89: (4)                 be assigned a new key to retain all edges. The new key is set
90: (4)                 to the lowest non-negative integer not already used as a key
91: (4)                 for edges between these two nodes. Note that this means non-numeric
92: (4)                 keys may be replaced by numeric keys.
93: (4)                 See Also
94: (4)                 --------
95: (4)                 convert_node_labels_to_integers
96: (4)                 """
97: (4)                 m = {n: mapping(n) for n in G} if callable(mapping) else mapping
98: (4)                 if copy:
99: (8)                     return _relabel_copy(G, m)
100: (4)                else:
101: (8)                    return _relabel_inplace(G, m)
102: (0)            def _relabel_inplace(G, mapping):
103: (4)                if len(mapping.keys() & mapping.values()) > 0:
104: (8)                    D = nx.DiGraph(list(mapping.items()))
105: (8)                    D.remove_edges_from(nx.selfloop_edges(D))
106: (8)                    try:
107: (12)                       nodes = reversed(list(nx.topological_sort(D)))
108: (8)                    except nx.NetworkXUnfeasible as err:
109: (12)                       raise nx.NetworkXUnfeasible(
110: (16)                           "The node label sets are overlapping and no ordering can "
111: (16)                           "resolve the mapping. Use copy=True."
112: (12)                       ) from err
113: (4)                else:
114: (8)                    nodes = [n for n in G if n in mapping]
115: (4)                multigraph = G.is_multigraph()
116: (4)                directed = G.is_directed()
117: (4)                for old in nodes:
118: (8)                    try:
119: (12)                       new = mapping[old]
120: (12)                       G.add_node(new, **G.nodes[old])
121: (8)                    except KeyError:
122: (12)                       continue
123: (8)                    if new == old:
124: (12)                       continue
125: (8)                    if multigraph:
126: (12)                       new_edges = [
127: (16)                           (new, new if old == target else target, key, data)
128: (16)                           for (_, target, key, data) in G.edges(old, data=True,
keys=True)
129: (12)                       ]
130: (12)                       if directed:
131: (16)                           new_edges += [
132: (20)                               (new if old == source else source, new, key, data)
```

```
133: (20)                                for (source, _, key, data) in G.in_edges(old, data=True,
keys=True)
134: (16)                            ]
135: (12)                        seen = set()
136: (12)                        for i, (source, target, key, data) in enumerate(new_edges):
137: (16)                            if target in G[source] and key in G[source][target]:
138: (20)                                new_key = 0 if not isinstance(key, int | float) else key
139: (20)                                while new_key in G[source][target] or (target, new_key) in
seen:
140: (24)                                    new_key += 1
141: (20)                                new_edges[i] = (source, target, new_key, data)
142: (20)                                seen.add((target, new_key))
143: (8)                    else:
144: (12)                        new_edges = [
145: (16)                            (new, new if old == target else target, data)
146: (16)                            for (_, target, data) in G.edges(old, data=True)
147: (12)                        ]
148: (12)                        if directed:
149: (16)                            new_edges += [
150: (20)                                (new if old == source else source, new, data)
151: (20)                                for (source, _, data) in G.in_edges(old, data=True)
152: (16)                            ]
153: (8)                    G.remove_node(old)
154: (8)                    G.add_edges_from(new_edges)
155: (4)                return G
156: (0)            def _relabel_copy(G, mapping):
157: (4)                H = G.__class__()
158: (4)                H.add_nodes_from(mapping.get(n, n) for n in G)
159: (4)                H._node.update((mapping.get(n, n), d.copy()) for n, d in G.nodes.items())
160: (4)                if G.is_multigraph():
161: (8)                    new_edges = [
162: (12)                       (mapping.get(n1, n1), mapping.get(n2, n2), k, d.copy())
163: (12)                       for (n1, n2, k, d) in G.edges(keys=True, data=True)
164: (8)                    ]
165: (8)                    undirected = not G.is_directed()
166: (8)                    seen_edges = set()
167: (8)                    for i, (source, target, key, data) in enumerate(new_edges):
168: (12)                       while (source, target, key) in seen_edges:
169: (16)                           if not isinstance(key, int | float):
170: (20)                               key = 0
171: (16)                           key += 1
172: (12)                       seen_edges.add((source, target, key))
173: (12)                       if undirected:
174: (16)                           seen_edges.add((target, source, key))
175: (12)                       new_edges[i] = (source, target, key, data)
176: (8)                    H.add_edges_from(new_edges)
177: (4)                else:
178: (8)                    H.add_edges_from(
179: (12)                       (mapping.get(n1, n1), mapping.get(n2, n2), d.copy())
180: (12)                       for (n1, n2, d) in G.edges(data=True)
181: (8)                    )
182: (4)                H.graph.update(G.graph)
183: (4)                return H
184: (0)            @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
185: (0)            def convert_node_labels_to_integers(
186: (4)                G, first_label=0, ordering="default", label_attribute=None
187: (0)            ):
188: (4)                """Returns a copy of the graph G with the nodes relabeled using
189: (4)                consecutive integers.
190: (4)                Parameters
191: (4)                ----------
192: (4)                G : graph
193: (7)                   A NetworkX graph
194: (4)                first_label : int, optional (default=0)
195: (7)                   An integer specifying the starting offset in numbering nodes.
196: (7)                   The new integer labels are numbered first_label, ..., n-1+first_label.
197: (4)                ordering : string
198: (7)                   "default" : inherit node ordering from G.nodes()
199: (7)                   "sorted"  : inherit node ordering from sorted(G.nodes())
```

```
200: (7)                    "increasing degree" : nodes are sorted by increasing degree
201: (7)                    "decreasing degree" : nodes are sorted by decreasing degree
202: (4)                label_attribute : string, optional (default=None)
203: (7)                   Name of node attribute to store old label.  If None no attribute
204: (7)                   is created.
205: (4)                Notes
206: (4)                -----
207: (4)                Node and edge attribute data are copied to the new (relabeled) graph.
208: (4)                There is no guarantee that the relabeling of nodes to integers will
209: (4)                give the same two integers for two (even identical graphs).
210: (4)                Use the `ordering` argument to try to preserve the order.
211: (4)                See Also
212: (4)                --------
213: (4)                relabel_nodes
214: (4)                """
215: (4)                N = G.number_of_nodes() + first_label
216: (4)                if ordering == "default":
217: (8)                    mapping = dict(zip(G.nodes(), range(first_label, N)))
218: (4)                elif ordering == "sorted":
219: (8)                    nlist = sorted(G.nodes())
220: (8)                    mapping = dict(zip(nlist, range(first_label, N)))
221: (4)                elif ordering == "increasing degree":
222: (8)                    dv_pairs = [(d, n) for (n, d) in G.degree()]
223: (8)                    dv_pairs.sort()  # in-place sort from lowest to highest degree
224: (8)                    mapping = dict(zip([n for d, n in dv_pairs], range(first_label, N)))
225: (4)                elif ordering == "decreasing degree":
226: (8)                    dv_pairs = [(d, n) for (n, d) in G.degree()]
227: (8)                    dv_pairs.sort()  # in-place sort from lowest to highest degree
228: (8)                    dv_pairs.reverse()
229: (8)                    mapping = dict(zip([n for d, n in dv_pairs], range(first_label, N)))
230: (4)                else:
231: (8)                    raise nx.NetworkXError(f"Unknown node ordering: {ordering}")
232: (4)                H = relabel_nodes(G, mapping)
233: (4)                if label_attribute is not None:
234: (8)                    nx.set_node_attributes(H, {v: k for k, v in mapping.items()},
label_attribute)
235: (4)                return H

----------------------------------------

File 8 - __init__.py:

1: (0)              from networkx.algorithms.assortativity import *
2: (0)              from networkx.algorithms.asteroidal import *
3: (0)              from networkx.algorithms.boundary import *
4: (0)              from networkx.algorithms.broadcasting import *
5: (0)              from networkx.algorithms.bridges import *
6: (0)              from networkx.algorithms.chains import *
7: (0)              from networkx.algorithms.centrality import *
8: (0)              from networkx.algorithms.chordal import *
9: (0)              from networkx.algorithms.cluster import *
10: (0)             from networkx.algorithms.clique import *
11: (0)             from networkx.algorithms.communicability_alg import *
12: (0)             from networkx.algorithms.components import *
13: (0)             from networkx.algorithms.coloring import *
14: (0)             from networkx.algorithms.core import *
15: (0)             from networkx.algorithms.covering import *
16: (0)             from networkx.algorithms.cycles import *
17: (0)             from networkx.algorithms.cuts import *
18: (0)             from networkx.algorithms.d_separation import *
19: (0)             from networkx.algorithms.dag import *
20: (0)             from networkx.algorithms.distance_measures import *
21: (0)             from networkx.algorithms.distance_regular import *
22: (0)             from networkx.algorithms.dominance import *
23: (0)             from networkx.algorithms.dominating import *
24: (0)             from networkx.algorithms.efficiency_measures import *
25: (0)             from networkx.algorithms.euler import *
26: (0)             from networkx.algorithms.graphical import *
27: (0)             from networkx.algorithms.hierarchy import *
```

```
(0)             from networkx.algorithms.hybrid import *
(0)             from networkx.algorithms.link_analysis import *
(0)             from networkx.algorithms.link_prediction import *
(0)             from networkx.algorithms.lowest_common_ancestors import *
(0)             from networkx.algorithms.isolate import *
(0)             from networkx.algorithms.matching import *
(0)             from networkx.algorithms.minors import *
(0)             from networkx.algorithms.mis import *
(0)             from networkx.algorithms.moral import *
(0)             from networkx.algorithms.non_randomness import *
(0)             from networkx.algorithms.operators import *
(0)             from networkx.algorithms.planarity import *
(0)             from networkx.algorithms.planar_drawing import *
(0)             from networkx.algorithms.polynomials import *
(0)             from networkx.algorithms.reciprocity import *
(0)             from networkx.algorithms.regular import *
(0)             from networkx.algorithms.richclub import *
(0)             from networkx.algorithms.shortest_paths import *
(0)             from networkx.algorithms.similarity import *
(0)             from networkx.algorithms.graph_hashing import *
(0)             from networkx.algorithms.simple_paths import *
(0)             from networkx.algorithms.smallworld import *
(0)             from networkx.algorithms.smetric import *
(0)             from networkx.algorithms.structuralholes import *
(0)             from networkx.algorithms.sparsifiers import *
(0)             from networkx.algorithms.summarization import *
(0)             from networkx.algorithms.swap import *
(0)             from networkx.algorithms.time_dependent import *
(0)             from networkx.algorithms.traversal import *
(0)             from networkx.algorithms.triads import *
(0)             from networkx.algorithms.vitality import *
(0)             from networkx.algorithms.voronoi import *
(0)             from networkx.algorithms.walks import *
(0)             from networkx.algorithms.wiener import *
(0)             from networkx.algorithms import approximation
(0)             from networkx.algorithms import assortativity
(0)             from networkx.algorithms import bipartite
(0)             from networkx.algorithms import node_classification
(0)             from networkx.algorithms import centrality
(0)             from networkx.algorithms import chordal
(0)             from networkx.algorithms import cluster
(0)             from networkx.algorithms import clique
(0)             from networkx.algorithms import components
(0)             from networkx.algorithms import connectivity
(0)             from networkx.algorithms import community
(0)             from networkx.algorithms import coloring
(0)             from networkx.algorithms import flow
(0)             from networkx.algorithms import isomorphism
(0)             from networkx.algorithms import link_analysis
(0)             from networkx.algorithms import lowest_common_ancestors
(0)             from networkx.algorithms import operators
(0)             from networkx.algorithms import shortest_paths
(0)             from networkx.algorithms import tournament
(0)             from networkx.algorithms import traversal
(0)             from networkx.algorithms import tree
(0)             from networkx.algorithms.bipartite import complete_bipartite_graph
(0)             from networkx.algorithms.bipartite import is_bipartite
(0)             from networkx.algorithms.bipartite import projected_graph
(0)             from networkx.algorithms.connectivity import all_pairs_node_connectivity
(0)             from networkx.algorithms.connectivity import all_node_cuts
(0)             from networkx.algorithms.connectivity import average_node_connectivity
(0)             from networkx.algorithms.connectivity import edge_connectivity
(0)             from networkx.algorithms.connectivity import edge_disjoint_paths
(0)             from networkx.algorithms.connectivity import k_components
(0)             from networkx.algorithms.connectivity import k_edge_components
(0)             from networkx.algorithms.connectivity import k_edge_subgraphs
(0)             from networkx.algorithms.connectivity import k_edge_augmentation
(0)             from networkx.algorithms.connectivity import is_k_edge_connected
(0)             from networkx.algorithms.connectivity import minimum_edge_cut
```

```
97: (0)             from networkx.algorithms.connectivity import minimum_node_cut
98: (0)             from networkx.algorithms.connectivity import node_connectivity
99: (0)             from networkx.algorithms.connectivity import node_disjoint_paths
100: (0)            from networkx.algorithms.connectivity import stoer_wagner
101: (0)            from networkx.algorithms.flow import capacity_scaling
102: (0)            from networkx.algorithms.flow import cost_of_flow
103: (0)            from networkx.algorithms.flow import gomory_hu_tree
104: (0)            from networkx.algorithms.flow import max_flow_min_cost
105: (0)            from networkx.algorithms.flow import maximum_flow
106: (0)            from networkx.algorithms.flow import maximum_flow_value
107: (0)            from networkx.algorithms.flow import min_cost_flow
108: (0)            from networkx.algorithms.flow import min_cost_flow_cost
109: (0)            from networkx.algorithms.flow import minimum_cut
110: (0)            from networkx.algorithms.flow import minimum_cut_value
111: (0)            from networkx.algorithms.flow import network_simplex
112: (0)            from networkx.algorithms.isomorphism import could_be_isomorphic
113: (0)            from networkx.algorithms.isomorphism import fast_could_be_isomorphic
114: (0)            from networkx.algorithms.isomorphism import faster_could_be_isomorphic
115: (0)            from networkx.algorithms.isomorphism import is_isomorphic
116: (0)            from networkx.algorithms.isomorphism.vf2pp import *
117: (0)            from networkx.algorithms.tree.branchings import maximum_branching
118: (0)            from networkx.algorithms.tree.branchings import maximum_spanning_arborescence
119: (0)            from networkx.algorithms.tree.branchings import minimum_branching
120: (0)            from networkx.algorithms.tree.branchings import minimum_spanning_arborescence
121: (0)            from networkx.algorithms.tree.branchings import ArborescenceIterator
122: (0)            from networkx.algorithms.tree.coding import *
123: (0)            from networkx.algorithms.tree.decomposition import *
124: (0)            from networkx.algorithms.tree.mst import *
125: (0)            from networkx.algorithms.tree.operations import *
126: (0)            from networkx.algorithms.tree.recognition import *
127: (0)            from networkx.algorithms.tournament import is_tournament

----------------------------------------

File 9 - asteroidal.py:

1: (0)              """
2: (0)              Algorithms for asteroidal triples and asteroidal numbers in graphs.
3: (0)              An asteroidal triple in a graph G is a set of three non-adjacent vertices
4: (0)              u, v and w such that there exist a path between any two of them that avoids
5: (0)              closed neighborhood of the third. More formally, v_j, v_k belongs to the same
6: (0)              connected component of G - N[v_i], where N[v_i] denotes the closed
neighborhood
7: (0)              of v_i. A graph which does not contain any asteroidal triples is called
8: (0)              an AT-free graph. The class of AT-free graphs is a graph class for which
9: (0)              many NP-complete problems are solvable in polynomial time. Amongst them,
10: (0)             independent set and coloring.
11: (0)             """
12: (0)             import networkx as nx
13: (0)             from networkx.utils import not_implemented_for
14: (0)             __all__ = ["is_at_free", "find_asteroidal_triple"]
15: (0)             @not_implemented_for("directed")
16: (0)             @not_implemented_for("multigraph")
17: (0)             @nx._dispatchable
18: (0)             def find_asteroidal_triple(G):
19: (4)                 r"""Find an asteroidal triple in the given graph.
20: (4)                 An asteroidal triple is a triple of non-adjacent vertices such that
21: (4)                 there exists a path between any two of them which avoids the closed
22: (4)                 neighborhood of the third. It checks all independent triples of vertices
23: (4)                 and whether they are an asteroidal triple or not. This is done with the
24: (4)                 help of a data structure called a component structure.
25: (4)                 A component structure encodes information about which vertices belongs to
26: (4)                 the same connected component when the closed neighborhood of a given
vertex
27: (4)                 is removed from the graph. The algorithm used to check is the trivial
28: (4)                 one, outlined in [1]_, which has a runtime of
29: (4)                 :math:`O(|V||\overline{E} + |V||E|)`, where the second term is the
30: (4)                 creation of the component structure.
31: (4)                 Parameters
```

```
    ----------
    G : NetworkX Graph
        The graph to check whether is AT-free or not
    Returns
    -------
    list or None
        An asteroidal triple is returned as a list of nodes. If no asteroidal
        triple exists, i.e. the graph is AT-free, then None is returned.
        The returned value depends on the certificate parameter. The default
        option is a bool which is True if the graph is AT-free, i.e. the
        given graph contains no asteroidal triples, and False otherwise, i.e.
        if the graph contains at least one asteroidal triple.
    Notes
    -----
    The component structure and the algorithm is described in [1]_. The current
    implementation implements the trivial algorithm for simple graphs.
    References
    ----------
    .. [1] Ekkehard Köhler,
       "Recognizing Graphs without asteroidal triples",
       Journal of Discrete Algorithms 2, pages 439-452, 2004.
       https://www.sciencedirect.com/science/article/pii/S157086670400019X
    """
    V = set(G.nodes)
    if len(V) < 6:
        return None
    component_structure = create_component_structure(G)
    E_complement = set(nx.complement(G).edges)
    for e in E_complement:
        u = e[0]
        v = e[1]
        u_neighborhood = set(G[u]).union([u])
        v_neighborhood = set(G[v]).union([v])
        union_of_neighborhoods = u_neighborhood.union(v_neighborhood)
        for w in V - union_of_neighborhoods:
            if (
                component_structure[u][v] == component_structure[u][w]
                and component_structure[v][u] == component_structure[v][w]
                and component_structure[w][u] == component_structure[w][v]
            ):
                return [u, v, w]
    return None
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable
def is_at_free(G):
    """Check if a graph is AT-free.
    The method uses the `find_asteroidal_triple` method to recognize
    an AT-free graph. If no asteroidal triple is found the graph is
    AT-free and True is returned. If at least one asteroidal triple is
    found the graph is not AT-free and False is returned.
    Parameters
    ----------
    G : NetworkX Graph
        The graph to check whether is AT-free or not.
    Returns
    -------
    bool
        True if G is AT-free and False otherwise.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (4, 5)])
    >>> nx.is_at_free(G)
    True
    >>> G = nx.cycle_graph(6)
    >>> nx.is_at_free(G)
    False
    """
```

```
100: (4)                return find_asteroidal_triple(G) is None
101: (0)            @not_implemented_for("directed")
102: (0)            @not_implemented_for("multigraph")
103: (0)            @nx._dispatchable
104: (0)            def create_component_structure(G):
105: (4)                r"""Create component structure for G.
106: (4)                A *component structure* is an `nxn` array, denoted `c`, where `n` is
107: (4)                the number of vertices,  where each row and column corresponds to a
vertex.
108: (4)                .. math::
109: (8)                    c_{uv} = \begin{cases} 0, if v \in N[u] \\
110: (12)                       k, if v \in component k of G \setminus N[u] \end{cases}
111: (4)                Where `k` is an arbitrary label for each component. The structure is used
112: (4)                to simplify the detection of asteroidal triples.
113: (4)                Parameters
114: (4)                ----------
115: (4)                G : NetworkX Graph
116: (8)                    Undirected, simple graph.
117: (4)                Returns
118: (4)                -------
119: (4)                component_structure : dictionary
120: (8)                    A dictionary of dictionaries, keyed by pairs of vertices.
121: (4)                """
122: (4)                V = set(G.nodes)
123: (4)                component_structure = {}
124: (4)                for v in V:
125: (8)                    label = 0
126: (8)                    closed_neighborhood = set(G[v]).union({v})
127: (8)                    row_dict = {}
128: (8)                    for u in closed_neighborhood:
129: (12)                       row_dict[u] = 0
130: (8)                    G_reduced = G.subgraph(set(G.nodes) - closed_neighborhood)
131: (8)                    for cc in nx.connected_components(G_reduced):
132: (12)                       label += 1
133: (12)                       for u in cc:
134: (16)                           row_dict[u] = label
135: (8)                    component_structure[v] = row_dict
136: (4)                return component_structure

----------------------------------------

File 10 - boundary.py:

1: (0)              """Routines to find the boundary of a set of nodes.
2: (0)              An edge boundary is a set of edges, each of which has exactly one
3: (0)              endpoint in a given set of nodes (or, in the case of directed graphs,
4: (0)              the set of edges whose source node is in the set).
5: (0)              A node boundary of a set *S* of nodes is the set of (out-)neighbors of
6: (0)              nodes in *S* that are outside *S*.
7: (0)              """
8: (0)              from itertools import chain
9: (0)              import networkx as nx
10: (0)             __all__ = ["edge_boundary", "node_boundary"]
11: (0)             @nx._dispatchable(edge_attrs={"data": "default"}, preserve_edge_attrs="data")
12: (0)             def edge_boundary(G, nbunch1, nbunch2=None, data=False, keys=False,
default=None):
13: (4)                 """Returns the edge boundary of `nbunch1`.
14: (4)                 The *edge boundary* of a set *S* with respect to a set *T* is the
15: (4)                 set of edges (*u*, *v*) such that *u* is in *S* and *v* is in *T*.
16: (4)                 If *T* is not specified, it is assumed to be the set of all nodes
17: (4)                 not in *S*.
18: (4)                 Parameters
19: (4)                 ----------
20: (4)                 G : NetworkX graph
21: (4)                 nbunch1 : iterable
22: (8)                     Iterable of nodes in the graph representing the set of nodes
23: (8)                     whose edge boundary will be returned. (This is the set *S* from
24: (8)                     the definition above.)
25: (4)                 nbunch2 : iterable
```

```
 26: (8)                    Iterable of nodes representing the target (or "exterior") set of
 27: (8)                    nodes. (This is the set *T* from the definition above.) If not
 28: (8)                    specified, this is assumed to be the set of all nodes in `G`
 29: (8)                    not in `nbunch1`.
 30: (4)                keys : bool
 31: (8)                    This parameter has the same meaning as in
 32: (8)                    :meth:`MultiGraph.edges`.
 33: (4)                data : bool or object
 34: (8)                    This parameter has the same meaning as in
 35: (8)                    :meth:`MultiGraph.edges`.
 36: (4)                default : object
 37: (8)                    This parameter has the same meaning as in
 38: (8)                    :meth:`MultiGraph.edges`.
 39: (4)                Returns
 40: (4)                -------
 41: (4)                iterator
 42: (8)                    An iterator over the edges in the boundary of `nbunch1` with
 43: (8)                    respect to `nbunch2`. If `keys`, `data`, or `default`
 44: (8)                    are specified and `G` is a multigraph, then edges are returned
 45: (8)                    with keys and/or data, as in :meth:`MultiGraph.edges`.
 46: (4)                Examples
 47: (4)                --------
 48: (4)                >>> G = nx.wheel_graph(6)
 49: (4)                When nbunch2=None:
 50: (4)                >>> list(nx.edge_boundary(G, (1, 3)))
 51: (4)                [(1, 0), (1, 2), (1, 5), (3, 0), (3, 2), (3, 4)]
 52: (4)                When nbunch2 is given:
 53: (4)                >>> list(nx.edge_boundary(G, (1, 3), (2, 0)))
 54: (4)                [(1, 0), (1, 2), (3, 0), (3, 2)]
 55: (4)                Notes
 56: (4)                -----
 57: (4)                Any element of `nbunch` that is not in the graph `G` will be
 58: (4)                ignored.
 59: (4)                `nbunch1` and `nbunch2` are usually meant to be disjoint, but in
 60: (4)                the interest of speed and generality, that is not required here.
 61: (4)                """
 62: (4)                nset1 = {n for n in nbunch1 if n in G}
 63: (4)                if G.is_multigraph():
 64: (8)                    edges = G.edges(nset1, data=data, keys=keys, default=default)
 65: (4)                else:
 66: (8)                    edges = G.edges(nset1, data=data, default=default)
 67: (4)                if nbunch2 is None:
 68: (8)                    return (e for e in edges if (e[0] in nset1) ^ (e[1] in nset1))
 69: (4)                nset2 = set(nbunch2)
 70: (4)                return (
 71: (8)                    e
 72: (8)                    for e in edges
 73: (8)                    if (e[0] in nset1 and e[1] in nset2) or (e[1] in nset1 and e[0] in
nset2)
 74: (4)                )
 75: (0)            @nx._dispatchable
 76: (0)            def node_boundary(G, nbunch1, nbunch2=None):
 77: (4)                """Returns the node boundary of `nbunch1`.
 78: (4)                The *node boundary* of a set *S* with respect to a set *T* is the
 79: (4)                set of nodes *v* in *T* such that for some *u* in *S*, there is an
 80: (4)                edge joining *u* to *v*. If *T* is not specified, it is assumed to
 81: (4)                be the set of all nodes not in *S*.
 82: (4)                Parameters
 83: (4)                ----------
 84: (4)                G : NetworkX graph
 85: (4)                nbunch1 : iterable
 86: (8)                    Iterable of nodes in the graph representing the set of nodes
 87: (8)                    whose node boundary will be returned. (This is the set *S* from
 88: (8)                    the definition above.)
 89: (4)                nbunch2 : iterable
 90: (8)                    Iterable of nodes representing the target (or "exterior") set of
 91: (8)                    nodes. (This is the set *T* from the definition above.) If not
 92: (8)                    specified, this is assumed to be the set of all nodes in `G`
 93: (8)                    not in `nbunch1`.
```

```
94: (4)                 Returns
95: (4)                 -------
96: (4)                 set
97: (8)                     The node boundary of `nbunch1` with respect to `nbunch2`.
98: (4)                 Examples
99: (4)                 --------
100: (4)                >>> G = nx.wheel_graph(6)
101: (4)                When nbunch2=None:
102: (4)                >>> list(nx.node_boundary(G, (3, 4)))
103: (4)                [0, 2, 5]
104: (4)                When nbunch2 is given:
105: (4)                >>> list(nx.node_boundary(G, (3, 4), (0, 1, 5)))
106: (4)                [0, 5]
107: (4)                Notes
108: (4)                -----
109: (4)                Any element of `nbunch` that is not in the graph `G` will be
110: (4)                ignored.
111: (4)                `nbunch1` and `nbunch2` are usually meant to be disjoint, but in
112: (4)                the interest of speed and generality, that is not required here.
113: (4)                """
114: (4)                nset1 = {n for n in nbunch1 if n in G}
115: (4)                bdy = set(chain.from_iterable(G[v] for v in nset1)) - nset1
116: (4)                if nbunch2 is not None:
117: (8)                    bdy &= set(nbunch2)
118: (4)                return bdy
```

File 11 - bridges.py:

```
1: (0)              """Bridge-finding algorithms."""
2: (0)              from itertools import chain
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["bridges", "has_bridges", "local_bridges"]
6: (0)              @not_implemented_for("directed")
7: (0)              @nx._dispatchable
8: (0)              def bridges(G, root=None):
9: (4)                  """Generate all bridges in a graph.
10: (4)                 A *bridge* in a graph is an edge whose removal causes the number of
11: (4)                 connected components of the graph to increase.  Equivalently, a bridge is
an
12: (4)                 edge that does not belong to any cycle. Bridges are also known as cut-
edges,
13: (4)                 isthmuses, or cut arcs.
14: (4)                 Parameters
15: (4)                 ----------
16: (4)                 G : undirected graph
17: (4)                 root : node (optional)
18: (7)                    A node in the graph `G`. If specified, only the bridges in the
19: (7)                    connected component containing this node will be returned.
20: (4)                 Yields
21: (4)                 ------
22: (4)                 e : edge
23: (7)                    An edge in the graph whose removal disconnects the graph (or
24: (7)                    causes the number of connected components to increase).
25: (4)                 Raises
26: (4)                 ------
27: (4)                 NodeNotFound
28: (7)                    If `root` is not in the graph `G`.
29: (4)                 NetworkXNotImplemented
30: (8)                     If `G` is a directed graph.
31: (4)                 Examples
32: (4)                 --------
33: (4)                 The barbell graph with parameter zero has a single bridge:
34: (4)                 >>> G = nx.barbell_graph(10, 0)
35: (4)                 >>> list(nx.bridges(G))
36: (4)                 [(9, 10)]
37: (4)                 Notes
```

```
38: (4)                 -----
39: (4)                 This is an implementation of the algorithm described in [1]_.  An edge is
a
40: (4)                 bridge if and only if it is not contained in any chain. Chains are found
41: (4)                 using the :func:`networkx.chain_decomposition` function.
42: (4)                 The algorithm described in [1]_ requires a simple graph. If the provided
43: (4)                 graph is a multigraph, we convert it to a simple graph and verify that any
44: (4)                 bridges discovered by the chain decomposition algorithm are not multi-
edges.
45: (4)                 Ignoring polylogarithmic factors, the worst-case time complexity is the
46: (4)                 same as the :func:`networkx.chain_decomposition` function,
47: (4)                 $O(m + n)$, where $n$ is the number of nodes in the graph and $m$ is
48: (4)                 the number of edges.
49: (4)                 References
50: (4)                 ----------
51: (4)                 .. [1] https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29#Bridge-
Finding_with_Chain_Decompositions
52: (4)                 """
53: (4)                 multigraph = G.is_multigraph()
54: (4)                 H = nx.Graph(G) if multigraph else G
55: (4)                 chains = nx.chain_decomposition(H, root=root)
56: (4)                 chain_edges = set(chain.from_iterable(chains))
57: (4)                 H_copy = H.copy()
58: (4)                 if root is not None:
59: (8)                     H = H.subgraph(nx.node_connected_component(H, root)).copy()
60: (4)                 for u, v in H.edges():
61: (8)                     if (u, v) not in chain_edges and (v, u) not in chain_edges:
62: (12)                        if multigraph and len(G[u][v]) > 1:
63: (16)                            continue
64: (12)                        yield u, v
65: (0)             @not_implemented_for("directed")
66: (0)             @nx._dispatchable
67: (0)             def has_bridges(G, root=None):
68: (4)                 """Decide whether a graph has any bridges.
69: (4)                 A *bridge* in a graph is an edge whose removal causes the number of
70: (4)                 connected components of the graph to increase.
71: (4)                 Parameters
72: (4)                 ----------
73: (4)                 G : undirected graph
74: (4)                 root : node (optional)
75: (7)                    A node in the graph `G`. If specified, only the bridges in the
76: (7)                    connected component containing this node will be considered.
77: (4)                 Returns
78: (4)                 -------
79: (4)                 bool
80: (7)                    Whether the graph (or the connected component containing `root`)
81: (7)                    has any bridges.
82: (4)                 Raises
83: (4)                 ------
84: (4)                 NodeNotFound
85: (7)                    If `root` is not in the graph `G`.
86: (4)                 NetworkXNotImplemented
87: (8)                     If `G` is a directed graph.
88: (4)                 Examples
89: (4)                 --------
90: (4)                 The barbell graph with parameter zero has a single bridge::
91: (8)                     >>> G = nx.barbell_graph(10, 0)
92: (8)                     >>> nx.has_bridges(G)
93: (8)                     True
94: (4)                 On the other hand, the cycle graph has no bridges::
95: (8)                     >>> G = nx.cycle_graph(5)
96: (8)                     >>> nx.has_bridges(G)
97: (8)                     False
98: (4)                 Notes
99: (4)                 -----
100: (4)                This implementation uses the :func:`networkx.bridges` function, so
101: (4)                it shares its worst-case time complexity, $O(m + n)$, ignoring
102: (4)                polylogarithmic factors, where $n$ is the number of nodes in the
103: (4)                graph and $m$ is the number of edges.
```

```
104: (4)                """
105: (4)                try:
106: (8)                    next(bridges(G, root=root))
107: (4)                except StopIteration:
108: (8)                    return False
109: (4)                else:
110: (8)                    return True
111: (0)            @not_implemented_for("multigraph")
112: (0)            @not_implemented_for("directed")
113: (0)            @nx._dispatchable(edge_attrs="weight")
114: (0)            def local_bridges(G, with_span=True, weight=None):
115: (4)                """Iterate over local bridges of `G` optionally computing the span
116: (4)                A *local bridge* is an edge whose endpoints have no common neighbors.
117: (4)                That is, the edge is not part of a triangle in the graph.
118: (4)                The *span* of a *local bridge* is the shortest path length between
119: (4)                the endpoints if the local bridge is removed.
120: (4)                Parameters
121: (4)                ----------
122: (4)                G : undirected graph
123: (4)                with_span : bool
124: (8)                    If True, yield a 3-tuple `(u, v, span)`
125: (4)                weight : function, string or None (default: None)
126: (8)                    If function, used to compute edge weights for the span.
127: (8)                    If string, the edge data attribute used in calculating span.
128: (8)                    If None, all edges have weight 1.
129: (4)                Yields
130: (4)                ------
131: (4)                e : edge
132: (8)                    The local bridges as an edge 2-tuple of nodes `(u, v)` or
133: (8)                    as a 3-tuple `(u, v, span)` when `with_span is True`.
134: (4)                Raises
135: (4)                ------
136: (4)                NetworkXNotImplemented
137: (8)                    If `G` is a directed graph or multigraph.
138: (4)                Examples
139: (4)                --------
140: (4)                A cycle graph has every edge a local bridge with span N-1.
141: (7)                   >>> G = nx.cycle_graph(9)
142: (7)                   >>> (0, 8, 8) in set(nx.local_bridges(G))
143: (7)                   True
144: (4)                """
145: (4)                if with_span is not True:
146: (8)                    for u, v in G.edges:
147: (12)                       if not (set(G[u]) & set(G[v])):
148: (16)                           yield u, v
149: (4)                else:
150: (8)                    wt = nx.weighted._weight_function(G, weight)
151: (8)                    for u, v in G.edges:
152: (12)                       if not (set(G[u]) & set(G[v])):
153: (16)                           enodes = {u, v}
154: (16)                           def hide_edge(n, nbr, d):
155: (20)                               if n not in enodes or nbr not in enodes:
156: (24)                                   return wt(n, nbr, d)
157: (20)                               return None
158: (16)                           try:
159: (20)                               span = nx.shortest_path_length(G, u, v, weight=hide_edge)
160: (20)                               yield u, v, span
161: (16)                           except nx.NetworkXNoPath:
162: (20)                               yield u, v, float("inf")
```

File 12 - broadcasting.py:

```
1: (0)              """Routines to calculate the broadcast time of certain graphs.
2: (0)              Broadcasting is an information dissemination problem in which a node in a
graph,
3: (0)              called the originator, must distribute a message to all other nodes by placing
4: (0)              a series of calls along the edges of the graph. Once informed, other nodes aid
```

```
5: (0)              the originator in distributing the message.
6: (0)              The broadcasting must be completed as quickly as possible subject to the
7: (0)              following constraints:
8: (0)              - Each call requires one unit of time.
9: (0)              - A node can only participate in one call per unit of time.
10: (0)             - Each call only involves two adjacent nodes: a sender and a receiver.
11: (0)             """
12: (0)             import networkx as nx
13: (0)             from networkx import NetworkXError
14: (0)             from networkx.utils import not_implemented_for
15: (0)             __all__ = [
16: (4)                 "tree_broadcast_center",
17: (4)                 "tree_broadcast_time",
18: (0)             ]
19: (0)             def _get_max_broadcast_value(G, U, v, values):
20: (4)                 adj = sorted(set(G.neighbors(v)) & U, key=values.get, reverse=True)
21: (4)                 return max(values[u] + i for i, u in enumerate(adj, start=1))
22: (0)             def _get_broadcast_centers(G, v, values, target):
23: (4)                 adj = sorted(G.neighbors(v), key=values.get, reverse=True)
24: (4)                 j = next(i for i, u in enumerate(adj, start=1) if values[u] + i == target)
25: (4)                 return set([v] + adj[:j])
26: (0)             @not_implemented_for("directed")
27: (0)             @not_implemented_for("multigraph")
28: (0)             @nx._dispatchable
29: (0)             def tree_broadcast_center(G):
30: (4)                 """Return the Broadcast Center of the tree `G`.
31: (4)                 The broadcast center of a graph G denotes the set of nodes having
32: (4)                 minimum broadcast time [1]_. This is a linear algorithm for determining
33: (4)                 the broadcast center of a tree with ``N`` nodes, as a by-product it also
34: (4)                 determines the broadcast time from the broadcast center.
35: (4)                 Parameters
36: (4)                 ----------
37: (4)                 G : undirected graph
38: (8)                     The graph should be an undirected tree
39: (4)                 Returns
40: (4)                 -------
41: (4)                 BC : (int, set) tuple
42: (8)                     minimum broadcast number of the tree, set of broadcast centers
43: (4)                 Raises
44: (4)                 ------
45: (4)                 NetworkXNotImplemented
46: (8)                     If the graph is directed or is a multigraph.
47: (4)                 References
48: (4)                 ----------
49: (4)                 .. [1] Slater, P.J., Cockayne, E.J., Hedetniemi, S.T,
50: (7)                    Information dissemination in trees. SIAM J.Comput. 10(4), 692–701
(1981)
51: (4)                 """
52: (4)                 if not nx.is_tree(G):
53: (8)                     NetworkXError("Input graph is not a tree")
54: (4)                 if G.number_of_nodes() == 2:
55: (8)                     return 1, set(G.nodes())
56: (4)                 if G.number_of_nodes() == 1:
57: (8)                     return 0, set(G.nodes())
58: (4)                 U = {node for node, deg in G.degree if deg == 1}
59: (4)                 values = {n: 0 for n in U}
60: (4)                 T = G.copy()
61: (4)                 T.remove_nodes_from(U)
62: (4)                 W = {node for node, deg in T.degree if deg == 1}
63: (4)                 values.update((w, G.degree[w] - 1) for w in W)
64: (4)                 while T.number_of_nodes() >= 2:
65: (8)                     w = min(W, key=lambda n: values[n])
66: (8)                     v = next(T.neighbors(w))
67: (8)                     U.add(w)
68: (8)                     W.remove(w)
69: (8)                     T.remove_node(w)
70: (8)                     if T.degree(v) == 1:
71: (12)                        values.update({v: _get_max_broadcast_value(G, U, v, values)})
72: (12)                        W.add(v)
```

```
73: (4)                 v = nx.utils.arbitrary_element(T)
74: (4)                 b_T = _get_max_broadcast_value(G, U, v, values)
75: (4)                 return b_T, _get_broadcast_centers(G, v, values, b_T)
76: (0)             @not_implemented_for("directed")
77: (0)             @not_implemented_for("multigraph")
78: (0)             @nx._dispatchable
79: (0)             def tree_broadcast_time(G, node=None):
80: (4)                 """Return the Broadcast Time of the tree `G`.
81: (4)                 The minimum broadcast time of a node is defined as the minimum amount
82: (4)                 of time required to complete broadcasting starting from the
83: (4)                 originator. The broadcast time of a graph is the maximum over
84: (4)                 all nodes of the minimum broadcast time from that node [1]_.
85: (4)                 This function returns the minimum broadcast time of `node`.
86: (4)                 If `node` is None the broadcast time for the graph is returned.
87: (4)                 Parameters
88: (4)                 ----------
89: (4)                 G : undirected graph
90: (8)                     The graph should be an undirected tree
91: (4)                 node: int, optional
92: (8)                     index of starting node. If `None`, the algorithm returns the broadcast
93: (8)                     time of the tree.
94: (4)                 Returns
95: (4)                 -------
96: (4)                 BT : int
97: (8)                     Broadcast Time of a node in a tree
98: (4)                 Raises
99: (4)                 ------
100: (4)                NetworkXNotImplemented
101: (8)                    If the graph is directed or is a multigraph.
102: (4)                References
103: (4)                ----------
104: (4)                .. [1] Harutyunyan, H. A. and Li, Z.
105: (8)                    "A Simple Construction of Broadcast Graphs."
106: (8)                    In Computing and Combinatorics. COCOON 2019
107: (8)                    (Ed. D. Z. Du and C. Tian.) Springer, pp. 240-253, 2019.
108: (4)                """
109: (4)                b_T, b_C = tree_broadcast_center(G)
110: (4)                if node is not None:
111: (8)                    return b_T + min(nx.shortest_path_length(G, node, u) for u in b_C)
112: (4)                dist_from_center = dict.fromkeys(G, len(G))
113: (4)                for u in b_C:
114: (8)                    for v, dist in nx.shortest_path_length(G, u).items():
115: (12)                       if dist < dist_from_center[v]:
116: (16)                           dist_from_center[v] = dist
117: (4)                return b_T + max(dist_from_center.values())

----------------------------------------

File 13 - chains.py:

1: (0)              """Functions for finding chains in a graph."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for
4: (0)              __all__ = ["chain_decomposition"]
5: (0)              @not_implemented_for("directed")
6: (0)              @not_implemented_for("multigraph")
7: (0)              @nx._dispatchable
8: (0)              def chain_decomposition(G, root=None):
9: (4)                  """Returns the chain decomposition of a graph.
10: (4)                 The *chain decomposition* of a graph with respect a depth-first
11: (4)                 search tree is a set of cycles or paths derived from the set of
12: (4)                 fundamental cycles of the tree in the following manner. Consider
13: (4)                 each fundamental cycle with respect to the given tree, represented
14: (4)                 as a list of edges beginning with the nontree edge oriented away
15: (4)                 from the root of the tree. For each fundamental cycle, if it
16: (4)                 overlaps with any previous fundamental cycle, just take the initial
17: (4)                 non-overlapping segment, which is a path instead of a cycle. Each
18: (4)                 cycle or path is called a *chain*. For more information, see [1]_.
19: (4)                 Parameters
```

```
(4)     ----------
(4)     G : undirected graph
(4)     root : node (optional)
(7)        A node in the graph `G`. If specified, only the chain
(7)        decomposition for the connected component containing this node
(7)        will be returned. This node indicates the root of the depth-first
(7)        search tree.
(4)     Yields
(4)     ------
(4)     chain : list
(7)        A list of edges representing a chain. There is no guarantee on
(7)        the orientation of the edges in each chain (for example, if a
(7)        chain includes the edge joining nodes 1 and 2, the chain may
(7)        include either (1, 2) or (2, 1)).
(4)     Raises
(4)     ------
(4)     NodeNotFound
(7)        If `root` is not in the graph `G`.
(4)     Examples
(4)     --------
(4)     >>> G = nx.Graph([(0, 1), (1, 4), (3, 4), (3, 5), (4, 5)])
(4)     >>> list(nx.chain_decomposition(G))
(4)     [[(4, 5), (5, 3), (3, 4)]]
(4)     Notes
(4)     -----
(4)     The worst-case running time of this implementation is linear in the
(4)     number of nodes and number of edges [1]_.
(4)     References
(4)     ----------
(4)     .. [1] Jens M. Schmidt (2013). "A simple test on 2-vertex-
(7)        and 2-edge-connectivity." *Information Processing Letters*,
(7)        113, 241–244. Elsevier. <https://doi.org/10.1016/j.ipl.2013.01.016>
(4)     """
(4)     def _dfs_cycle_forest(G, root=None):
(8)         """Builds a directed graph composed of cycles from the given graph.
(8)         `G` is an undirected simple graph. `root` is a node in the graph
(8)         from which the depth-first search is started.
(8)         This function returns both the depth-first search cycle graph
(8)         (as a :class:`~networkx.DiGraph`) and the list of nodes in
(8)         depth-first preorder. The depth-first search cycle graph is a
(8)         directed graph whose edges are the edges of `G` oriented toward
(8)         the root if the edge is a tree edge and away from the root if
(8)         the edge is a non-tree edge. If `root` is not specified, this
(8)         performs a depth-first search on each connected component of `G`
(8)         and returns a directed forest instead.
(8)         If `root` is not in the graph, this raises :exc:`KeyError`.
(8)         """
(8)         H = nx.DiGraph()
(8)         nodes = []
(8)         for u, v, d in nx.dfs_labeled_edges(G, source=root):
(12)            if d == "forward":
(16)                if u == v:
(20)                    H.add_node(v, parent=None)
(20)                    nodes.append(v)
(16)                else:
(20)                    H.add_node(v, parent=u)
(20)                    H.add_edge(v, u, nontree=False)
(20)                    nodes.append(v)
(12)            elif d == "nontree" and v not in H[u]:
(16)                H.add_edge(v, u, nontree=True)
(12)            else:
(16)                pass
(8)         return H, nodes
(4)     def _build_chain(G, u, v, visited):
(8)         """Generate the chain starting from the given nontree edge.
(8)         `G` is a DFS cycle graph as constructed by
(8)         :func:`_dfs_cycle_graph`. The edge (`u`, `v`) is a nontree edge
(8)         that begins a chain. `visited` is a set representing the nodes
(8)         in `G` that have already been visited.
```

```
89: (8)                     This function yields the edges in an initial segment of the
90: (8)                     fundamental cycle of `G` starting with the nontree edge (`u`,
91: (8)                     `v`) that includes all the edges up until the first node that
92: (8)                     appears in `visited`. The tree edges are given by the 'parent'
93: (8)                     node attribute. The `visited` set is updated to add each node in
94: (8)                     an edge yielded by this function.
95: (8)                     """
96: (8)                     while v not in visited:
97: (12)                        yield u, v
98: (12)                        visited.add(v)
99: (12)                        u, v = v, G.nodes[v]["parent"]
100: (8)                    yield u, v
101: (4)                if root is not None and root not in G:
102: (8)                    raise nx.NodeNotFound(f"Root node {root} is not in graph")
103: (4)                H, nodes = _dfs_cycle_forest(G, root)
104: (4)                visited = set()
105: (4)                for u in nodes:
106: (8)                    visited.add(u)
107: (8)                    edges = ((u, v) for u, v, d in H.out_edges(u, data="nontree") if d)
108: (8)                    for u, v in edges:
109: (12)                       chain = list(_build_chain(H, u, v, visited))
110: (12)                       yield chain
```

File 14 - clique.py:

```
1: (0)              """Functions for finding and manipulating cliques.
2: (0)              Finding the largest clique in a graph is NP-complete problem, so most of
3: (0)              these algorithms have an exponential running time; for more information,
4: (0)              see the Wikipedia article on the clique problem [1]_.
5: (0)              .. [1] clique problem:: https://en.wikipedia.org/wiki/Clique_problem
6: (0)              """
7: (0)              from collections import defaultdict, deque
8: (0)              from itertools import chain, combinations, islice
9: (0)              import networkx as nx
10: (0)             from networkx.utils import not_implemented_for
11: (0)             __all__ = [
12: (4)                 "find_cliques",
13: (4)                 "find_cliques_recursive",
14: (4)                 "make_max_clique_graph",
15: (4)                 "make_clique_bipartite",
16: (4)                 "node_clique_number",
17: (4)                 "number_of_cliques",
18: (4)                 "enumerate_all_cliques",
19: (4)                 "max_weight_clique",
20: (0)             ]
21: (0)             @not_implemented_for("directed")
22: (0)             @nx._dispatchable
23: (0)             def enumerate_all_cliques(G):
24: (4)                 """Returns all cliques in an undirected graph.
25: (4)                 This function returns an iterator over cliques, each of which is a
26: (4)                 list of nodes. The iteration is ordered by cardinality of the
27: (4)                 cliques: first all cliques of size one, then all cliques of size
28: (4)                 two, etc.
29: (4)                 Parameters
30: (4)                 ----------
31: (4)                 G : NetworkX graph
32: (8)                     An undirected graph.
33: (4)                 Returns
34: (4)                 -------
35: (4)                 iterator
36: (8)                     An iterator over cliques, each of which is a list of nodes in
37: (8)                     `G`. The cliques are ordered according to size.
38: (4)                 Notes
39: (4)                 -----
40: (4)                 To obtain a list of all cliques, use
41: (4)                 `list(enumerate_all_cliques(G))`. However, be aware that in the
42: (4)                 worst-case, the length of this list can be exponential in the number
```

```
43: (4)                 of nodes in the graph (for example, when the graph is the complete
44: (4)                 graph). This function avoids storing all cliques in memory by only
45: (4)                 keeping current candidate node lists in memory during its search.
46: (4)                 The implementation is adapted from the algorithm by Zhang, et
47: (4)                 al. (2005) [1]_ to output all cliques discovered.
48: (4)                 This algorithm ignores self-loops and parallel edges, since cliques
49: (4)                 are not conventionally defined with such edges.
50: (4)                 References
51: (4)                 ----------
52: (4)                 .. [1] Yun Zhang, Abu-Khzam, F.N., Baldwin, N.E., Chesler, E.J.,
53: (11)                       Langston, M.A., Samatova, N.F.,
54: (11)                       "Genome-Scale Computational Approaches to Memory-Intensive
55: (11)                       Applications in Systems Biology".
56: (11)                       *Supercomputing*, 2005. Proceedings of the ACM/IEEE SC 2005
57: (11)                       Conference, pp. 12, 12--18 Nov. 2005.
58: (11)                       <https://doi.org/10.1109/SC.2005.29>.
59: (4)                 """
60: (4)                 index = {}
61: (4)                 nbrs = {}
62: (4)                 for u in G:
63: (8)                     index[u] = len(index)
64: (8)                     nbrs[u] = {v for v in G[u] if v not in index}
65: (4)                 queue = deque(([u], sorted(nbrs[u], key=index.__getitem__)) for u in G)
66: (4)                 while queue:
67: (8)                     base, cnbrs = map(list, queue.popleft())
68: (8)                     yield base
69: (8)                     for i, u in enumerate(cnbrs):
70: (12)                        queue.append(
71: (16)                            (
72: (20)                                chain(base, [u]),
73: (20)                                filter(nbrs[u].__contains__, islice(cnbrs, i + 1, None)),
74: (16)                            )
75: (12)                        )
76: (0)             @not_implemented_for("directed")
77: (0)             @nx._dispatchable
78: (0)             def find_cliques(G, nodes=None):
79: (4)                 """Returns all maximal cliques in an undirected graph.
80: (4)                 For each node *n*, a *maximal clique for n* is a largest complete
81: (4)                 subgraph containing *n*. The largest maximal clique is sometimes
82: (4)                 called the *maximum clique*.
83: (4)                 This function returns an iterator over cliques, each of which is a
84: (4)                 list of nodes. It is an iterative implementation, so should not
85: (4)                 suffer from recursion depth issues.
86: (4)                 This function accepts a list of `nodes` and only the maximal cliques
87: (4)                 containing all of these `nodes` are returned. It can considerably speed up
88: (4)                 the running time if some specific cliques are desired.
89: (4)                 Parameters
90: (4)                 ----------
91: (4)                 G : NetworkX graph
92: (8)                     An undirected graph.
93: (4)                 nodes : list, optional (default=None)
94: (8)                     If provided, only yield *maximal cliques* containing all nodes in
`nodes`.
95: (8)                     If `nodes` isn't a clique itself, a ValueError is raised.
96: (4)                 Returns
97: (4)                 -------
98: (4)                 iterator
99: (8)                     An iterator over maximal cliques, each of which is a list of
100: (8)                    nodes in `G`. If `nodes` is provided, only the maximal cliques
101: (8)                    containing all the nodes in `nodes` are returned. The order of
102: (8)                    cliques is arbitrary.
103: (4)                Raises
104: (4)                ------
105: (4)                ValueError
106: (8)                    If `nodes` is not a clique.
107: (4)                Examples
108: (4)                --------
109: (4)                >>> from pprint import pprint  # For nice dict formatting
110: (4)                >>> G = nx.karate_club_graph()
```

```
(4)                >>> sum(1 for c in nx.find_cliques(G))  # The number of maximal cliques in
G
(4)                36
(4)                >>> max(nx.find_cliques(G), key=len)  # The largest maximal clique in G
(4)                [0, 1, 2, 3, 13]
(4)                The size of the largest maximal clique is known as the *clique number* of
(4)                the graph, which can be found directly with:
(4)                >>> max(len(c) for c in nx.find_cliques(G))
(4)                5
(4)                One can also compute the number of maximal cliques in `G` that contain a
given
(4)                node. The following produces a dictionary keyed by node whose
(4)                values are the number of maximal cliques in `G` that contain the node:
(4)                >>> pprint({n: sum(1 for c in nx.find_cliques(G) if n in c) for n in G})
(4)                {0: 13,
(5)                 1: 6,
(5)                 2: 7,
(5)                 3: 3,
(5)                 4: 2,
(5)                 5: 3,
(5)                 6: 3,
(5)                 7: 1,
(5)                 8: 3,
(5)                 9: 2,
(5)                 10: 2,
(5)                 11: 1,
(5)                 12: 1,
(5)                 13: 2,
(5)                 14: 1,
(5)                 15: 1,
(5)                 16: 1,
(5)                 17: 1,
(5)                 18: 1,
(5)                 19: 2,
(5)                 20: 1,
(5)                 21: 1,
(5)                 22: 1,
(5)                 23: 3,
(5)                 24: 2,
(5)                 25: 2,
(5)                 26: 1,
(5)                 27: 3,
(5)                 28: 2,
(5)                 29: 2,
(5)                 30: 2,
(5)                 31: 4,
(5)                 32: 9,
(5)                 33: 14}
(4)                Or, similarly, the maximal cliques in `G` that contain a given node.
(4)                For example, the 4 maximal cliques that contain node 31:
(4)                >>> [c for c in nx.find_cliques(G) if 31 in c]
(4)                [[0, 31], [33, 32, 31], [33, 28, 31], [24, 25, 31]]
(4)                See Also
(4)                --------
(4)                find_cliques_recursive
(8)                    A recursive version of the same algorithm.
(4)                Notes
(4)                -----
(4)                To obtain a list of all maximal cliques, use
(4)                `list(find_cliques(G))`. However, be aware that in the worst-case,
(4)                the length of this list can be exponential in the number of nodes in
(4)                the graph. This function avoids storing all cliques in memory by
(4)                only keeping current candidate node lists in memory during its search.
(4)                This implementation is based on the algorithm published by Bron and
(4)                Kerbosch (1973) [1]_, as adapted by Tomita, Tanaka and Takahashi
(4)                (2006) [2]_ and discussed in Cazals and Karande (2008) [3]_. It
(4)                essentially unrolls the recursion used in the references to avoid
(4)                issues of recursion stack depth (for a recursive implementation, see
(4)                :func:`find_cliques_recursive`).
```

```
178: (4)                This algorithm ignores self-loops and parallel edges, since cliques
179: (4)                are not conventionally defined with such edges.
180: (4)                References
181: (4)                ----------
182: (4)                .. [1] Bron, C. and Kerbosch, J.
183: (7)                   "Algorithm 457: finding all cliques of an undirected graph".
184: (7)                   *Communications of the ACM* 16, 9 (Sep. 1973), 575--577.
185: (7)                   <http://portal.acm.org/citation.cfm?doid=362342.362367>
186: (4)                .. [2] Etsuji Tomita, Akira Tanaka, Haruhisa Takahashi,
187: (7)                   "The worst-case time complexity for generating all maximal
188: (7)                   cliques and computational experiments",
189: (7)                   *Theoretical Computer Science*, Volume 363, Issue 1,
190: (7)                   Computing and Combinatorics,
191: (7)                   10th Annual International Conference on
192: (7)                   Computing and Combinatorics (COCOON 2004), 25 October 2006, Pages 28-
-42
193: (7)                   <https://doi.org/10.1016/j.tcs.2006.06.015>
194: (4)                .. [3] F. Cazals, C. Karande,
195: (7)                   "A note on the problem of reporting maximal cliques",
196: (7)                   *Theoretical Computer Science*,
197: (7)                   Volume 407, Issues 1--3, 6 November 2008, Pages 564--568,
198: (7)                   <https://doi.org/10.1016/j.tcs.2008.05.010>
199: (4)                """
200: (4)                if len(G) == 0:
201: (8)                    return
202: (4)                adj = {u: {v for v in G[u] if v != u} for u in G}
203: (4)                Q = nodes[:] if nodes is not None else []
204: (4)                cand = set(G)
205: (4)                for node in Q:
206: (8)                    if node not in cand:
207: (12)                       raise ValueError(f"The given `nodes` {nodes} do not form a
clique")
208: (8)                    cand &= adj[node]
209: (4)                if not cand:
210: (8)                    yield Q[:]
211: (8)                    return
212: (4)                subg = cand.copy()
213: (4)                stack = []
214: (4)                Q.append(None)
215: (4)                u = max(subg, key=lambda u: len(cand & adj[u]))
216: (4)                ext_u = cand - adj[u]
217: (4)                try:
218: (8)                    while True:
219: (12)                       if ext_u:
220: (16)                           q = ext_u.pop()
221: (16)                           cand.remove(q)
222: (16)                           Q[-1] = q
223: (16)                           adj_q = adj[q]
224: (16)                           subg_q = subg & adj_q
225: (16)                           if not subg_q:
226: (20)                               yield Q[:]
227: (16)                           else:
228: (20)                               cand_q = cand & adj_q
229: (20)                               if cand_q:
230: (24)                                   stack.append((subg, cand, ext_u))
231: (24)                                   Q.append(None)
232: (24)                                   subg = subg_q
233: (24)                                   cand = cand_q
234: (24)                                   u = max(subg, key=lambda u: len(cand & adj[u]))
235: (24)                                   ext_u = cand - adj[u]
236: (12)                       else:
237: (16)                           Q.pop()
238: (16)                           subg, cand, ext_u = stack.pop()
239: (4)                except IndexError:
240: (8)                    pass
241: (0)            @nx._dispatchable
242: (0)            def find_cliques_recursive(G, nodes=None):
243: (4)                """Returns all maximal cliques in a graph.
244: (4)                For each node *v*, a *maximal clique for v* is a largest complete
```

```
    subgraph containing *v*. The largest maximal clique is sometimes
    called the *maximum clique*.
    This function returns an iterator over cliques, each of which is a
    list of nodes. It is a recursive implementation, so may suffer from
    recursion depth issues, but is included for pedagogical reasons.
    For a non-recursive implementation, see :func:`find_cliques`.
    This function accepts a list of `nodes` and only the maximal cliques
    containing all of these `nodes` are returned. It can considerably speed up
    the running time if some specific cliques are desired.
    Parameters
    ----------
    G : NetworkX graph
    nodes : list, optional (default=None)
        If provided, only yield *maximal cliques* containing all nodes in `nodes`.

        If `nodes` isn't a clique itself, a ValueError is raised.
    Returns
    -------
    iterator
        An iterator over maximal cliques, each of which is a list of
        nodes in `G`. If `nodes` is provided, only the maximal cliques
        containing all the nodes in `nodes` are yielded. The order of
        cliques is arbitrary.
    Raises
    ------
    ValueError
        If `nodes` is not a clique.
    See Also
    --------
    find_cliques
        An iterative version of the same algorithm. See docstring for examples.
    Notes
    -----
    To obtain a list of all maximal cliques, use
    `list(find_cliques_recursive(G))`. However, be aware that in the
    worst-case, the length of this list can be exponential in the number
    of nodes in the graph. This function avoids storing all cliques in memory
    by only keeping current candidate node lists in memory during its search.
    This implementation is based on the algorithm published by Bron and
    Kerbosch (1973) [1]_, as adapted by Tomita, Tanaka and Takahashi
    (2006) [2]_ and discussed in Cazals and Karande (2008) [3]_. For a
    non-recursive implementation, see :func:`find_cliques`.
    This algorithm ignores self-loops and parallel edges, since cliques
    are not conventionally defined with such edges.
    References
    ----------
    .. [1] Bron, C. and Kerbosch, J.
       "Algorithm 457: finding all cliques of an undirected graph".
       *Communications of the ACM* 16, 9 (Sep. 1973), 575--577.
       <http://portal.acm.org/citation.cfm?doid=362342.362367>
    .. [2] Etsuji Tomita, Akira Tanaka, Haruhisa Takahashi,
       "The worst-case time complexity for generating all maximal
       cliques and computational experiments",
       *Theoretical Computer Science*, Volume 363, Issue 1,
       Computing and Combinatorics,
       10th Annual International Conference on
       Computing and Combinatorics (COCOON 2004), 25 October 2006, Pages 28--42

       <https://doi.org/10.1016/j.tcs.2006.06.015>
    .. [3] F. Cazals, C. Karande,
       "A note on the problem of reporting maximal cliques",
       *Theoretical Computer Science*,
       Volume 407, Issues 1--3, 6 November 2008, Pages 564--568,
       <https://doi.org/10.1016/j.tcs.2008.05.010>
    """
    if len(G) == 0:
        return iter([])
    adj = {u: {v for v in G[u] if v != u} for u in G}
```

```
(4)                 Q = nodes[:] if nodes is not None else []
(4)                 cand_init = set(G)
(4)                 for node in Q:
(8)                     if node not in cand_init:
(12)                        raise ValueError(f"The given `nodes` {nodes} do not form a clique")
(8)                     cand_init &= adj[node]
(4)                 if not cand_init:
(8)                     return iter([Q])
(4)                 subg_init = cand_init.copy()
(4)                 def expand(subg, cand):
(8)                     u = max(subg, key=lambda u: len(cand & adj[u]))
(8)                     for q in cand - adj[u]:
(12)                        cand.remove(q)
(12)                        Q.append(q)
(12)                        adj_q = adj[q]
(12)                        subg_q = subg & adj_q
(12)                        if not subg_q:
(16)                            yield Q[:]
(12)                        else:
(16)                            cand_q = cand & adj_q
(16)                            if cand_q:
(20)                                yield from expand(subg_q, cand_q)
(12)                        Q.pop()
(4)                 return expand(subg_init, cand_init)
(0)             @nx._dispatchable(returns_graph=True)
(0)             def make_max_clique_graph(G, create_using=None):
(4)                 """Returns the maximal clique graph of the given graph.
(4)                 The nodes of the maximal clique graph of `G` are the cliques of
(4)                 `G` and an edge joins two cliques if the cliques are not disjoint.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 NetworkX graph
(8)                     A graph whose nodes are the cliques of `G` and whose edges
(8)                     join two cliques if they are not disjoint.
(4)                 Notes
(4)                 -----
(4)                 This function behaves like the following code::
(8)                     import networkx as nx
(8)                     G = nx.make_clique_bipartite(G)
(8)                     cliques = [v for v in G.nodes() if G.nodes[v]["bipartite"] == 0]
(8)                     G = nx.bipartite.projected_graph(G, cliques)
(8)                     G = nx.relabel_nodes(G, {-v: v - 1 for v in G})
(4)                 It should be faster, though, since it skips all the intermediate
(4)                 steps.
(4)                 """
(4)                 if create_using is None:
(8)                     B = G.__class__()
(4)                 else:
(8)                     B = nx.empty_graph(0, create_using)
(4)                 cliques = list(enumerate(set(c) for c in find_cliques(G)))
(4)                 B.add_nodes_from(i for i, c in cliques)
(4)                 clique_pairs = combinations(cliques, 2)
(4)                 B.add_edges_from((i, j) for (i, c1), (j, c2) in clique_pairs if c1 & c2)
(4)                 return B
(0)             @nx._dispatchable(returns_graph=True)
(0)             def make_clique_bipartite(G, fpos=None, create_using=None, name=None):
(4)                 """Returns the bipartite clique graph corresponding to `G`.
(4)                 In the returned bipartite graph, the "bottom" nodes are the nodes of
(4)                 `G` and the "top" nodes represent the maximal cliques of `G`.
(4)                 There is an edge from node *v* to clique *C* in the returned graph
(4)                 if and only if *v* is an element of *C*.
(4)                 Parameters
(4)                 ----------
```

```
    G : NetworkX graph
        An undirected graph.
    fpos : bool
        If True or not None, the returned graph will have an
        additional attribute, `pos`, a dictionary mapping node to
        position in the Euclidean plane.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    NetworkX graph
        A bipartite graph whose "bottom" set is the nodes of the graph
        `G`, whose "top" set is the cliques of `G`, and whose edges
        join nodes of `G` to the cliques that contain them.
        The nodes of the graph `G` have the node attribute
        'bipartite' set to 1 and the nodes representing cliques
        have the node attribute 'bipartite' set to 0, as is the
        convention for bipartite graphs in NetworkX.
    """
    B = nx.empty_graph(0, create_using)
    B.clear()
    B.add_nodes_from(G, bipartite=1)
    for i, cl in enumerate(find_cliques(G)):
        name = -i - 1
        B.add_node(name, bipartite=0)
        B.add_edges_from((v, name) for v in cl)
    return B
@nx._dispatchable
def node_clique_number(G, nodes=None, cliques=None, separate_nodes=False):
    """Returns the size of the largest maximal clique containing each given node.
    Returns a single or list depending on input nodes.
    An optional list of cliques can be input if already computed.
    Parameters
    ----------
    G : NetworkX graph
        An undirected graph.
    cliques : list, optional (default=None)
        A list of cliques, each of which is itself a list of nodes.
        If not specified, the list of all cliques will be computed
        using :func:`find_cliques`.
    Returns
    -------
    int or dict
        If `nodes` is a single node, returns the size of the
        largest maximal clique in `G` containing that node.
        Otherwise return a dict keyed by node to the size
        of the largest maximal clique containing that node.
    See Also
    --------
    find_cliques
        find_cliques yields the maximal cliques of G.
        It accepts a `nodes` argument which restricts consideration to
        maximal cliques containing all the given `nodes`.
        The search for the cliques is optimized for `nodes`.
    """
    if cliques is None:
        if nodes is not None:
            if nodes in G:
                return max(len(c) for c in find_cliques(nx.ego_graph(G, nodes)))
            return {
                n: max(len(c) for c in find_cliques(nx.ego_graph(G, n))) for n in nodes
            }
        cliques = list(find_cliques(G))
    if nodes in G:
        return max(len(c) for c in cliques if nodes in c)
    size_for_n = defaultdict(int)
```

```
445: (4)                for c in cliques:
446: (8)                    size_of_c = len(c)
447: (8)                    for n in c:
448: (12)                       if size_for_n[n] < size_of_c:
449: (16)                           size_for_n[n] = size_of_c
450: (4)                if nodes is None:
451: (8)                    return size_for_n
452: (4)                return {n: size_for_n[n] for n in nodes}
453: (0)            def number_of_cliques(G, nodes=None, cliques=None):
454: (4)                """Returns the number of maximal cliques for each node.
455: (4)                Returns a single or list depending on input nodes.
456: (4)                Optional list of cliques can be input if already computed.
457: (4)                """
458: (4)                if cliques is None:
459: (8)                    cliques = list(find_cliques(G))
460: (4)                if nodes is None:
461: (8)                    nodes = list(G.nodes())  # none, get entire graph
462: (4)                if not isinstance(nodes, list):  # check for a list
463: (8)                    v = nodes
464: (8)                    numcliq = len([1 for c in cliques if v in c])
465: (4)                else:
466: (8)                    numcliq = {}
467: (8)                    for v in nodes:
468: (12)                       numcliq[v] = len([1 for c in cliques if v in c])
469: (4)                return numcliq
470: (0)            class MaxWeightClique:
471: (4)                """A class for the maximum weight clique algorithm.
472: (4)                This class is a helper for the `max_weight_clique` function.  The class
473: (4)                should not normally be used directly.
474: (4)                Parameters
475: (4)                ----------
476: (4)                G : NetworkX graph
477: (8)                    The undirected graph for which a maximum weight clique is sought
478: (4)                weight : string or None, optional (default='weight')
479: (8)                    The node attribute that holds the integer value used as a weight.
480: (8)                    If None, then each node has weight 1.
481: (4)                Attributes
482: (4)                ----------
483: (4)                G : NetworkX graph
484: (8)                    The undirected graph for which a maximum weight clique is sought
485: (4)                node_weights: dict
486: (8)                    The weight of each node
487: (4)                incumbent_nodes : list
488: (8)                    The nodes of the incumbent clique (the best clique found so far)
489: (4)                incumbent_weight: int
490: (8)                    The weight of the incumbent clique
491: (4)                """
492: (4)                def __init__(self, G, weight):
493: (8)                    self.G = G
494: (8)                    self.incumbent_nodes = []
495: (8)                    self.incumbent_weight = 0
496: (8)                    if weight is None:
497: (12)                       self.node_weights = {v: 1 for v in G.nodes()}
498: (8)                    else:
499: (12)                       for v in G.nodes():
500: (16)                           if weight not in G.nodes[v]:
501: (20)                               errmsg = f"Node {v!r} does not have the requested weight
field."
502: (20)                               raise KeyError(errmsg)
503: (16)                           if not isinstance(G.nodes[v][weight], int):
504: (20)                               errmsg = f"The {weight!r} field of node {v!r} is not an
integer."
505: (20)                               raise ValueError(errmsg)
506: (12)                       self.node_weights = {v: G.nodes[v][weight] for v in G.nodes()}
507: (4)                def update_incumbent_if_improved(self, C, C_weight):
508: (8)                    """Update the incumbent if the node set C has greater weight.
509: (8)                    C is assumed to be a clique.
510: (8)                    """
511: (8)                    if C_weight > self.incumbent_weight:
```

```
(12)                        self.incumbent_nodes = C[:]
(12)                        self.incumbent_weight = C_weight
(4)                 def greedily_find_independent_set(self, P):
(8)                     """Greedily find an independent set of nodes from a set of
(8)                     nodes P."""
(8)                     independent_set = []
(8)                     P = P[:]
(8)                     while P:
(12)                        v = P[0]
(12)                        independent_set.append(v)
(12)                        P = [w for w in P if v != w and not self.G.has_edge(v, w)]
(8)                     return independent_set
(4)                 def find_branching_nodes(self, P, target):
(8)                     """Find a set of nodes to branch on."""
(8)                     residual_wt = {v: self.node_weights[v] for v in P}
(8)                     total_wt = 0
(8)                     P = P[:]
(8)                     while P:
(12)                        independent_set = self.greedily_find_independent_set(P)
(12)                        min_wt_in_class = min(residual_wt[v] for v in independent_set)
(12)                        total_wt += min_wt_in_class
(12)                        if total_wt > target:
(16)                            break
(12)                        for v in independent_set:
(16)                            residual_wt[v] -= min_wt_in_class
(12)                        P = [v for v in P if residual_wt[v] != 0]
(8)                     return P
(4)                 def expand(self, C, C_weight, P):
(8)                     """Look for the best clique that contains all the nodes in C and zero
or
(8)                     more of the nodes in P, backtracking if it can be shown that no such
(8)                     clique has greater weight than the incumbent.
(8)                     """
(8)                     self.update_incumbent_if_improved(C, C_weight)
(8)                     branching_nodes = self.find_branching_nodes(P, self.incumbent_weight -
C_weight)
(8)                     while branching_nodes:
(12)                        v = branching_nodes.pop()
(12)                        P.remove(v)
(12)                        new_C = C + [v]
(12)                        new_C_weight = C_weight + self.node_weights[v]
(12)                        new_P = [w for w in P if self.G.has_edge(v, w)]
(12)                        self.expand(new_C, new_C_weight, new_P)
(4)                 def find_max_weight_clique(self):
(8)                     """Find a maximum weight clique."""
(8)                     nodes = sorted(self.G.nodes(), key=lambda v: self.G.degree(v),
reverse=True)
(8)                     nodes = [v for v in nodes if self.node_weights[v] > 0]
(8)                     self.expand([], 0, nodes)
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable(node_attrs="weight")
(0)             def max_weight_clique(G, weight="weight"):
(4)                 """Find a maximum weight clique in G.
(4)                 A *clique* in a graph is a set of nodes such that every two distinct nodes
(4)                 are adjacent.  The *weight* of a clique is the sum of the weights of its
(4)                 nodes.  A *maximum weight clique* of graph G is a clique C in G such that
(4)                 no clique in G has weight greater than the weight of C.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Undirected graph
(4)                 weight : string or None, optional (default='weight')
(8)                     The node attribute that holds the integer value used as a weight.
(8)                     If None, then each node has weight 1.
(4)                 Returns
(4)                 -------
(4)                 clique : list
(8)                     the nodes of a maximum weight clique
(4)                 weight : int
```

```
578: (8)                the weight of a maximum weight clique
579: (4)            Notes
580: (4)            -----
581: (4)            The implementation is recursive, and therefore it may run into recursion
582: (4)            depth issues if G contains a clique whose number of nodes is close to the
583: (4)            recursion depth limit.
584: (4)            At each search node, the algorithm greedily constructs a weighted
585: (4)            independent set cover of part of the graph in order to find a small set of
586: (4)            nodes on which to branch.  The algorithm is very similar to the algorithm
587: (4)            of Tavares et al. [1]_, other than the fact that the NetworkX version does
588: (4)            not use bitsets.  This style of algorithm for maximum weight clique (and
589: (4)            maximum weight independent set, which is the same problem but on the
590: (4)            complement graph) has a decades-long history.  See Algorithm B of Warren
591: (4)            and Hicks [2]_ and the references in that paper.
592: (4)            References
593: (4)            ----------
594: (4)            .. [1] Tavares, W.A., Neto, M.B.C., Rodrigues, C.D., Michelon, P.: Um
595: (11)                  algoritmo de branch and bound para o problema da clique máxima
596: (11)                  ponderada.  Proceedings of XLVII SBPO 1 (2015).
597: (4)            .. [2] Warren, Jeffrey S, Hicks, Illya V.: Combinatorial Branch-and-Bound
598: (11)                  for the Maximum Weight Independent Set Problem.  Technical Report,
599: (11)                  Texas A&M University (2016).
600: (4)            """
601: (4)            mwc = MaxWeightClique(G, weight)
602: (4)            mwc.find_max_weight_clique()
603: (4)            return mwc.incumbent_nodes, mwc.incumbent_weight

----------------------------------------

File 15 - chordal.py:

1: (0)          """
2: (0)          Algorithms for chordal graphs.
3: (0)          A graph is chordal if every cycle of length at least 4 has a chord
4: (0)          (an edge joining two nodes not adjacent in the cycle).
5: (0)          https://en.wikipedia.org/wiki/Chordal_graph
6: (0)          """
7: (0)          import sys
8: (0)          import networkx as nx
9: (0)          from networkx.algorithms.components import connected_components
10: (0)         from networkx.utils import arbitrary_element, not_implemented_for
11: (0)         __all__ = [
12: (4)             "is_chordal",
13: (4)             "find_induced_nodes",
14: (4)             "chordal_graph_cliques",
15: (4)             "chordal_graph_treewidth",
16: (4)             "NetworkXTreewidthBoundExceeded",
17: (4)             "complete_to_chordal_graph",
18: (0)         ]
19: (0)         class NetworkXTreewidthBoundExceeded(nx.NetworkXException):
20: (4)             """Exception raised when a treewidth bound has been provided and it has
21: (4)             been exceeded"""
22: (0)         @not_implemented_for("directed")
23: (0)         @not_implemented_for("multigraph")
24: (0)         @nx._dispatchable
25: (0)         def is_chordal(G):
26: (4)             """Checks whether G is a chordal graph.
27: (4)             A graph is chordal if every cycle of length at least 4 has a chord
28: (4)             (an edge joining two nodes not adjacent in the cycle).
29: (4)             Parameters
30: (4)             ----------
31: (4)             G : graph
32: (6)               A NetworkX graph.
33: (4)             Returns
34: (4)             -------
35: (4)             chordal : bool
36: (6)               True if G is a chordal graph and False otherwise.
37: (4)             Raises
38: (4)             ------
```

```
(4)                 NetworkXNotImplemented
(8)                     The algorithm does not support DiGraph, MultiGraph and MultiDiGraph.
(4)                 Examples
(4)                 --------
(4)                 >>> e = [
(4)                 ...     (1, 2),
(4)                 ...     (1, 3),
(4)                 ...     (2, 3),
(4)                 ...     (2, 4),
(4)                 ...     (3, 4),
(4)                 ...     (3, 5),
(4)                 ...     (3, 6),
(4)                 ...     (4, 5),
(4)                 ...     (4, 6),
(4)                 ...     (5, 6),
(4)                 ... ]
(4)                 >>> G = nx.Graph(e)
(4)                 >>> nx.is_chordal(G)
(4)                 True
(4)                 Notes
(4)                 -----
(4)                 The routine tries to go through every node following maximum cardinality
(4)                 search. It returns False when it finds that the separator for any node
(4)                 is not a clique.  Based on the algorithms in [1]_.
(4)                 Self loops are ignored.
(4)                 References
(4)                 ----------
(4)                 .. [1] R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms
(7)                    to test chordality of graphs, test acyclicity of hypergraphs, and
(7)                    selectively reduce acyclic hypergraphs, SIAM J. Comput., 13 (1984),
(7)                    pp. 566–579.
(4)                 """
(4)                 if len(G.nodes) <= 3:
(8)                     return True
(4)                 return len(_find_chordality_breaker(G)) == 0
(0)             @nx._dispatchable
(0)             def find_induced_nodes(G, s, t, treewidth_bound=sys.maxsize):
(4)                 """Returns the set of induced nodes in the path from s to t.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A chordal NetworkX graph
(4)                 s : node
(8)                     Source node to look for induced nodes
(4)                 t : node
(8)                     Destination node to look for induced nodes
(4)                 treewidth_bound: float
(8)                     Maximum treewidth acceptable for the graph H. The search
(8)                     for induced nodes will end as soon as the treewidth_bound is exceeded.
(4)                 Returns
(4)                 -------
(4)                 induced_nodes : Set of nodes
(8)                     The set of induced nodes in the path from s to t in G
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     The algorithm does not support DiGraph, MultiGraph and MultiDiGraph.
(8)                     If the input graph is an instance of one of these classes, a
(8)                     :exc:`NetworkXError` is raised.
(8)                     The algorithm can only be applied to chordal graphs. If the input
(8)                     graph is found to be non-chordal, a :exc:`NetworkXError` is raised.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> G = nx.generators.classic.path_graph(10)
(4)                 >>> induced_nodes = nx.find_induced_nodes(G, 1, 9, 2)
(4)                 >>> sorted(induced_nodes)
(4)                 [1, 2, 3, 4, 5, 6, 7, 8, 9]
(4)                 Notes
```

```
    -----
    G must be a chordal graph and (s,t) an edge that is not in G.
    If a treewidth_bound is provided, the search for induced nodes will end
    as soon as the treewidth_bound is exceeded.
    The algorithm is inspired by Algorithm 4 in [1]_.
    A formal definition of induced node can also be found on that reference.
    Self Loops are ignored
    References
    ----------
    .. [1] Learning Bounded Treewidth Bayesian Networks.
       Gal Elidan, Stephen Gould; JMLR, 9(Dec):2699--2731, 2008.
       http://jmlr.csail.mit.edu/papers/volume9/elidan08a/elidan08a.pdf
    """
    if not is_chordal(G):
        raise nx.NetworkXError("Input graph is not chordal.")
    H = nx.Graph(G)
    H.add_edge(s, t)
    induced_nodes = set()
    triplet = _find_chordality_breaker(H, s, treewidth_bound)
    while triplet:
        (u, v, w) = triplet
        induced_nodes.update(triplet)
        for n in triplet:
            if n != s:
                H.add_edge(s, n)
        triplet = _find_chordality_breaker(H, s, treewidth_bound)
    if induced_nodes:
        induced_nodes.add(t)
        for u in G[s]:
            if len(induced_nodes & set(G[u])) == 2:
                induced_nodes.add(u)
                break
    return induced_nodes
@nx._dispatchable
def chordal_graph_cliques(G):
    """Returns all maximal cliques of a chordal graph.
    The algorithm breaks the graph in connected components and performs a
    maximum cardinality search in each component to get the cliques.
    Parameters
    ----------
    G : graph
      A NetworkX graph
    Yields
    ------
    frozenset of nodes
        Maximal cliques, each of which is a frozenset of
        nodes in `G`. The order of cliques is arbitrary.
    Raises
    ------
    NetworkXError
        The algorithm does not support DiGraph, MultiGraph and MultiDiGraph.
        The algorithm can only be applied to chordal graphs. If the input
        graph is found to be non-chordal, a :exc:`NetworkXError` is raised.
    Examples
    --------
    >>> e = [
    ...     (1, 2),
    ...     (1, 3),
    ...     (2, 3),
    ...     (2, 4),
    ...     (3, 4),
    ...     (3, 5),
    ...     (3, 6),
    ...     (4, 5),
    ...     (4, 6),
    ...     (5, 6),
    ...     (7, 8),
    ... ]
    >>> G = nx.Graph(e)
```

```
177: (4)                >>> G.add_node(9)
178: (4)                >>> cliques = [c for c in chordal_graph_cliques(G)]
179: (4)                >>> cliques[0]
180: (4)                frozenset({1, 2, 3})
181: (4)                """
182: (4)                for C in (G.subgraph(c).copy() for c in connected_components(G)):
183: (8)                    if C.number_of_nodes() == 1:
184: (12)                       if nx.number_of_selfloops(C) > 0:
185: (16)                           raise nx.NetworkXError("Input graph is not chordal.")
186: (12)                       yield frozenset(C.nodes())
187: (8)                    else:
188: (12)                       unnumbered = set(C.nodes())
189: (12)                       v = arbitrary_element(C)
190: (12)                       unnumbered.remove(v)
191: (12)                       numbered = {v}
192: (12)                       clique_wanna_be = {v}
193: (12)                       while unnumbered:
194: (16)                           v = _max_cardinality_node(C, unnumbered, numbered)
195: (16)                           unnumbered.remove(v)
196: (16)                           numbered.add(v)
197: (16)                           new_clique_wanna_be = set(C.neighbors(v)) & numbered
198: (16)                           sg = C.subgraph(clique_wanna_be)
199: (16)                           if _is_complete_graph(sg):
200: (20)                               new_clique_wanna_be.add(v)
201: (20)                               if not new_clique_wanna_be >= clique_wanna_be:
202: (24)                                   yield frozenset(clique_wanna_be)
203: (20)                               clique_wanna_be = new_clique_wanna_be
204: (16)                           else:
205: (20)                               raise nx.NetworkXError("Input graph is not chordal.")
206: (12)                       yield frozenset(clique_wanna_be)
207: (0)            @nx._dispatchable
208: (0)            def chordal_graph_treewidth(G):
209: (4)                """Returns the treewidth of the chordal graph G.
210: (4)                Parameters
211: (4)                ----------
212: (4)                G : graph
213: (6)                  A NetworkX graph
214: (4)                Returns
215: (4)                -------
216: (4)                treewidth : int
217: (8)                    The size of the largest clique in the graph minus one.
218: (4)                Raises
219: (4)                ------
220: (4)                NetworkXError
221: (8)                    The algorithm does not support DiGraph, MultiGraph and MultiDiGraph.
222: (8)                    The algorithm can only be applied to chordal graphs. If the input
223: (8)                    graph is found to be non-chordal, a :exc:`NetworkXError` is raised.
224: (4)                Examples
225: (4)                --------
226: (4)                >>> e = [
227: (4)                ...     (1, 2),
228: (4)                ...     (1, 3),
229: (4)                ...     (2, 3),
230: (4)                ...     (2, 4),
231: (4)                ...     (3, 4),
232: (4)                ...     (3, 5),
233: (4)                ...     (3, 6),
234: (4)                ...     (4, 5),
235: (4)                ...     (4, 6),
236: (4)                ...     (5, 6),
237: (4)                ...     (7, 8),
238: (4)                ... ]
239: (4)                >>> G = nx.Graph(e)
240: (4)                >>> G.add_node(9)
241: (4)                >>> nx.chordal_graph_treewidth(G)
242: (4)                3
243: (4)                References
244: (4)                ----------
245: (4)                .. [1] https://en.wikipedia.org/wiki/Tree_decomposition#Treewidth
```

```
    """
    if not is_chordal(G):
        raise nx.NetworkXError("Input graph is not chordal.")
    max_clique = -1
    for clique in nx.chordal_graph_cliques(G):
        max_clique = max(max_clique, len(clique))
    return max_clique - 1
def _is_complete_graph(G):
    """Returns True if G is a complete graph."""
    if nx.number_of_selfloops(G) > 0:
        raise nx.NetworkXError("Self loop found in _is_complete_graph()")
    n = G.number_of_nodes()
    if n < 2:
        return True
    e = G.number_of_edges()
    max_edges = (n * (n - 1)) / 2
    return e == max_edges
def _find_missing_edge(G):
    """Given a non-complete graph G, returns a missing edge."""
    nodes = set(G)
    for u in G:
        missing = nodes - set(list(G[u].keys()) + [u])
        if missing:
            return (u, missing.pop())
def _max_cardinality_node(G, choices, wanna_connect):
    """Returns a the node in choices that has more connections in G
    to nodes in wanna_connect.
    """
    max_number = -1
    for x in choices:
        number = len([y for y in G[x] if y in wanna_connect])
        if number > max_number:
            max_number = number
            max_cardinality_node = x
    return max_cardinality_node
def _find_chordality_breaker(G, s=None, treewidth_bound=sys.maxsize):
    """Given a graph G, starts a max cardinality search
    (starting from s if s is given and from an arbitrary node otherwise)
    trying to find a non-chordal cycle.
    If it does find one, it returns (u,v,w) where u,v,w are the three
    nodes that together with s are involved in the cycle.
    It ignores any self loops.
    """
    if len(G) == 0:
        raise nx.NetworkXPointlessConcept("Graph has no nodes.")
    unnumbered = set(G)
    if s is None:
        s = arbitrary_element(G)
    unnumbered.remove(s)
    numbered = {s}
    current_treewidth = -1
    while unnumbered:  # and current_treewidth <= treewidth_bound:
        v = _max_cardinality_node(G, unnumbered, numbered)
        unnumbered.remove(v)
        numbered.add(v)
        clique_wanna_be = set(G[v]) & numbered
        sg = G.subgraph(clique_wanna_be)
        if _is_complete_graph(sg):
            current_treewidth = max(current_treewidth, len(clique_wanna_be))
            if current_treewidth > treewidth_bound:
                raise nx.NetworkXTreewidthBoundExceeded(
                    f"treewidth_bound exceeded: {current_treewidth}"
                )
        else:
            (u, w) = _find_missing_edge(sg)
            return (u, v, w)
    return ()
@not_implemented_for("directed")
@nx._dispatchable(returns_graph=True)
```

```
(0)             def complete_to_chordal_graph(G):
(4)                 """Return a copy of G completed to a chordal graph
(4)                 Adds edges to a copy of G to create a chordal graph. A graph G=(V,E) is
(4)                 called chordal if for each cycle with length bigger than 3, there exist
(4)                 two non-adjacent nodes connected by an edge (called a chord).
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Undirected graph
(4)                 Returns
(4)                 -------
(4)                 H : NetworkX graph
(8)                     The chordal enhancement of G
(4)                 alpha : Dictionary
(12)                        The elimination ordering of nodes of G
(4)                 Notes
(4)                 -----
(4)                 There are different approaches to calculate the chordal
(4)                 enhancement of a graph. The algorithm used here is called
(4)                 MCS-M and gives at least minimal (local) triangulation of graph. Note
(4)                 that this triangulation is not necessarily a global minimum.
(4)                 https://en.wikipedia.org/wiki/Chordal_graph
(4)                 References
(4)                 ----------
(4)                 .. [1] Berry, Anne & Blair, Jean & Heggernes, Pinar & Peyton, Barry.
(2004)
(11)                       Maximum Cardinality Search for Computing Minimal Triangulations of
(11)                       Graphs.  Algorithmica. 39. 287-298. 10.1007/s00453-004-1084-3.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms.chordal import complete_to_chordal_graph
(4)                 >>> G = nx.wheel_graph(10)
(4)                 >>> H, alpha = complete_to_chordal_graph(G)
(4)                 """
(4)                 H = G.copy()
(4)                 alpha = {node: 0 for node in H}
(4)                 if nx.is_chordal(H):
(8)                     return H, alpha
(4)                 chords = set()
(4)                 weight = {node: 0 for node in H.nodes()}
(4)                 unnumbered_nodes = list(H.nodes())
(4)                 for i in range(len(H.nodes()), 0, -1):
(8)                     z = max(unnumbered_nodes, key=lambda node: weight[node])
(8)                     unnumbered_nodes.remove(z)
(8)                     alpha[z] = i
(8)                     update_nodes = []
(8)                     for y in unnumbered_nodes:
(12)                        if G.has_edge(y, z):
(16)                            update_nodes.append(y)
(12)                        else:
(16)                            y_weight = weight[y]
(16)                            lower_nodes = [
(20)                                node for node in unnumbered_nodes if weight[node] <
y_weight
(16)                            ]
(16)                            if nx.has_path(H.subgraph(lower_nodes + [z, y]), y, z):
(20)                                update_nodes.append(y)
(20)                                chords.add((z, y))
(8)                     for node in update_nodes:
(12)                        weight[node] += 1
(4)                 H.add_edges_from(chords)
(4)                 return H, alpha
```

File 16 - cluster.py:

```
(0)             """Algorithms to characterize the number of triangles in a graph."""
(0)             from collections import Counter
```

```
3: (0)              from itertools import chain, combinations
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = [
7: (4)                  "triangles",
8: (4)                  "average_clustering",
9: (4)                  "clustering",
10: (4)                 "transitivity",
11: (4)                 "square_clustering",
12: (4)                 "generalized_degree",
13: (0)             ]
14: (0)             @not_implemented_for("directed")
15: (0)             @nx._dispatchable
16: (0)             def triangles(G, nodes=None):
17: (4)                 """Compute the number of triangles.
18: (4)                 Finds the number of triangles that include a node as one vertex.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : graph
22: (7)                    A networkx graph
23: (4)                 nodes : node, iterable of nodes, or None (default=None)
24: (8)                     If a singleton node, return the number of triangles for that node.
25: (8)                     If an iterable, compute the number of triangles for each of those
nodes.
26: (8)                     If `None` (the default) compute the number of triangles for all nodes
in `G`.
27: (4)                 Returns
28: (4)                 -------
29: (4)                 out : dict or int
30: (7)                    If `nodes` is a container of nodes, returns number of triangles keyed
by node (dict).
31: (7)                    If `nodes` is a specific node, returns number of triangles for the node
(int).
32: (4)                 Examples
33: (4)                 --------
34: (4)                 >>> G = nx.complete_graph(5)
35: (4)                 >>> print(nx.triangles(G, 0))
36: (4)                 6
37: (4)                 >>> print(nx.triangles(G))
38: (4)                 {0: 6, 1: 6, 2: 6, 3: 6, 4: 6}
39: (4)                 >>> print(list(nx.triangles(G, [0, 1]).values()))
40: (4)                 [6, 6]
41: (4)                 Notes
42: (4)                 -----
43: (4)                 Self loops are ignored.
44: (4)                 """
45: (4)                 if nodes is not None:
46: (8)                     if nodes in G:
47: (12)                        return next(_triangles_and_degree_iter(G, nodes))[2] // 2
48: (8)                     return {v: t // 2 for v, d, t, _ in _triangles_and_degree_iter(G,
nodes)}
49: (4)                 later_nbrs = {}
50: (4)                 for node, neighbors in G.adjacency():
51: (8)                     later_nbrs[node] = {n for n in neighbors if n not in later_nbrs and n
!= node}
52: (4)                 triangle_counts = Counter(dict.fromkeys(G, 0))
53: (4)                 for node1, neighbors in later_nbrs.items():
54: (8)                     for node2 in neighbors:
55: (12)                        third_nodes = neighbors & later_nbrs[node2]
56: (12)                        m = len(third_nodes)
57: (12)                        triangle_counts[node1] += m
58: (12)                        triangle_counts[node2] += m
59: (12)                        triangle_counts.update(third_nodes)
60: (4)                 return dict(triangle_counts)
61: (0)             @not_implemented_for("multigraph")
62: (0)             def _triangles_and_degree_iter(G, nodes=None):
63: (4)                 """Return an iterator of (node, degree, triangles, generalized degree).
64: (4)                 This double counts triangles so you may want to divide by 2.
65: (4)                 See degree(), triangles() and generalized_degree() for definitions
```

```
66: (4)                 and details.
67: (4)                 """
68: (4)                 if nodes is None:
69: (8)                     nodes_nbrs = G.adj.items()
70: (4)                 else:
71: (8)                     nodes_nbrs = ((n, G[n]) for n in G.nbunch_iter(nodes))
72: (4)                 for v, v_nbrs in nodes_nbrs:
73: (8)                     vs = set(v_nbrs) - {v}
74: (8)                     gen_degree = Counter(len(vs & (set(G[w]) - {w})) for w in vs)
75: (8)                     ntriangles = sum(k * val for k, val in gen_degree.items())
76: (8)                     yield (v, len(vs), ntriangles, gen_degree)
77: (0)             @not_implemented_for("multigraph")
78: (0)             def _weighted_triangles_and_degree_iter(G, nodes=None, weight="weight"):
79: (4)                 """Return an iterator of (node, degree, weighted_triangles).
80: (4)                 Used for weighted clustering.
81: (4)                 Note: this returns the geometric average weight of edges in the triangle.
82: (4)                 Also, each triangle is counted twice (each direction).
83: (4)                 So you may want to divide by 2.
84: (4)                 """
85: (4)                 import numpy as np
86: (4)                 if weight is None or G.number_of_edges() == 0:
87: (8)                     max_weight = 1
88: (4)                 else:
89: (8)                     max_weight = max(d.get(weight, 1) for u, v, d in G.edges(data=True))
90: (4)                 if nodes is None:
91: (8)                     nodes_nbrs = G.adj.items()
92: (4)                 else:
93: (8)                     nodes_nbrs = ((n, G[n]) for n in G.nbunch_iter(nodes))
94: (4)                 def wt(u, v):
95: (8)                     return G[u][v].get(weight, 1) / max_weight
96: (4)                 for i, nbrs in nodes_nbrs:
97: (8)                     inbrs = set(nbrs) - {i}
98: (8)                     weighted_triangles = 0
99: (8)                     seen = set()
100: (8)                    for j in inbrs:
101: (12)                       seen.add(j)
102: (12)                       jnbrs = set(G[j]) - seen
103: (12)                       wij = wt(i, j)
104: (12)                       weighted_triangles += np.cbrt(
105: (16)                           [(wij * wt(j, k) * wt(k, i)) for k in inbrs & jnbrs]
106: (12)                       ).sum()
107: (8)                    yield (i, len(inbrs), 2 * float(weighted_triangles))
108: (0)            @not_implemented_for("multigraph")
109: (0)            def _directed_triangles_and_degree_iter(G, nodes=None):
110: (4)                """Return an iterator of
111: (4)                (node, total_degree, reciprocal_degree, directed_triangles).
112: (4)                Used for directed clustering.
113: (4)                Note that unlike `_triangles_and_degree_iter()`, this function counts
114: (4)                directed triangles so does not count triangles twice.
115: (4)                """
116: (4)                nodes_nbrs = ((n, G._pred[n], G._succ[n]) for n in G.nbunch_iter(nodes))
117: (4)                for i, preds, succs in nodes_nbrs:
118: (8)                    ipreds = set(preds) - {i}
119: (8)                    isuccs = set(succs) - {i}
120: (8)                    directed_triangles = 0
121: (8)                    for j in chain(ipreds, isuccs):
122: (12)                       jpreds = set(G._pred[j]) - {j}
123: (12)                       jsuccs = set(G._succ[j]) - {j}
124: (12)                       directed_triangles += sum(
125: (16)                           1
126: (16)                           for k in chain(
127: (20)                               (ipreds & jpreds),
128: (20)                               (ipreds & jsuccs),
129: (20)                               (isuccs & jpreds),
130: (20)                               (isuccs & jsuccs),
131: (16)                           )
132: (12)                       )
133: (8)                    dtotal = len(ipreds) + len(isuccs)
134: (8)                    dbidirectional = len(ipreds & isuccs)
```

```
135: (8)                    yield (i, dtotal, dbidirectional, directed_triangles)
136: (0)            @not_implemented_for("multigraph")
137: (0)            def _directed_weighted_triangles_and_degree_iter(G, nodes=None,
weight="weight"):
138: (4)                """Return an iterator of
139: (4)                (node, total_degree, reciprocal_degree, directed_weighted_triangles).
140: (4)                Used for directed weighted clustering.
141: (4)                Note that unlike `_weighted_triangles_and_degree_iter()`, this function
counts
142: (4)                directed triangles so does not count triangles twice.
143: (4)                """
144: (4)                import numpy as np
145: (4)                if weight is None or G.number_of_edges() == 0:
146: (8)                    max_weight = 1
147: (4)                else:
148: (8)                    max_weight = max(d.get(weight, 1) for u, v, d in G.edges(data=True))
149: (4)                nodes_nbrs = ((n, G._pred[n], G._succ[n]) for n in G.nbunch_iter(nodes))
150: (4)                def wt(u, v):
151: (8)                    return G[u][v].get(weight, 1) / max_weight
152: (4)                for i, preds, succs in nodes_nbrs:
153: (8)                    ipreds = set(preds) - {i}
154: (8)                    isuccs = set(succs) - {i}
155: (8)                    directed_triangles = 0
156: (8)                    for j in ipreds:
157: (12)                       jpreds = set(G._pred[j]) - {j}
158: (12)                       jsuccs = set(G._succ[j]) - {j}
159: (12)                       directed_triangles += np.cbrt(
160: (16)                           [(wt(j, i) * wt(k, i) * wt(k, j)) for k in ipreds & jpreds]
161: (12)                       ).sum()
162: (12)                       directed_triangles += np.cbrt(
163: (16)                           [(wt(j, i) * wt(k, i) * wt(j, k)) for k in ipreds & jsuccs]
164: (12)                       ).sum()
165: (12)                       directed_triangles += np.cbrt(
166: (16)                           [(wt(j, i) * wt(i, k) * wt(k, j)) for k in isuccs & jpreds]
167: (12)                       ).sum()
168: (12)                       directed_triangles += np.cbrt(
169: (16)                           [(wt(j, i) * wt(i, k) * wt(j, k)) for k in isuccs & jsuccs]
170: (12)                       ).sum()
171: (8)                    for j in isuccs:
172: (12)                       jpreds = set(G._pred[j]) - {j}
173: (12)                       jsuccs = set(G._succ[j]) - {j}
174: (12)                       directed_triangles += np.cbrt(
175: (16)                           [(wt(i, j) * wt(k, i) * wt(k, j)) for k in ipreds & jpreds]
176: (12)                       ).sum()
177: (12)                       directed_triangles += np.cbrt(
178: (16)                           [(wt(i, j) * wt(k, i) * wt(j, k)) for k in ipreds & jsuccs]
179: (12)                       ).sum()
180: (12)                       directed_triangles += np.cbrt(
181: (16)                           [(wt(i, j) * wt(i, k) * wt(k, j)) for k in isuccs & jpreds]
182: (12)                       ).sum()
183: (12)                       directed_triangles += np.cbrt(
184: (16)                           [(wt(i, j) * wt(i, k) * wt(j, k)) for k in isuccs & jsuccs]
185: (12)                       ).sum()
186: (8)                    dtotal = len(ipreds) + len(isuccs)
187: (8)                    dbidirectional = len(ipreds & isuccs)
188: (8)                    yield (i, dtotal, dbidirectional, float(directed_triangles))
189: (0)            @nx._dispatchable(edge_attrs="weight")
190: (0)            def average_clustering(G, nodes=None, weight=None, count_zeros=True):
191: (4)                r"""Compute the average clustering coefficient for the graph G.
192: (4)                The clustering coefficient for the graph is the average,
193: (4)                .. math::
194: (7)                   C = \frac{1}{n}\sum_{v \in G} c_v,
195: (4)                where :math:`n` is the number of nodes in `G`.
196: (4)                Parameters
197: (4)                ----------
198: (4)                G : graph
199: (4)                nodes : container of nodes, optional (default=all nodes in G)
200: (7)                   Compute average clustering for nodes in this container.
201: (4)                weight : string or None, optional (default=None)
```

```
202: (7)                   The edge attribute that holds the numerical value used as a weight.
203: (7)                   If None, then each edge has weight 1.
204: (4)                count_zeros : bool
205: (7)                   If False include only the nodes with nonzero clustering in the average.
206: (4)                Returns
207: (4)                -------
208: (4)                avg : float
209: (7)                   Average clustering
210: (4)                Examples
211: (4)                --------
212: (4)                >>> G = nx.complete_graph(5)
213: (4)                >>> print(nx.average_clustering(G))
214: (4)                1.0
215: (4)                Notes
216: (4)                -----
217: (4)                This is a space saving routine; it might be faster
218: (4)                to use the clustering function to get a list and then take the average.
219: (4)                Self loops are ignored.
220: (4)                References
221: (4)                ----------
222: (4)                .. [1] Generalizations of the clustering coefficient to weighted
223: (7)                   complex networks by J. Saramäki, M. Kivelä, J.-P. Onnela,
224: (7)                   K. Kaski, and J. Kertész, Physical Review E, 75 027105 (2007).
225: (7)                   http://jponnela.com/web_documents/a9.pdf
226: (4)                .. [2] Marcus Kaiser,  Mean clustering coefficients: the role of isolated
227: (7)                   nodes and leafs on clustering measures for small-world networks.
228: (7)                   https://arxiv.org/abs/0802.2512
229: (4)                """
230: (4)                c = clustering(G, nodes, weight=weight).values()
231: (4)                if not count_zeros:
232: (8)                    c = [v for v in c if abs(v) > 0]
233: (4)                return sum(c) / len(c)
234: (0)            @nx._dispatchable(edge_attrs="weight")
235: (0)            def clustering(G, nodes=None, weight=None):
236: (4)                r"""Compute the clustering coefficient for nodes.
237: (4)                For unweighted graphs, the clustering of a node :math:`u`
238: (4)                is the fraction of possible triangles through that node that exist,
239: (4)                .. math::
240: (6)                  c_u = \frac{2 T(u)}{deg(u)(deg(u)-1)},
241: (4)                where :math:`T(u)` is the number of triangles through node :math:`u` and
242: (4)                :math:`deg(u)` is the degree of :math:`u`.
243: (4)                For weighted graphs, there are several ways to define clustering [1]_.
244: (4)                the one used here is defined
245: (4)                as the geometric average of the subgraph edge weights [2]_,
246: (4)                .. math::
247: (7)                   c_u = \frac{1}{deg(u)(deg(u)-1))}
248: (13)                        \sum_{vw} (\hat{w}_{uv} \hat{w}_{uw} \hat{w}_{vw})^{1/3}.
249: (4)                The edge weights :math:`\hat{w}_{uv}` are normalized by the maximum weight
250: (4)                in the network :math:`\hat{w}_{uv} = w_{uv}/\max(w)`.
251: (4)                The value of :math:`c_u` is assigned to 0 if :math:`deg(u) < 2`.
252: (4)                Additionally, this weighted definition has been generalized to support
negative edge weights [3]_.
253: (4)                For directed graphs, the clustering is similarly defined as the fraction
254: (4)                of all possible directed triangles or geometric average of the subgraph
255: (4)                edge weights for unweighted and weighted directed graph respectively [4]_.
256: (4)                .. math::
257: (7)                   c_u = \frac{T(u)}{2(deg^{tot}(u)(deg^{tot}(u)-1) -
2deg^{\leftrightarrow}(u))},
258: (4)                where :math:`T(u)` is the number of directed triangles through node
259: (4)                :math:`u`, :math:`deg^{tot}(u)` is the sum of in degree and out degree of
260: (4)                :math:`u` and :math:`deg^{\leftrightarrow}(u)` is the reciprocal degree of
261: (4)                :math:`u`.
262: (4)                Parameters
263: (4)                ----------
264: (4)                G : graph
265: (4)                nodes : node, iterable of nodes, or None (default=None)
266: (8)                    If a singleton node, return the number of triangles for that node.
267: (8)                    If an iterable, compute the number of triangles for each of those
nodes.
```

```
268: (8)                    If `None` (the default) compute the number of triangles for all nodes
in `G`.
269: (4)                weight : string or None, optional (default=None)
270: (7)                   The edge attribute that holds the numerical value used as a weight.
271: (7)                   If None, then each edge has weight 1.
272: (4)                Returns
273: (4)                -------
274: (4)                out : float, or dictionary
275: (7)                   Clustering coefficient at specified nodes
276: (4)                Examples
277: (4)                --------
278: (4)                >>> G = nx.complete_graph(5)
279: (4)                >>> print(nx.clustering(G, 0))
280: (4)                1.0
281: (4)                >>> print(nx.clustering(G))
282: (4)                {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0}
283: (4)                Notes
284: (4)                -----
285: (4)                Self loops are ignored.
286: (4)                References
287: (4)                ----------
288: (4)                .. [1] Generalizations of the clustering coefficient to weighted
289: (7)                   complex networks by J. Saramäki, M. Kivelä, J.-P. Onnela,
290: (7)                   K. Kaski, and J. Kertész, Physical Review E, 75 027105 (2007).
291: (7)                   http://jponnela.com/web_documents/a9.pdf
292: (4)                .. [2] Intensity and coherence of motifs in weighted complex
293: (7)                   networks by J. P. Onnela, J. Saramäki, J. Kertész, and K. Kaski,
294: (7)                   Physical Review E, 71(6), 065103 (2005).
295: (4)                .. [3] Generalization of Clustering Coefficients to Signed Correlation
Networks
296: (7)                   by G. Costantini and M. Perugini, PloS one, 9(2), e88669 (2014).
297: (4)                .. [4] Clustering in complex directed networks by G. Fagiolo,
298: (7)                   Physical Review E, 76(2), 026107 (2007).
299: (4)                """
300: (4)                if G.is_directed():
301: (8)                    if weight is not None:
302: (12)                       td_iter = _directed_weighted_triangles_and_degree_iter(G, nodes,
weight)
303: (12)                       clusterc = {
304: (16)                           v: 0 if t == 0 else t / ((dt * (dt - 1) - 2 * db) * 2)
305: (16)                           for v, dt, db, t in td_iter
306: (12)                       }
307: (8)                    else:
308: (12)                       td_iter = _directed_triangles_and_degree_iter(G, nodes)
309: (12)                       clusterc = {
310: (16)                           v: 0 if t == 0 else t / ((dt * (dt - 1) - 2 * db) * 2)
311: (16)                           for v, dt, db, t in td_iter
312: (12)                       }
313: (4)                else:
314: (8)                    if weight is not None:
315: (12)                       td_iter = _weighted_triangles_and_degree_iter(G, nodes, weight)
316: (12)                       clusterc = {v: 0 if t == 0 else t / (d * (d - 1)) for v, d, t in
td_iter}
317: (8)                    else:
318: (12)                       td_iter = _triangles_and_degree_iter(G, nodes)
319: (12)                       clusterc = {v: 0 if t == 0 else t / (d * (d - 1)) for v, d, t, _
in td_iter}
320: (4)                if nodes in G:
321: (8)                    return clusterc[nodes]
322: (4)                return clusterc
323: (0)            @nx._dispatchable
324: (0)            def transitivity(G):
325: (4)                r"""Compute graph transitivity, the fraction of all possible triangles
326: (4)                present in G.
327: (4)                Possible triangles are identified by the number of "triads"
328: (4)                (two edges with a shared vertex).
329: (4)                The transitivity is
330: (4)                .. math::
331: (8)                    T = 3\frac{\#triangles}{\#triads}.
```

```
    Parameters
    ----------
    G : graph
    Returns
    -------
    out : float
       Transitivity
    Notes
    -----
    Self loops are ignored.
    Examples
    --------
    >>> G = nx.complete_graph(5)
    >>> print(nx.transitivity(G))
    1.0
    """
    triangles_contri = [
        (t, d * (d - 1)) for v, d, t, _ in _triangles_and_degree_iter(G)
    ]
    if len(triangles_contri) == 0:
        return 0
    triangles, contri = map(sum, zip(*triangles_contri))
    return 0 if triangles == 0 else triangles / contri
@nx._dispatchable
def square_clustering(G, nodes=None):
    r"""Compute the squares clustering coefficient for nodes.
    For each node return the fraction of possible squares that exist at
    the node [1]_
    .. math::
       C_4(v) = \frac{ \sum_{u=1}^{k_v}
       \sum_{w=u+1}^{k_v} q_v(u,w) }{ \sum_{u=1}^{k_v}
       \sum_{w=u+1}^{k_v} [a_v(u,w) + q_v(u,w)]},
    where :math:`q_v(u,w)` are the number of common neighbors of :math:`u` and
    :math:`w` other than :math:`v` (ie squares), and :math:`a_v(u,w) = (k_u -
    (1+q_v(u,w)+\theta_{uv})) + (k_w - (1+q_v(u,w)+\theta_{uw}))`, where
    :math:`\theta_{uw} = 1` if :math:`u` and :math:`w` are connected and 0
    otherwise. [2]_
    Parameters
    ----------
    G : graph
    nodes : container of nodes, optional (default=all nodes in G)
       Compute clustering for nodes in this container.
    Returns
    -------
    c4 : dictionary
       A dictionary keyed by node with the square clustering coefficient
value.
    Examples
    --------
    >>> G = nx.complete_graph(5)
    >>> print(nx.square_clustering(G, 0))
    1.0
    >>> print(nx.square_clustering(G))
    {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0}
    Notes
    -----
    While :math:`C_3(v)` (triangle clustering) gives the probability that
    two neighbors of node v are connected with each other, :math:`C_4(v)` is
    the probability that two neighbors of node v share a common
    neighbor different from v. This algorithm can be applied to both
    bipartite and unipartite networks.
    References
    ----------
    .. [1] Pedro G. Lind, Marta C. González, and Hans J. Herrmann. 2005
        Cycles and clustering in bipartite networks.
        Physical Review E (72) 056127.
    .. [2] Zhang, Peng et al. Clustering Coefficient and Community Structure
of
        Bipartite Networks. Physica A: Statistical Mechanics and its
```

```
Applications 387.27 (2008): 6869–6875.
399: (8)                    https://arxiv.org/abs/0710.0117v1
400: (4)                """
401: (4)                if nodes is None:
402: (8)                    node_iter = G
403: (4)                else:
404: (8)                    node_iter = G.nbunch_iter(nodes)
405: (4)                clustering = {}
406: (4)                for v in node_iter:
407: (8)                    clustering[v] = 0
408: (8)                    potential = 0
409: (8)                    for u, w in combinations(G[v], 2):
410: (12)                       squares = len((set(G[u]) & set(G[w])) - {v})
411: (12)                       clustering[v] += squares
412: (12)                       degm = squares + 1
413: (12)                       if w in G[u]:
414: (16)                           degm += 1
415: (12)                       potential += (len(G[u]) - degm) + (len(G[w]) - degm) + squares
416: (8)                    if potential > 0:
417: (12)                       clustering[v] /= potential
418: (4)                if nodes in G:
419: (8)                    return clustering[nodes]
420: (4)                return clustering
421: (0)            @not_implemented_for("directed")
422: (0)            @nx._dispatchable
423: (0)            def generalized_degree(G, nodes=None):
424: (4)                r"""Compute the generalized degree for nodes.
425: (4)                For each node, the generalized degree shows how many edges of given
426: (4)                triangle multiplicity the node is connected to. The triangle multiplicity
427: (4)                of an edge is the number of triangles an edge participates in. The
428: (4)                generalized degree of node :math:`i` can be written as a vector
429: (4)                :math:`\mathbf{k}_i=(k_i^{(0)}, \dotsc, k_i^{(N-2)})` where
430: (4)                :math:`k_i^{(j)}` is the number of edges attached to node :math:`i` that
431: (4)                participate in :math:`j` triangles.
432: (4)                Parameters
433: (4)                ----------
434: (4)                G : graph
435: (4)                nodes : container of nodes, optional (default=all nodes in G)
436: (7)                   Compute the generalized degree for nodes in this container.
437: (4)                Returns
438: (4)                -------
439: (4)                out : Counter, or dictionary of Counters
440: (7)                   Generalized degree of specified nodes. The Counter is keyed by edge
441: (7)                   triangle multiplicity.
442: (4)                Examples
443: (4)                --------
444: (4)                >>> G = nx.complete_graph(5)
445: (4)                >>> print(nx.generalized_degree(G, 0))
446: (4)                Counter({3: 4})
447: (4)                >>> print(nx.generalized_degree(G))
448: (4)                {0: Counter({3: 4}), 1: Counter({3: 4}), 2: Counter({3: 4}), 3:
Counter({3: 4}), 4: Counter({3: 4})}
449: (4)                To recover the number of triangles attached to a node:
450: (4)                >>> k1 = nx.generalized_degree(G, 0)
451: (4)                >>> sum([k * v for k, v in k1.items()]) / 2 == nx.triangles(G, 0)
452: (4)                True
453: (4)                Notes
454: (4)                -----
455: (4)                Self loops are ignored.
456: (4)                In a network of N nodes, the highest triangle multiplicity an edge can
have
457: (4)                is N-2.
458: (4)                The return value does not include a `zero` entry if no edges of a
459: (4)                particular triangle multiplicity are present.
460: (4)                The number of triangles node :math:`i` is attached to can be recovered
from
461: (4)                the generalized degree :math:`\mathbf{k}_i=(k_i^{(0)}, \dotsc,
462: (4)                k_i^{(N-2)})` by :math:`(k_i^{(1)}+2k_i^{(2)}+\dotsc +(N-2)k_i^{(N-
2)})/2`.
```

```
463: (4)                References
464: (4)                ----------
465: (4)                .. [1] Networks with arbitrary edge multiplicities by V. Zlatić,
466: (8)                    D. Garlaschelli and G. Caldarelli, EPL (Europhysics Letters),
467: (8)                    Volume 97, Number 2 (2012).
468: (8)                    https://iopscience.iop.org/article/10.1209/0295-5075/97/28005
469: (4)                """
470: (4)                if nodes in G:
471: (8)                    return next(_triangles_and_degree_iter(G, nodes))[3]
472: (4)                return {v: gd for v, d, t, gd in _triangles_and_degree_iter(G, nodes)}
```

File 17 - communicability_alg.py:

```
1: (0)              """
2: (0)              Communicability.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = ["communicability", "communicability_exp"]
7: (0)              @not_implemented_for("directed")
8: (0)              @not_implemented_for("multigraph")
9: (0)              @nx._dispatchable
10: (0)             def communicability(G):
11: (4)                 r"""Returns communicability between all pairs of nodes in G.
12: (4)                 The communicability between pairs of nodes in G is the sum of
13: (4)                 walks of different lengths starting at node u and ending at node v.
14: (4)                 Parameters
15: (4)                 ----------
16: (4)                 G: graph
17: (4)                 Returns
18: (4)                 -------
19: (4)                 comm: dictionary of dictionaries
20: (8)                     Dictionary of dictionaries keyed by nodes with communicability
21: (8)                     as the value.
22: (4)                 Raises
23: (4)                 ------
24: (4)                 NetworkXError
25: (7)                    If the graph is not undirected and simple.
26: (4)                 See Also
27: (4)                 --------
28: (4)                 communicability_exp:
29: (7)                    Communicability between all pairs of nodes in G  using spectral
30: (7)                    decomposition.
31: (4)                 communicability_betweenness_centrality:
32: (7)                    Communicability betweenness centrality for each node in G.
33: (4)                 Notes
34: (4)                 -----
35: (4)                 This algorithm uses a spectral decomposition of the adjacency matrix.
36: (4)                 Let G=(V,E) be a simple undirected graph.  Using the connection between
37: (4)                 the powers  of the adjacency matrix and the number of walks in the graph,
38: (4)                 the communicability  between nodes `u` and `v` based on the graph spectrum
39: (4)                 is [1]_
40: (4)                 .. math::
41: (8)                     C(u,v)=\sum_{j=1}^{n}\phi_{j}(u)\phi_{j}(v)e^{\lambda_{j}},
42: (4)                 where `\phi_{j}(u)` is the `u\rm{th}` element of the `j\rm{th}`
orthonormal
43: (4)                 eigenvector of the adjacency matrix associated with the eigenvalue
44: (4)                 `\lambda_{j}`.
45: (4)                 References
46: (4)                 ----------
47: (4)                 .. [1] Ernesto Estrada, Naomichi Hatano,
48: (7)                    "Communicability in complex networks",
49: (7)                    Phys. Rev. E 77, 036111 (2008).
50: (7)                    https://arxiv.org/abs/0707.0756
51: (4)                 Examples
52: (4)                 --------
53: (4)                 >>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3),
```

```
(3, 6)])
    >>> c = nx.communicability(G)
    """
    import numpy as np
    nodelist = list(G)  # ordering of nodes in matrix
    A = nx.to_numpy_array(G, nodelist)
    A[A != 0.0] = 1
    w, vec = np.linalg.eigh(A)
    expw = np.exp(w)
    mapping = dict(zip(nodelist, range(len(nodelist))))
    c = {}
    for u in G:
        c[u] = {}
        for v in G:
            s = 0
            p = mapping[u]
            q = mapping[v]
            for j in range(len(nodelist)):
                s += vec[:, j][p] * vec[:, j][q] * expw[j]
            c[u][v] = float(s)
    return c
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable
def communicability_exp(G):
    r"""Returns communicability between all pairs of nodes in G.
    Communicability between pair of node (u,v) of node in G is the sum of
    walks of different lengths starting at node u and ending at node v.
    Parameters
    ----------
    G: graph
    Returns
    -------
    comm: dictionary of dictionaries
        Dictionary of dictionaries keyed by nodes with communicability
        as the value.
    Raises
    ------
    NetworkXError
        If the graph is not undirected and simple.
    See Also
    --------
    communicability:
       Communicability between pairs of nodes in G.
    communicability_betweenness_centrality:
       Communicability betweenness centrality for each node in G.
    Notes
    -----
    This algorithm uses matrix exponentiation of the adjacency matrix.
    Let G=(V,E) be a simple undirected graph.  Using the connection between
    the powers  of the adjacency matrix and the number of walks in the graph,
    the communicability between nodes u and v is [1]_,
    .. math::
        C(u,v) = (e^A)_{uv},
    where `A` is the adjacency matrix of G.
    References
    ----------
    .. [1] Ernesto Estrada, Naomichi Hatano,
       "Communicability in complex networks",
       Phys. Rev. E 77, 036111 (2008).
       https://arxiv.org/abs/0707.0756
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3),
(3, 6)])
    >>> c = nx.communicability_exp(G)
    """
    import scipy as sp
    nodelist = list(G)  # ordering of nodes in matrix
```

```
121: (4)                A = nx.to_numpy_array(G, nodelist)
122: (4)                A[A != 0.0] = 1
123: (4)                expA = sp.linalg.expm(A)
124: (4)                mapping = dict(zip(nodelist, range(len(nodelist))))
125: (4)                c = {}
126: (4)                for u in G:
127: (8)                    c[u] = {}
128: (8)                    for v in G:
129: (12)                       c[u][v] = float(expA[mapping[u], mapping[v]])
130: (4)                return c
```

File 18 - core.py:

```
1: (0)              """
2: (0)              Find the k-cores of a graph.
3: (0)              The k-core is found by recursively pruning nodes with degrees less than k.
4: (0)              See the following references for details:
5: (0)              An O(m) Algorithm for Cores Decomposition of Networks
6: (0)              Vladimir Batagelj and Matjaz Zaversnik, 2003.
7: (0)              https://arxiv.org/abs/cs.DS/0310049
8: (0)              Generalized Cores
9: (0)              Vladimir Batagelj and Matjaz Zaversnik, 2002.
10: (0)             https://arxiv.org/pdf/cs/0202039
11: (0)             For directed graphs a more general notion is that of D-cores which
12: (0)             looks at (k, l) restrictions on (in, out) degree. The (k, k) D-core
13: (0)             is the k-core.
14: (0)             D-cores: Measuring Collaboration of Directed Graphs Based on Degeneracy
15: (0)             Christos Giatsidis, Dimitrios M. Thilikos, Michalis Vazirgiannis, ICDM 2011.
16: (0)             http://www.graphdegeneracy.org/dcores_ICDM_2011.pdf
17: (0)             Multi-scale structure and topological anomaly detection via a new network \
18: (0)             statistic: The onion decomposition
19: (0)             L. Hébert-Dufresne, J. A. Grochow, and A. Allard
20: (0)             Scientific Reports 6, 31708 (2016)
21: (0)             http://doi.org/10.1038/srep31708
22: (0)             """
23: (0)             import networkx as nx
24: (0)             __all__ = [
25: (4)                 "core_number",
26: (4)                 "k_core",
27: (4)                 "k_shell",
28: (4)                 "k_crust",
29: (4)                 "k_corona",
30: (4)                 "k_truss",
31: (4)                 "onion_layers",
32: (0)             ]
33: (0)             @nx.utils.not_implemented_for("multigraph")
34: (0)             @nx._dispatchable
35: (0)             def core_number(G):
36: (4)                 """Returns the core number for each node.
37: (4)                 A k-core is a maximal subgraph that contains nodes of degree k or more.
38: (4)                 The core number of a node is the largest value k of a k-core containing
39: (4)                 that node.
40: (4)                 Parameters
41: (4)                 ----------
42: (4)                 G : NetworkX graph
43: (7)                    An undirected or directed graph
44: (4)                 Returns
45: (4)                 -------
46: (4)                 core_number : dictionary
47: (7)                    A dictionary keyed by node to the core number.
48: (4)                 Raises
49: (4)                 ------
50: (4)                 NetworkXNotImplemented
51: (8)                     If `G` is a multigraph or contains self loops.
52: (4)                 Notes
53: (4)                 -----
54: (4)                 For directed graphs the node degree is defined to be the
```

```
 55: (4)                in-degree + out-degree.
 56: (4)                Examples
 57: (4)                --------
 58: (4)                >>> degrees = [0, 1, 2, 2, 2, 2, 3]
 59: (4)                >>> H = nx.havel_hakimi_graph(degrees)
 60: (4)                >>> nx.core_number(H)
 61: (4)                {0: 1, 1: 2, 2: 2, 3: 2, 4: 1, 5: 2, 6: 0}
 62: (4)                >>> G = nx.DiGraph()
 63: (4)                >>> G.add_edges_from([(1, 2), (2, 1), (2, 3), (2, 4), (3, 4), (4, 3)])
 64: (4)                >>> nx.core_number(G)
 65: (4)                {1: 2, 2: 2, 3: 2, 4: 2}
 66: (4)                References
 67: (4)                ----------
 68: (4)                .. [1] An O(m) Algorithm for Cores Decomposition of Networks
 69: (7)                   Vladimir Batagelj and Matjaz Zaversnik, 2003.
 70: (7)                   https://arxiv.org/abs/cs.DS/0310049
 71: (4)                """
 72: (4)                if nx.number_of_selfloops(G) > 0:
 73: (8)                    msg = (
 74: (12)                       "Input graph has self loops which is not permitted; "
 75: (12)                       "Consider using G.remove_edges_from(nx.selfloop_edges(G))."
 76: (8)                    )
 77: (8)                    raise nx.NetworkXNotImplemented(msg)
 78: (4)                degrees = dict(G.degree())
 79: (4)                nodes = sorted(degrees, key=degrees.get)
 80: (4)                bin_boundaries = [0]
 81: (4)                curr_degree = 0
 82: (4)                for i, v in enumerate(nodes):
 83: (8)                    if degrees[v] > curr_degree:
 84: (12)                       bin_boundaries.extend([i] * (degrees[v] - curr_degree))
 85: (12)                       curr_degree = degrees[v]
 86: (4)                node_pos = {v: pos for pos, v in enumerate(nodes)}
 87: (4)                core = degrees
 88: (4)                nbrs = {v: list(nx.all_neighbors(G, v)) for v in G}
 89: (4)                for v in nodes:
 90: (8)                    for u in nbrs[v]:
 91: (12)                       if core[u] > core[v]:
 92: (16)                           nbrs[u].remove(v)
 93: (16)                           pos = node_pos[u]
 94: (16)                           bin_start = bin_boundaries[core[u]]
 95: (16)                           node_pos[u] = bin_start
 96: (16)                           node_pos[nodes[bin_start]] = pos
 97: (16)                           nodes[bin_start], nodes[pos] = nodes[pos], nodes[bin_start]
 98: (16)                           bin_boundaries[core[u]] += 1
 99: (16)                           core[u] -= 1
100: (4)                return core
101: (0)            def _core_subgraph(G, k_filter, k=None, core=None):
102: (4)                """Returns the subgraph induced by nodes passing filter `k_filter`.
103: (4)                Parameters
104: (4)                ----------
105: (4)                G : NetworkX graph
106: (7)                   The graph or directed graph to process
107: (4)                k_filter : filter function
108: (7)                   This function filters the nodes chosen. It takes three inputs:
109: (7)                   A node of G, the filter's cutoff, and the core dict of the graph.
110: (7)                   The function should return a Boolean value.
111: (4)                k : int, optional
112: (6)                  The order of the core. If not specified use the max core number.
113: (6)                  This value is used as the cutoff for the filter.
114: (4)                core : dict, optional
115: (6)                  Precomputed core numbers keyed by node for the graph `G`.
116: (6)                  If not specified, the core numbers will be computed from `G`.
117: (4)                """
118: (4)                if core is None:
119: (8)                    core = core_number(G)
120: (4)                if k is None:
121: (8)                    k = max(core.values())
122: (4)                nodes = (v for v in core if k_filter(v, k, core))
123: (4)                return G.subgraph(nodes).copy()
```

```
(0)            @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
(0)            def k_core(G, k=None, core_number=None):
(4)                """Returns the k-core of G.
(4)                A k-core is a maximal subgraph that contains nodes of degree `k` or more.
(4)                .. deprecated:: 3.3
(7)                   `k_core` will not accept `MultiGraph` objects in version 3.5.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX graph
(6)                  A graph or directed graph
(4)                k : int, optional
(6)                  The order of the core. If not specified return the main core.
(4)                core_number : dictionary, optional
(6)                  Precomputed core numbers for the graph G.
(4)                Returns
(4)                -------
(4)                G : NetworkX graph
(6)                  The k-core subgraph
(4)                Raises
(4)                ------
(4)                NetworkXNotImplemented
(6)                  The k-core is not defined for multigraphs or graphs with self loops.
(4)                Notes
(4)                -----
(4)                The main core is the core with `k` as the largest core_number.
(4)                For directed graphs the node degree is defined to be the
(4)                in-degree + out-degree.
(4)                Graph, node, and edge attributes are copied to the subgraph.
(4)                Examples
(4)                --------
(4)                >>> degrees = [0, 1, 2, 2, 2, 2, 3]
(4)                >>> H = nx.havel_hakimi_graph(degrees)
(4)                >>> H.degree
(4)                DegreeView({0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 0})
(4)                >>> nx.k_core(H).nodes
(4)                NodeView((1, 2, 3, 5))
(4)                See Also
(4)                --------
(4)                core_number
(4)                References
(4)                ----------
(4)                .. [1] An O(m) Algorithm for Cores Decomposition of Networks
(7)                   Vladimir Batagelj and Matjaz Zaversnik,  2003.
(7)                   https://arxiv.org/abs/cs.DS/0310049
(4)                """
(4)                import warnings
(4)                if G.is_multigraph():
(8)                    warnings.warn(
(12)                       (
(16)                           "\n\n`k_core` will not accept `MultiGraph` objects in version
3.5.\n"
(16)                           "Convert it to an undirected graph instead, using::\n\n"
(16)                           "\tG = nx.Graph(G)\n"
(12)                       ),
(12)                       category=DeprecationWarning,
(12)                       stacklevel=5,
(8)                    )
(4)                def k_filter(v, k, c):
(8)                    return c[v] >= k
(4)                return _core_subgraph(G, k_filter, k, core_number)
(0)            @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
(0)            def k_shell(G, k=None, core_number=None):
(4)                """Returns the k-shell of G.
(4)                The k-shell is the subgraph induced by nodes with core number k.
(4)                That is, nodes in the k-core that are not in the (k+1)-core.
(4)                .. deprecated:: 3.3
(7)                   `k_shell` will not accept `MultiGraph` objects in version 3.5.
(4)                Parameters
(4)                ----------
```

```
    G : NetworkX graph
      A graph or directed graph.
    k : int, optional
      The order of the shell. If not specified return the outer shell.
    core_number : dictionary, optional
      Precomputed core numbers for the graph G.
    Returns
    -------
    G : NetworkX graph
       The k-shell subgraph
    Raises
    ------
    NetworkXNotImplemented
        The k-shell is not implemented for multigraphs or graphs with self loops.
    Notes
    -----
    This is similar to k_corona but in that case only neighbors in the
    k-core are considered.
    For directed graphs the node degree is defined to be the
    in-degree + out-degree.
    Graph, node, and edge attributes are copied to the subgraph.
    Examples
    --------
    >>> degrees = [0, 1, 2, 2, 2, 2, 3]
    >>> H = nx.havel_hakimi_graph(degrees)
    >>> H.degree
    DegreeView({0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 0})
    >>> nx.k_shell(H, k=1).nodes
    NodeView((0, 4))
    See Also
    --------
    core_number
    k_corona
    References
    ----------
    .. [1] A model of Internet topology using k-shell decomposition
       Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt,
       and Eran Shir, PNAS  July 3, 2007   vol. 104  no. 27  11150-11154
       http://www.pnas.org/content/104/27/11150.full
    """
    import warnings
    if G.is_multigraph():
        warnings.warn(
            (
                "\n\n`k_shell` will not accept `MultiGraph` objects in version 3.5.\n"
                "Convert it to an undirected graph instead, using::\n\n"
                "\tG = nx.Graph(G)\n"
            ),
            category=DeprecationWarning,
            stacklevel=5,
        )
    def k_filter(v, k, c):
        return c[v] == k
    return _core_subgraph(G, k_filter, k, core_number)
@nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
def k_crust(G, k=None, core_number=None):
    """Returns the k-crust of G.
    The k-crust is the graph G with the edges of the k-core removed
    and isolated nodes found after the removal of edges are also removed.
    .. deprecated:: 3.3
       `k_crust` will not accept `MultiGraph` objects in version 3.5.
    Parameters
    ----------
    G : NetworkX graph
       A graph or directed graph.
    k : int, optional
      The order of the shell. If not specified return the main crust.
```

```
(4)                 core_number : dictionary, optional
(6)                   Precomputed core numbers for the graph G.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX graph
(7)                    The k-crust subgraph
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     The k-crust is not implemented for multigraphs or graphs with self loops.
(4)                 Notes
(4)                 -----
(4)                 This definition of k-crust is different than the definition in [1]_.
(4)                 The k-crust in [1]_ is equivalent to the k+1 crust of this algorithm.
(4)                 For directed graphs the node degree is defined to be the
(4)                 in-degree + out-degree.
(4)                 Graph, node, and edge attributes are copied to the subgraph.
(4)                 Examples
(4)                 --------
(4)                 >>> degrees = [0, 1, 2, 2, 2, 2, 3]
(4)                 >>> H = nx.havel_hakimi_graph(degrees)
(4)                 >>> H.degree
(4)                 DegreeView({0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 0})
(4)                 >>> nx.k_crust(H, k=1).nodes
(4)                 NodeView((0, 4, 6))
(4)                 See Also
(4)                 --------
(4)                 core_number
(4)                 References
(4)                 ----------
(4)                 .. [1] A model of Internet topology using k-shell decomposition
(7)                    Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt,
(7)                    and Eran Shir, PNAS  July 3, 2007   vol. 104  no. 27  11150-11154
(7)                    http://www.pnas.org/content/104/27/11150.full
(4)                 """
(4)                 import warnings
(4)                 if G.is_multigraph():
(8)                     warnings.warn(
(12)                        (
(16)                            "\n\n`k_crust` will not accept `MultiGraph` objects in version 3.5.\n"
(16)                            "Convert it to an undirected graph instead, using::\n\n"
(16)                            "\tG = nx.Graph(G)\n"
(12)                        ),
(12)                        category=DeprecationWarning,
(12)                        stacklevel=5,
(8)                     )
(4)                 if core_number is None:
(8)                     core_number = nx.core_number(G)
(4)                 if k is None:
(8)                     k = max(core_number.values()) - 1
(4)                 nodes = (v for v in core_number if core_number[v] <= k)
(4)                 return G.subgraph(nodes).copy()
(0)             @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
(0)             def k_corona(G, k, core_number=None):
(4)                 """Returns the k-corona of G.
(4)                 The k-corona is the subgraph of nodes in the k-core which have
(4)                 exactly k neighbors in the k-core.
(4)                 .. deprecated:: 3.3
(7)                    `k_corona` will not accept `MultiGraph` objects in version 3.5.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    A graph or directed graph
(4)                 k : int
(7)                    The order of the corona.
(4)                 core_number : dictionary, optional
(7)                    Precomputed core numbers for the graph G.
```

```
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX graph
(7)                    The k-corona subgraph
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     The k-corona is not defined for multigraphs or graphs with self loops.
(4)                 Notes
(4)                 -----
(4)                 For directed graphs the node degree is defined to be the
(4)                 in-degree + out-degree.
(4)                 Graph, node, and edge attributes are copied to the subgraph.
(4)                 Examples
(4)                 --------
(4)                 >>> degrees = [0, 1, 2, 2, 2, 2, 3]
(4)                 >>> H = nx.havel_hakimi_graph(degrees)
(4)                 >>> H.degree
(4)                 DegreeView({0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 0})
(4)                 >>> nx.k_corona(H, k=2).nodes
(4)                 NodeView((1, 2, 3, 5))
(4)                 See Also
(4)                 --------
(4)                 core_number
(4)                 References
(4)                 ----------
(4)                 .. [1]  k -core (bootstrap) percolation on complex networks:
(7)                    Critical phenomena and nonlocal effects,
(7)                    A. V. Goltsev, S. N. Dorogovtsev, and J. F. F. Mendes,
(7)                    Phys. Rev. E 73, 056101 (2006)
(7)                    http://link.aps.org/doi/10.1103/PhysRevE.73.056101
(4)                 """
(4)                 import warnings
(4)                 if G.is_multigraph():
(8)                     warnings.warn(
(12)                        (
(16)                            "\n\n`k_corona` will not accept `MultiGraph` objects in version 3.5.\n"
(16)                            "Convert it to an undirected graph instead, using::\n\n"
(16)                            "\tG = nx.Graph(G)\n"
(12)                        ),
(12)                        category=DeprecationWarning,
(12)                        stacklevel=5,
(8)                     )
(4)                 def func(v, k, c):
(8)                     return c[v] == k and k == sum(1 for w in G[v] if c[w] >= k)
(4)                 return _core_subgraph(G, func, k, core_number)
(0)             @nx.utils.not_implemented_for("directed")
(0)             @nx.utils.not_implemented_for("multigraph")
(0)             @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
(0)             def k_truss(G, k):
(4)                 """Returns the k-truss of `G`.
(4)                 The k-truss is the maximal induced subgraph of `G` which contains at least
(4)                 three vertices where every edge is incident to at least `k-2` triangles.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(6)                   An undirected graph
(4)                 k : int
(6)                   The order of the truss
(4)                 Returns
(4)                 -------
(4)                 H : NetworkX graph
(6)                   The k-truss subgraph
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(6)                   If `G` is a multigraph or directed graph or if it contains self loops.
(4)                 Notes
```

```
    -----
    A k-clique is a (k-2)-truss and a k-truss is a (k+1)-core.
    Graph, node, and edge attributes are copied to the subgraph.
    K-trusses were originally defined in [2] which states that the k-truss
    is the maximal induced subgraph where each edge belongs to at least
    `k-2` triangles. A more recent paper, [1], uses a slightly different
    definition requiring that each edge belong to at least `k` triangles.
    This implementation uses the original definition of `k-2` triangles.
    Examples
    --------
    >>> degrees = [0, 1, 2, 2, 2, 2, 3]
    >>> H = nx.havel_hakimi_graph(degrees)
    >>> H.degree
    DegreeView({0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 0})
    >>> nx.k_truss(H, k=2).nodes
    NodeView((0, 1, 2, 3, 4, 5))
    References
    ----------
    .. [1] Bounds and Algorithms for k-truss. Paul Burkhardt, Vance Faber,
       David G. Harris, 2018. https://arxiv.org/abs/1806.05523v2
    .. [2] Trusses: Cohesive Subgraphs for Social Network Analysis. Jonathan
       Cohen, 2005.
    """
    if nx.number_of_selfloops(G) > 0:
        msg = (
            "Input graph has self loops which is not permitted; "
            "Consider using G.remove_edges_from(nx.selfloop_edges(G))."
        )
        raise nx.NetworkXNotImplemented(msg)
    H = G.copy()
    n_dropped = 1
    while n_dropped > 0:
        n_dropped = 0
        to_drop = []
        seen = set()
        for u in H:
            nbrs_u = set(H[u])
            seen.add(u)
            new_nbrs = [v for v in nbrs_u if v not in seen]
            for v in new_nbrs:
                if len(nbrs_u & set(H[v])) < (k - 2):
                    to_drop.append((u, v))
        H.remove_edges_from(to_drop)
        n_dropped = len(to_drop)
        H.remove_nodes_from(list(nx.isolates(H)))
    return H
@nx.utils.not_implemented_for("multigraph")
@nx.utils.not_implemented_for("directed")
@nx._dispatchable
def onion_layers(G):
    """Returns the layer of each vertex in an onion decomposition of the graph.

    The onion decomposition refines the k-core decomposition by providing
    information on the internal organization of each k-shell. It is usually
    used alongside the `core numbers`.
    Parameters
    ----------
    G : NetworkX graph
        An undirected graph without self loops.
    Returns
    -------
    od_layers : dictionary
        A dictionary keyed by node to the onion layer. The layers are
        contiguous integers starting at 1.
    Raises
    ------
    NetworkXNotImplemented
        If `G` is a multigraph or directed graph or if it contains self loops.
    Examples
```

```
462: (4)                --------
463: (4)                >>> degrees = [0, 1, 2, 2, 2, 2, 3]
464: (4)                >>> H = nx.havel_hakimi_graph(degrees)
465: (4)                >>> H.degree
466: (4)                DegreeView({0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 0})
467: (4)                >>> nx.onion_layers(H)
468: (4)                {6: 1, 0: 2, 4: 3, 1: 4, 2: 4, 3: 4, 5: 4}
469: (4)                See Also
470: (4)                --------
471: (4)                core_number
472: (4)                References
473: (4)                ----------
474: (4)                .. [1] Multi-scale structure and topological anomaly detection via a new
475: (7)                   network statistic: The onion decomposition
476: (7)                   L. Hébert-Dufresne, J. A. Grochow, and A. Allard
477: (7)                   Scientific Reports 6, 31708 (2016)
478: (7)                   http://doi.org/10.1038/srep31708
479: (4)                .. [2] Percolation and the effective structure of complex networks
480: (7)                   A. Allard and L. Hébert-Dufresne
481: (7)                   Physical Review X 9, 011023 (2019)
482: (7)                   http://doi.org/10.1103/PhysRevX.9.011023
483: (4)                """
484: (4)                if nx.number_of_selfloops(G) > 0:
485: (8)                    msg = (
486: (12)                       "Input graph contains self loops which is not permitted; "
487: (12)                       "Consider using G.remove_edges_from(nx.selfloop_edges(G))."
488: (8)                    )
489: (8)                    raise nx.NetworkXNotImplemented(msg)
490: (4)                od_layers = {}
491: (4)                neighbors = {v: list(nx.all_neighbors(G, v)) for v in G}
492: (4)                degrees = dict(G.degree())
493: (4)                current_core = 1
494: (4)                current_layer = 1
495: (4)                isolated_nodes = list(nx.isolates(G))
496: (4)                if len(isolated_nodes) > 0:
497: (8)                    for v in isolated_nodes:
498: (12)                       od_layers[v] = current_layer
499: (12)                       degrees.pop(v)
500: (8)                    current_layer = 2
501: (4)                while len(degrees) > 0:
502: (8)                    nodes = sorted(degrees, key=degrees.get)
503: (8)                    min_degree = degrees[nodes[0]]
504: (8)                    if min_degree > current_core:
505: (12)                       current_core = min_degree
506: (8)                    this_layer = []
507: (8)                    for n in nodes:
508: (12)                       if degrees[n] > current_core:
509: (16)                           break
510: (12)                       this_layer.append(n)
511: (8)                    for v in this_layer:
512: (12)                       od_layers[v] = current_layer
513: (12)                       for n in neighbors[v]:
514: (16)                           neighbors[n].remove(v)
515: (16)                           degrees[n] = degrees[n] - 1
516: (12)                       degrees.pop(v)
517: (8)                    current_layer = current_layer + 1
518: (4)                return od_layers
```

File 19 - covering.py:

```
1: (0)              """ Functions related to graph covers."""
2: (0)              from functools import partial
3: (0)              from itertools import chain
4: (0)              import networkx as nx
5: (0)              from networkx.utils import arbitrary_element, not_implemented_for
6: (0)              __all__ = ["min_edge_cover", "is_edge_cover"]
7: (0)              @not_implemented_for("directed")
```

```
8: (0)              @not_implemented_for("multigraph")
9: (0)              @nx._dispatchable
10: (0)             def min_edge_cover(G, matching_algorithm=None):
11: (4)                 """Returns the min cardinality edge cover of the graph as a set of edges.
12: (4)                 A smallest edge cover can be found in polynomial time by finding
13: (4)                 a maximum matching and extending it greedily so that all nodes
14: (4)                 are covered. This function follows that process. A maximum matching
15: (4)                 algorithm can be specified for the first step of the algorithm.
16: (4)                 The resulting set may return a set with one 2-tuple for each edge,
17: (4)                 (the usual case) or with both 2-tuples `(u, v)` and `(v, u)` for
18: (4)                 each edge. The latter is only done when a bipartite matching algorithm
19: (4)                 is specified as `matching_algorithm`.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : NetworkX graph
23: (8)                     An undirected graph.
24: (4)                 matching_algorithm : function
25: (8)                     A function that returns a maximum cardinality matching for `G`.
26: (8)                     The function must take one input, the graph `G`, and return
27: (8)                     either a set of edges (with only one direction for the pair of nodes)
28: (8)                     or a dictionary mapping each node to its mate. If not specified,
29: (8)                     :func:`~networkx.algorithms.matching.max_weight_matching` is used.
30: (8)                     Common bipartite matching functions include
31: (8)                     :func:`~networkx.algorithms.bipartite.matching.hopcroft_karp_matching`
32: (8)                     or
33: (8)                     :func:`~networkx.algorithms.bipartite.matching.eppstein_matching`.
34: (4)                 Returns
35: (4)                 -------
36: (4)                 min_cover : set
37: (8)                     A set of the edges in a minimum edge cover in the form of tuples.
38: (8)                     It contains only one of the equivalent 2-tuples `(u, v)` and `(v, u)`
39: (8)                     for each edge. If a bipartite method is used to compute the matching,
40: (8)                     the returned set contains both the 2-tuples `(u, v)` and `(v, u)`
41: (8)                     for each edge of a minimum edge cover.
42: (4)                 Examples
43: (4)                 --------
44: (4)                 >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
45: (4)                 >>> sorted(nx.min_edge_cover(G))
46: (4)                 [(2, 1), (3, 0)]
47: (4)                 Notes
48: (4)                 -----
49: (4)                 An edge cover of a graph is a set of edges such that every node of
50: (4)                 the graph is incident to at least one edge of the set.
51: (4)                 The minimum edge cover is an edge covering of smallest cardinality.
52: (4)                 Due to its implementation, the worst-case running time of this algorithm
53: (4)                 is bounded by the worst-case running time of the function
54: (4)                 ``matching_algorithm``.
55: (4)                 Minimum edge cover for `G` can also be found using the `min_edge_covering`
56: (4)                 function in :mod:`networkx.algorithms.bipartite.covering` which is
57: (4)                 simply this function with a default matching algorithm of
58: (4)                 :func:`~networkx.algorithms.bipartite.matching.hopcraft_karp_matching`
59: (4)                 """
60: (4)                 if len(G) == 0:
61: (8)                     return set()
62: (4)                 if nx.number_of_isolates(G) > 0:
63: (8)                     raise nx.NetworkXException(
64: (12)                        "Graph has a node with no edge incident on it, so no edge cover
exists."
65: (8)                     )
66: (4)                 if matching_algorithm is None:
67: (8)                     matching_algorithm = partial(nx.max_weight_matching,
maxcardinality=True)
68: (4)                 maximum_matching = matching_algorithm(G)
69: (4)                 try:
70: (8)                     min_cover = set(maximum_matching.items())
71: (8)                     bipartite_cover = True
72: (4)                 except AttributeError:
73: (8)                     min_cover = maximum_matching
74: (8)                     bipartite_cover = False
```

```
75: (4)                 uncovered_nodes = set(G) - {v for u, v in min_cover} - {u for u, v in
min_cover}
76: (4)                 for v in uncovered_nodes:
77: (8)                     u = arbitrary_element(G[v])
78: (8)                     min_cover.add((u, v))
79: (8)                     if bipartite_cover:
80: (12)                        min_cover.add((v, u))
81: (4)                 return min_cover
82: (0)             @not_implemented_for("directed")
83: (0)             @nx._dispatchable
84: (0)             def is_edge_cover(G, cover):
85: (4)                 """Decides whether a set of edges is a valid edge cover of the graph.
86: (4)                 Given a set of edges, whether it is an edge covering can
87: (4)                 be decided if we just check whether all nodes of the graph
88: (4)                 has an edge from the set, incident on it.
89: (4)                 Parameters
90: (4)                 ----------
91: (4)                 G : NetworkX graph
92: (8)                     An undirected bipartite graph.
93: (4)                 cover : set
94: (8)                     Set of edges to be checked.
95: (4)                 Returns
96: (4)                 -------
97: (4)                 bool
98: (8)                     Whether the set of edges is a valid edge cover of the graph.
99: (4)                 Examples
100: (4)                --------
101: (4)                >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
102: (4)                >>> cover = {(2, 1), (3, 0)}
103: (4)                >>> nx.is_edge_cover(G, cover)
104: (4)                True
105: (4)                Notes
106: (4)                -----
107: (4)                An edge cover of a graph is a set of edges such that every node of
108: (4)                the graph is incident to at least one edge of the set.
109: (4)                """
110: (4)                return set(G) <= set(chain.from_iterable(cover))
```

File 20 - cuts.py:

```
1: (0)              """Functions for finding and evaluating cuts in a graph.
2: (0)              """
3: (0)              from itertools import chain
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "boundary_expansion",
7: (4)                  "conductance",
8: (4)                  "cut_size",
9: (4)                  "edge_expansion",
10: (4)                 "mixing_expansion",
11: (4)                 "node_expansion",
12: (4)                 "normalized_cut_size",
13: (4)                 "volume",
14: (0)             ]
15: (0)             @nx._dispatchable(edge_attrs="weight")
16: (0)             def cut_size(G, S, T=None, weight=None):
17: (4)                 """Returns the size of the cut between two sets of nodes.
18: (4)                 A *cut* is a partition of the nodes of a graph into two sets. The
19: (4)                 *cut size* is the sum of the weights of the edges "between" the two
20: (4)                 sets of nodes.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : NetworkX graph
24: (4)                 S : collection
25: (8)                     A collection of nodes in `G`.
26: (4)                 T : collection
27: (8)                     A collection of nodes in `G`. If not specified, this is taken to
```

```
 28: (8)                    be the set complement of `S`.
 29: (4)                weight : object
 30: (8)                    Edge attribute key to use as weight. If not specified, edges
 31: (8)                    have weight one.
 32: (4)                Returns
 33: (4)                -------
 34: (4)                number
 35: (8)                    Total weight of all edges from nodes in set `S` to nodes in
 36: (8)                    set `T` (and, in the case of directed graphs, all edges from
 37: (8)                    nodes in `T` to nodes in `S`).
 38: (4)                Examples
 39: (4)                --------
 40: (4)                In the graph with two cliques joined by a single edges, the natural
 41: (4)                bipartition of the graph into two blocks, one for each clique,
 42: (4)                yields a cut of weight one::
 43: (8)                    >>> G = nx.barbell_graph(3, 0)
 44: (8)                    >>> S = {0, 1, 2}
 45: (8)                    >>> T = {3, 4, 5}
 46: (8)                    >>> nx.cut_size(G, S, T)
 47: (8)                    1
 48: (4)                Each parallel edge in a multigraph is counted when determining the
 49: (4)                cut size::
 50: (8)                    >>> G = nx.MultiGraph(["ab", "ab"])
 51: (8)                    >>> S = {"a"}
 52: (8)                    >>> T = {"b"}
 53: (8)                    >>> nx.cut_size(G, S, T)
 54: (8)                    2
 55: (4)                Notes
 56: (4)                -----
 57: (4)                In a multigraph, the cut size is the total weight of edges including
 58: (4)                multiplicity.
 59: (4)                """
 60: (4)                edges = nx.edge_boundary(G, S, T, data=weight, default=1)
 61: (4)                if G.is_directed():
 62: (8)                    edges = chain(edges, nx.edge_boundary(G, T, S, data=weight,
default=1))
 63: (4)                return sum(weight for u, v, weight in edges)
 64: (0)            @nx._dispatchable(edge_attrs="weight")
 65: (0)            def volume(G, S, weight=None):
 66: (4)                """Returns the volume of a set of nodes.
 67: (4)                The *volume* of a set *S* is the sum of the (out-)degrees of nodes
 68: (4)                in *S* (taking into account parallel edges in multigraphs). [1]
 69: (4)                Parameters
 70: (4)                ----------
 71: (4)                G : NetworkX graph
 72: (4)                S : collection
 73: (8)                    A collection of nodes in `G`.
 74: (4)                weight : object
 75: (8)                    Edge attribute key to use as weight. If not specified, edges
 76: (8)                    have weight one.
 77: (4)                Returns
 78: (4)                -------
 79: (4)                number
 80: (8)                    The volume of the set of nodes represented by `S` in the graph
 81: (8)                    `G`.
 82: (4)                See also
 83: (4)                --------
 84: (4)                conductance
 85: (4)                cut_size
 86: (4)                edge_expansion
 87: (4)                edge_boundary
 88: (4)                normalized_cut_size
 89: (4)                References
 90: (4)                ----------
 91: (4)                .. [1] David Gleich.
 92: (11)                      *Hierarchical Directed Spectral Graph Partitioning*.
 93: (11)
<https://www.cs.purdue.edu/homes/dgleich/publications/Gleich%202005%20-
%20hierarchical%20directed%20spectral.pdf>
```

```
94: (4)                 """
95: (4)                 degree = G.out_degree if G.is_directed() else G.degree
96: (4)                 return sum(d for v, d in degree(S, weight=weight))
97: (0)             @nx._dispatchable(edge_attrs="weight")
98: (0)             def normalized_cut_size(G, S, T=None, weight=None):
99: (4)                 """Returns the normalized size of the cut between two sets of nodes.
100: (4)                The *normalized cut size* is the cut size times the sum of the
101: (4)                reciprocal sizes of the volumes of the two sets. [1]
102: (4)                Parameters
103: (4)                ----------
104: (4)                G : NetworkX graph
105: (4)                S : collection
106: (8)                    A collection of nodes in `G`.
107: (4)                T : collection
108: (8)                    A collection of nodes in `G`.
109: (4)                weight : object
110: (8)                    Edge attribute key to use as weight. If not specified, edges
111: (8)                    have weight one.
112: (4)                Returns
113: (4)                -------
114: (4)                number
115: (8)                    The normalized cut size between the two sets `S` and `T`.
116: (4)                Notes
117: (4)                -----
118: (4)                In a multigraph, the cut size is the total weight of edges including
119: (4)                multiplicity.
120: (4)                See also
121: (4)                --------
122: (4)                conductance
123: (4)                cut_size
124: (4)                edge_expansion
125: (4)                volume
126: (4)                References
127: (4)                ----------
128: (4)                .. [1] David Gleich.
129: (11)                      *Hierarchical Directed Spectral Graph Partitioning*.
130: (11)
<https://www.cs.purdue.edu/homes/dgleich/publications/Gleich%202005%20-
%20hierarchical%20directed%20spectral.pdf>
131: (4)                """
132: (4)                if T is None:
133: (8)                    T = set(G) - set(S)
134: (4)                num_cut_edges = cut_size(G, S, T=T, weight=weight)
135: (4)                volume_S = volume(G, S, weight=weight)
136: (4)                volume_T = volume(G, T, weight=weight)
137: (4)                return num_cut_edges * ((1 / volume_S) + (1 / volume_T))
138: (0)            @nx._dispatchable(edge_attrs="weight")
139: (0)            def conductance(G, S, T=None, weight=None):
140: (4)                """Returns the conductance of two sets of nodes.
141: (4)                The *conductance* is the quotient of the cut size and the smaller of
142: (4)                the volumes of the two sets. [1]
143: (4)                Parameters
144: (4)                ----------
145: (4)                G : NetworkX graph
146: (4)                S : collection
147: (8)                    A collection of nodes in `G`.
148: (4)                T : collection
149: (8)                    A collection of nodes in `G`.
150: (4)                weight : object
151: (8)                    Edge attribute key to use as weight. If not specified, edges
152: (8)                    have weight one.
153: (4)                Returns
154: (4)                -------
155: (4)                number
156: (8)                    The conductance between the two sets `S` and `T`.
157: (4)                See also
158: (4)                --------
159: (4)                cut_size
160: (4)                edge_expansion
```

```
(4)                 normalized_cut_size
(4)                 volume
(4)                 References
(4)                 ----------
(4)                 .. [1] David Gleich.
(11)                       *Hierarchical Directed Spectral Graph Partitioning*.
(11)
<https://www.cs.purdue.edu/homes/dgleich/publications/Gleich%202005%20-
%20hierarchical%20directed%20spectral.pdf>
(4)                 """
(4)                 if T is None:
(8)                     T = set(G) - set(S)
(4)                 num_cut_edges = cut_size(G, S, T, weight=weight)
(4)                 volume_S = volume(G, S, weight=weight)
(4)                 volume_T = volume(G, T, weight=weight)
(4)                 return num_cut_edges / min(volume_S, volume_T)
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def edge_expansion(G, S, T=None, weight=None):
(4)                 """Returns the edge expansion between two node sets.
(4)                 The *edge expansion* is the quotient of the cut size and the smaller
(4)                 of the cardinalities of the two sets. [1]
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 S : collection
(8)                     A collection of nodes in `G`.
(4)                 T : collection
(8)                     A collection of nodes in `G`.
(4)                 weight : object
(8)                     Edge attribute key to use as weight. If not specified, edges
(8)                     have weight one.
(4)                 Returns
(4)                 -------
(4)                 number
(8)                     The edge expansion between the two sets `S` and `T`.
(4)                 See also
(4)                 --------
(4)                 boundary_expansion
(4)                 mixing_expansion
(4)                 node_expansion
(4)                 References
(4)                 ----------
(4)                 .. [1] Fan Chung.
(11)                       *Spectral Graph Theory*.
(11)                       (CBMS Regional Conference Series in Mathematics, No. 92),
(11)                       American Mathematical Society, 1997, ISBN 0-8218-0315-8
(11)                       <http://www.math.ucsd.edu/~fan/research/revised.html>
(4)                 """
(4)                 if T is None:
(8)                     T = set(G) - set(S)
(4)                 num_cut_edges = cut_size(G, S, T=T, weight=weight)
(4)                 return num_cut_edges / min(len(S), len(T))
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def mixing_expansion(G, S, T=None, weight=None):
(4)                 """Returns the mixing expansion between two node sets.
(4)                 The *mixing expansion* is the quotient of the cut size and twice the
(4)                 number of edges in the graph. [1]
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 S : collection
(8)                     A collection of nodes in `G`.
(4)                 T : collection
(8)                     A collection of nodes in `G`.
(4)                 weight : object
(8)                     Edge attribute key to use as weight. If not specified, edges
(8)                     have weight one.
(4)                 Returns
(4)                 -------
```

```
228: (4)                number
229: (8)                    The mixing expansion between the two sets `S` and `T`.
230: (4)                See also
231: (4)                --------
232: (4)                boundary_expansion
233: (4)                edge_expansion
234: (4)                node_expansion
235: (4)                References
236: (4)                ----------
237: (4)                .. [1] Vadhan, Salil P.
238: (11)                      "Pseudorandomness."
239: (11)                      *Foundations and Trends
240: (11)                      in Theoretical Computer Science* 7.1–3 (2011): 1–336.
241: (11)                      <https://doi.org/10.1561/0400000010>
242: (4)                """
243: (4)                num_cut_edges = cut_size(G, S, T=T, weight=weight)
244: (4)                num_total_edges = G.number_of_edges()
245: (4)                return num_cut_edges / (2 * num_total_edges)
246: (0)            @nx._dispatchable
247: (0)            def node_expansion(G, S):
248: (4)                """Returns the node expansion of the set `S`.
249: (4)                The *node expansion* is the quotient of the size of the node
250: (4)                boundary of *S* and the cardinality of *S*. [1]
251: (4)                Parameters
252: (4)                ----------
253: (4)                G : NetworkX graph
254: (4)                S : collection
255: (8)                    A collection of nodes in `G`.
256: (4)                Returns
257: (4)                -------
258: (4)                number
259: (8)                    The node expansion of the set `S`.
260: (4)                See also
261: (4)                --------
262: (4)                boundary_expansion
263: (4)                edge_expansion
264: (4)                mixing_expansion
265: (4)                References
266: (4)                ----------
267: (4)                .. [1] Vadhan, Salil P.
268: (11)                      "Pseudorandomness."
269: (11)                      *Foundations and Trends
270: (11)                      in Theoretical Computer Science* 7.1–3 (2011): 1–336.
271: (11)                      <https://doi.org/10.1561/0400000010>
272: (4)                """
273: (4)                neighborhood = set(chain.from_iterable(G.neighbors(v) for v in S))
274: (4)                return len(neighborhood) / len(S)
275: (0)            @nx._dispatchable
276: (0)            def boundary_expansion(G, S):
277: (4)                """Returns the boundary expansion of the set `S`.
278: (4)                The *boundary expansion* is the quotient of the size
279: (4)                of the node boundary and the cardinality of *S*. [1]
280: (4)                Parameters
281: (4)                ----------
282: (4)                G : NetworkX graph
283: (4)                S : collection
284: (8)                    A collection of nodes in `G`.
285: (4)                Returns
286: (4)                -------
287: (4)                number
288: (8)                    The boundary expansion of the set `S`.
289: (4)                See also
290: (4)                --------
291: (4)                edge_expansion
292: (4)                mixing_expansion
293: (4)                node_expansion
294: (4)                References
295: (4)                ----------
296: (4)                .. [1] Vadhan, Salil P.
```

```
297: (11)                       "Pseudorandomness."
298: (11)                       *Foundations and Trends in Theoretical Computer Science*
299: (11)                       7.1–3 (2011): 1–336.
300: (11)                       <https://doi.org/10.1561/0400000010>
301: (4)                 """
302: (4)                 return len(nx.node_boundary(G, S)) / len(S)
```

File 21 - cycles.py:

```
1: (0)               """
2: (0)               ========================
3: (0)               Cycle finding algorithms
4: (0)               ========================
5: (0)               """
6: (0)               from collections import Counter, defaultdict
7: (0)               from itertools import combinations, product
8: (0)               from math import inf
9: (0)               import networkx as nx
10: (0)              from networkx.utils import not_implemented_for, pairwise
11: (0)              __all__ = [
12: (4)                  "cycle_basis",
13: (4)                  "simple_cycles",
14: (4)                  "recursive_simple_cycles",
15: (4)                  "find_cycle",
16: (4)                  "minimum_cycle_basis",
17: (4)                  "chordless_cycles",
18: (4)                  "girth",
19: (0)              ]
20: (0)              @not_implemented_for("directed")
21: (0)              @not_implemented_for("multigraph")
22: (0)              @nx._dispatchable
23: (0)              def cycle_basis(G, root=None):
24: (4)                  """Returns a list of cycles which form a basis for cycles of G.
25: (4)                  A basis for cycles of a network is a minimal collection of
26: (4)                  cycles such that any cycle in the network can be written
27: (4)                  as a sum of cycles in the basis.  Here summation of cycles
28: (4)                  is defined as "exclusive or" of the edges. Cycle bases are
29: (4)                  useful, e.g. when deriving equations for electric circuits
30: (4)                  using Kirchhoff's Laws.
31: (4)                  Parameters
32: (4)                  ----------
33: (4)                  G : NetworkX Graph
34: (4)                  root : node, optional
35: (7)                     Specify starting node for basis.
36: (4)                  Returns
37: (4)                  -------
38: (4)                  A list of cycle lists.  Each cycle list is a list of nodes
39: (4)                  which forms a cycle (loop) in G.
40: (4)                  Examples
41: (4)                  --------
42: (4)                  >>> G = nx.Graph()
43: (4)                  >>> nx.add_cycle(G, [0, 1, 2, 3])
44: (4)                  >>> nx.add_cycle(G, [0, 3, 4, 5])
45: (4)                  >>> nx.cycle_basis(G, 0)
46: (4)                  [[3, 4, 5, 0], [1, 2, 3, 0]]
47: (4)                  Notes
48: (4)                  -----
49: (4)                  This is adapted from algorithm CACM 491 [1]_.
50: (4)                  References
51: (4)                  ----------
52: (4)                  .. [1] Paton, K. An algorithm for finding a fundamental set of
53: (7)                     cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.
54: (4)                  See Also
55: (4)                  --------
56: (4)                  simple_cycles
57: (4)                  minimum_cycle_basis
58: (4)                  """
```

```
59: (4)                 gnodes = dict.fromkeys(G)  # set-like object that maintains node order
60: (4)                 cycles = []
61: (4)                 while gnodes:  # loop over connected components
62: (8)                     if root is None:
63: (12)                        root = gnodes.popitem()[0]
64: (8)                     stack = [root]
65: (8)                     pred = {root: root}
66: (8)                     used = {root: set()}
67: (8)                     while stack:  # walk the spanning tree finding cycles
68: (12)                        z = stack.pop()  # use last-in so cycles easier to find
69: (12)                        zused = used[z]
70: (12)                        for nbr in G[z]:
71: (16)                            if nbr not in used:  # new node
72: (20)                                pred[nbr] = z
73: (20)                                stack.append(nbr)
74: (20)                                used[nbr] = {z}
75: (16)                            elif nbr == z:  # self loops
76: (20)                                cycles.append([z])
77: (16)                            elif nbr not in zused:  # found a cycle
78: (20)                                pn = used[nbr]
79: (20)                                cycle = [nbr, z]
80: (20)                                p = pred[z]
81: (20)                                while p not in pn:
82: (24)                                    cycle.append(p)
83: (24)                                    p = pred[p]
84: (20)                                cycle.append(p)
85: (20)                                cycles.append(cycle)
86: (20)                                used[nbr].add(z)
87: (8)                     for node in pred:
88: (12)                        gnodes.pop(node, None)
89: (8)                     root = None
90: (4)                 return cycles
91: (0)             @nx._dispatchable
92: (0)             def simple_cycles(G, length_bound=None):
93: (4)                 """Find simple cycles (elementary circuits) of a graph.
94: (4)                 A `simple cycle`, or `elementary circuit`, is a closed path where
95: (4)                 no node appears twice.  In a directed graph, two simple cycles are
distinct
96: (4)                 if they are not cyclic permutations of each other.  In an undirected
graph,
97: (4)                 two simple cycles are distinct if they are not cyclic permutations of each
98: (4)                 other nor of the other's reversal.
99: (4)                 Optionally, the cycles are bounded in length.  In the unbounded case, we
use
100: (4)                a nonrecursive, iterator/generator version of Johnson's algorithm [1]_.
In
101: (4)                the bounded case, we use a version of the algorithm of Gupta and
102: (4)                Suzumura[2]_. There may be better algorithms for some cases [3]_ [4]_
[5]_.
103: (4)                The algorithms of Johnson, and Gupta and Suzumura, are enhanced by some
104: (4)                well-known preprocessing techniques.  When G is directed, we restrict our
105: (4)                attention to strongly connected components of G, generate all simple
cycles
106: (4)                containing a certain node, remove that node, and further decompose the
107: (4)                remainder into strongly connected components.  When G is undirected, we
108: (4)                restrict our attention to biconnected components, generate all simple
cycles
109: (4)                containing a particular edge, remove that edge, and further decompose the
110: (4)                remainder into biconnected components.
111: (4)                Note that multigraphs are supported by this function -- and in undirected
112: (4)                multigraphs, a pair of parallel edges is considered a cycle of length 2.
113: (4)                Likewise, self-loops are considered to be cycles of length 1.  We define
114: (4)                cycles as sequences of nodes; so the presence of loops and parallel edges
115: (4)                does not change the number of simple cycles in a graph.
116: (4)                Parameters
117: (4)                ----------
118: (4)                G : NetworkX DiGraph
119: (7)                   A directed graph
120: (4)                length_bound : int or None, optional (default=None)
```

```
   If length_bound is an int, generate all simple cycles of G with length at
   most length_bound.  Otherwise, generate all simple cycles of G.
Yields
------
list of nodes
   Each cycle is represented by a list of nodes along the cycle.
Examples
--------
>>> edges = [(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> G = nx.DiGraph(edges)
>>> sorted(nx.simple_cycles(G))
[[0], [0, 1, 2], [0, 2], [1, 2], [2]]
To filter the cycles so that they don't include certain nodes or edges,
copy your graph and eliminate those nodes or edges before calling.
For example, to exclude self-loops from the above example:
>>> H = G.copy()
>>> H.remove_edges_from(nx.selfloop_edges(G))
>>> sorted(nx.simple_cycles(H))
[[0, 1, 2], [0, 2], [1, 2]]
Notes
-----
When length_bound is None, the time complexity is $O((n+e)(c+1))$ for $n$
nodes, $e$ edges and $c$ simple circuits.  Otherwise, when length_bound > 1,
the time complexity is $O((c+n)(k-1)d^k)$ where $d$ is the average degree of
the nodes of G and $k$ = length_bound.
Raises
------
ValueError
    when length_bound < 0.
References
----------
.. [1] Finding all the elementary circuits of a directed graph.
   D. B. Johnson, SIAM Journal on Computing 4, no. 1, 77-84, 1975.
   https://doi.org/10.1137/0204007
.. [2] Finding All Bounded-Length Simple Cycles in a Directed Graph
   A. Gupta and T. Suzumura https://arxiv.org/abs/2105.10094
.. [3] Enumerating the cycles of a digraph: a new preprocessing strategy.
   G. Loizou and P. Thanish, Information Sciences, v. 27, 163-182, 1982.
.. [4] A search strategy for the elementary cycles of a directed graph.
   J.L. Szwarcfiter and P.E. Lauer, BIT NUMERICAL MATHEMATICS,
   v. 16, no. 2, 192-204, 1976.
.. [5] Optimal Listing of Cycles and st-Paths in Undirected Graphs
    R. Ferreira and R. Grossi and A. Marino and N. Pisanti and R. Rizzi and
    G. Sacomoto https://arxiv.org/abs/1205.2766
See Also
--------
cycle_basis
chordless_cycles
"""
if length_bound is not None:
    if length_bound == 0:
        return
    elif length_bound < 0:
        raise ValueError("length bound must be non-negative")
directed = G.is_directed()
yield from ([v] for v, Gv in G.adj.items() if v in Gv)
if length_bound is not None and length_bound == 1:
    return
if G.is_multigraph() and not directed:
    visited = set()
    for u, Gu in G.adj.items():
        multiplicity = ((v, len(Guv)) for v, Guv in Gu.items() if v in visited)
        yield from ([u, v] for v, m in multiplicity if m > 1)
        visited.add(u)
```

```
    if directed:
        G = nx.DiGraph((u, v) for u, Gu in G.adj.items() for v in Gu if v !=
u)
    else:
        G = nx.Graph((u, v) for u, Gu in G.adj.items() for v in Gu if v != u)
    if length_bound is not None and length_bound == 2:
        if directed:
            visited = set()
            for u, Gu in G.adj.items():
                yield from (
                    [v, u] for v in visited.intersection(Gu) if G.has_edge(v,
u)
                )
                visited.add(u)
        return
    if directed:
        yield from _directed_cycle_search(G, length_bound)
    else:
        yield from _undirected_cycle_search(G, length_bound)
def _directed_cycle_search(G, length_bound):
    """A dispatch function for `simple_cycles` for directed graphs.
    We generate all cycles of G through binary partition.
        1. Pick a node v in G which belongs to at least one cycle
            a. Generate all cycles of G which contain the node v.
            b. Recursively generate all cycles of G \\ v.
    This is accomplished through the following:
        1. Compute the strongly connected components SCC of G.
        2. Select and remove a biconnected component C from BCC.  Select a
           non-tree edge (u, v) of a depth-first search of G[C].
        3. For each simple cycle P containing v in G[C], yield P.
        4. Add the biconnected components of G[C \\ v] to BCC.
    If the parameter length_bound is not None, then step 3 will be limited to
    simple cycles of length at most length_bound.
    Parameters
    ----------
    G : NetworkX DiGraph
       A directed graph
    length_bound : int or None
       If length_bound is an int, generate all simple cycles of G with length
at most length_bound.
       Otherwise, generate all simple cycles of G.
    Yields
    ------
    list of nodes
       Each cycle is represented by a list of nodes along the cycle.
    """
    scc = nx.strongly_connected_components
    components = [c for c in scc(G) if len(c) >= 2]
    while components:
        c = components.pop()
        Gc = G.subgraph(c)
        v = next(iter(c))
        if length_bound is None:
            yield from _johnson_cycle_search(Gc, [v])
        else:
            yield from _bounded_cycle_search(Gc, [v], length_bound)
        G.remove_node(v)
        components.extend(c for c in scc(Gc) if len(c) >= 2)
def _undirected_cycle_search(G, length_bound):
    """A dispatch function for `simple_cycles` for undirected graphs.
    We generate all cycles of G through binary partition.
        1. Pick an edge (u, v) in G which belongs to at least one cycle
            a. Generate all cycles of G which contain the edge (u, v)
            b. Recursively generate all cycles of G \\ (u, v)
    This is accomplished through the following:
        1. Compute the biconnected components BCC of G.
        2. Select and remove a biconnected component C from BCC.  Select a
           non-tree edge (u, v) of a depth-first search of G[C].
        3. For each (v -> u) path P remaining in G[C] \\ (u, v), yield P.
```

```
(8)                 4. Add the biconnected components of G[C] \\ (u, v) to BCC.
(4)             If the parameter length_bound is not None, then step 3 will be limited to simple paths
(4)             of length at most length_bound.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(7)                An undirected graph
(4)             length_bound : int or None
(7)                If length_bound is an int, generate all simple cycles of G with length at most length_bound.
(7)                Otherwise, generate all simple cycles of G.
(4)             Yields
(4)             ------
(4)             list of nodes
(7)                Each cycle is represented by a list of nodes along the cycle.
(4)             """
(4)             bcc = nx.biconnected_components
(4)             components = [c for c in bcc(G) if len(c) >= 3]
(4)             while components:
(8)                 c = components.pop()
(8)                 Gc = G.subgraph(c)
(8)                 uv = list(next(iter(Gc.edges)))
(8)                 G.remove_edge(*uv)
(8)                 if length_bound is None:
(12)                    yield from _johnson_cycle_search(Gc, uv)
(8)                 else:
(12)                    yield from _bounded_cycle_search(Gc, uv, length_bound)
(8)                 components.extend(c for c in bcc(Gc) if len(c) >= 3)
(0)         class _NeighborhoodCache(dict):
(4)             """Very lightweight graph wrapper which caches neighborhoods as list.
(4)             This dict subclass uses the __missing__ functionality to query graphs for
(4)             their neighborhoods, and store the result as a list.  This is used to avoid
(4)             the performance penalty incurred by subgraph views.
(4)             """
(4)             def __init__(self, G):
(8)                 self.G = G
(4)             def __missing__(self, v):
(8)                 Gv = self[v] = list(self.G[v])
(8)                 return Gv
(0)         def _johnson_cycle_search(G, path):
(4)             """The main loop of the cycle-enumeration algorithm of Johnson.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph or DiGraph
(7)                A graph
(4)             path : list
(7)                A cycle prefix.  All cycles generated will begin with this prefix.
(4)             Yields
(4)             ------
(4)             list of nodes
(7)                Each cycle is represented by a list of nodes along the cycle.
(4)             References
(4)             ----------
(8)                 .. [1] Finding all the elementary circuits of a directed graph.
(7)                D. B. Johnson, SIAM Journal on Computing 4, no. 1, 77-84, 1975.
(7)                https://doi.org/10.1137/0204007
(4)             """
(4)             G = _NeighborhoodCache(G)
(4)             blocked = set(path)
(4)             B = defaultdict(set)  # graph portions that yield no elementary circuit
(4)             start = path[0]
(4)             stack = [iter(G[path[-1]])]
(4)             closed = [False]
(4)             while stack:
(8)                 nbrs = stack[-1]
(8)                 for w in nbrs:
(12)                    if w == start:
```

```
(16)                            yield path[:]
(16)                            closed[-1] = True
(12)                        elif w not in blocked:
(16)                            path.append(w)
(16)                            closed.append(False)
(16)                            stack.append(iter(G[w]))
(16)                            blocked.add(w)
(16)                            break
(8)                     else:  # no more nbrs
(12)                        stack.pop()
(12)                        v = path.pop()
(12)                        if closed.pop():
(16)                            if closed:
(20)                                closed[-1] = True
(16)                            unblock_stack = {v}
(16)                            while unblock_stack:
(20)                                u = unblock_stack.pop()
(20)                                if u in blocked:
(24)                                    blocked.remove(u)
(24)                                    unblock_stack.update(B[u])
(24)                                    B[u].clear()
(12)                        else:
(16)                            for w in G[v]:
(20)                                B[w].add(v)
(0)             def _bounded_cycle_search(G, path, length_bound):
(4)                 """The main loop of the cycle-enumeration algorithm of Gupta and Suzumura.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph or DiGraph
(7)                    A graph
(4)                 path : list
(7)                    A cycle prefix.  All cycles generated will begin with this prefix.
(4)                 length_bound: int
(8)                     A length bound.  All cycles generated will have length at most
length_bound.
(4)                 Yields
(4)                 ------
(4)                 list of nodes
(7)                    Each cycle is represented by a list of nodes along the cycle.
(4)                 References
(4)                 ----------
(4)                 .. [1] Finding All Bounded-Length Simple Cycles in a Directed Graph
(7)                    A. Gupta and T. Suzumura https://arxiv.org/abs/2105.10094
(4)                 """
(4)                 G = _NeighborhoodCache(G)
(4)                 lock = {v: 0 for v in path}
(4)                 B = defaultdict(set)
(4)                 start = path[0]
(4)                 stack = [iter(G[path[-1]])]
(4)                 blen = [length_bound]
(4)                 while stack:
(8)                     nbrs = stack[-1]
(8)                     for w in nbrs:
(12)                        if w == start:
(16)                            yield path[:]
(16)                            blen[-1] = 1
(12)                        elif len(path) < lock.get(w, length_bound):
(16)                            path.append(w)
(16)                            blen.append(length_bound)
(16)                            lock[w] = len(path)
(16)                            stack.append(iter(G[w]))
(16)                            break
(8)                     else:
(12)                        stack.pop()
(12)                        v = path.pop()
(12)                        bl = blen.pop()
(12)                        if blen:
(16)                            blen[-1] = min(blen[-1], bl)
(12)                        if bl < length_bound:
```

```
385: (16)                            relax_stack = [(bl, v)]
386: (16)                            while relax_stack:
387: (20)                                bl, u = relax_stack.pop()
388: (20)                                if lock.get(u, length_bound) < length_bound - bl + 1:
389: (24)                                    lock[u] = length_bound - bl + 1
390: (24)                                    relax_stack.extend((bl + 1, w) for w in
B[u].difference(path))
391: (12)                        else:
392: (16)                            for w in G[v]:
393: (20)                                B[w].add(v)
394: (0)            @nx._dispatchable
395: (0)            def chordless_cycles(G, length_bound=None):
396: (4)                """Find simple chordless cycles of a graph.
397: (4)                A `simple cycle` is a closed path where no node appears twice.  In a
simple
398: (4)                cycle, a `chord` is an additional edge between two nodes in the cycle.  A
399: (4)                `chordless cycle` is a simple cycle without chords.  Said differently, a
400: (4)                chordless cycle is a cycle C in a graph G where the number of edges in the
401: (4)                induced graph G[C] is equal to the length of `C`.
402: (4)                Note that some care must be taken in the case that G is not a simple graph
403: (4)                nor a simple digraph.  Some authors limit the definition of chordless
cycles
404: (4)                to have a prescribed minimum length; we do not.
405: (8)                    1. We interpret self-loops to be chordless cycles, except in
multigraphs
406: (11)                      with multiple loops in parallel.  Likewise, in a chordless cycle of
407: (11)                      length greater than 1, there can be no nodes with self-loops.
408: (8)                    2. We interpret directed two-cycles to be chordless cycles, except in
409: (11)                      multi-digraphs when any edge in a two-cycle has a parallel copy.
410: (8)                    3. We interpret parallel pairs of undirected edges as two-cycles,
except
411: (11)                      when a third (or more) parallel edge exists between the two nodes.
412: (8)                    4. Generalizing the above, edges with parallel clones may not occur in
413: (11)                      chordless cycles.
414: (4)                In a directed graph, two chordless cycles are distinct if they are not
415: (4)                cyclic permutations of each other.  In an undirected graph, two chordless
416: (4)                cycles are distinct if they are not cyclic permutations of each other nor
of
417: (4)                the other's reversal.
418: (4)                Optionally, the cycles are bounded in length.
419: (4)                We use an algorithm strongly inspired by that of Dias et al [1]_.  It has
420: (4)                been modified in the following ways:
421: (8)                    1. Recursion is avoided, per Python's limitations
422: (8)                    2. The labeling function is not necessary, because the starting paths
423: (12)                       are chosen (and deleted from the host graph) to prevent multiple
424: (12)                       occurrences of the same path
425: (8)                    3. The search is optionally bounded at a specified length
426: (8)                    4. Support for directed graphs is provided by extending cycles along
427: (12)                       forward edges, and blocking nodes along forward and reverse edges
428: (8)                    5. Support for multigraphs is provided by omitting digons from the set
429: (12)                       of forward edges
430: (4)                Parameters
431: (4)                ----------
432: (4)                G : NetworkX DiGraph
433: (7)                   A directed graph
434: (4)                length_bound : int or None, optional (default=None)
435: (7)                   If length_bound is an int, generate all simple cycles of G with length
at
436: (7)                   most length_bound.  Otherwise, generate all simple cycles of G.
437: (4)                Yields
438: (4)                ------
439: (4)                list of nodes
440: (7)                   Each cycle is represented by a list of nodes along the cycle.
441: (4)                Examples
442: (4)                --------
443: (4)                >>> sorted(list(nx.chordless_cycles(nx.complete_graph(4))))
444: (4)                [[1, 0, 2], [1, 0, 3], [2, 0, 3], [2, 1, 3]]
445: (4)                Notes
446: (4)                -----
```

```
(4)                 When length_bound is None, and the graph is simple, the time complexity is
(4)                 $O((n+e)(c+1))$ for $n$ nodes, $e$ edges and $c$ chordless cycles.
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     when length_bound < 0.
(4)                 References
(4)                 ----------
(4)                 .. [1] Efficient enumeration of chordless cycles
(7)                    E. Dias and D. Castonguay and H. Longo and W.A.R. Jradi
(7)                    https://arxiv.org/abs/1309.1051
(4)                 See Also
(4)                 --------
(4)                 simple_cycles
(4)                 """
(4)                 if length_bound is not None:
(8)                     if length_bound == 0:
(12)                        return
(8)                     elif length_bound < 0:
(12)                        raise ValueError("length bound must be non-negative")
(4)                 directed = G.is_directed()
(4)                 multigraph = G.is_multigraph()
(4)                 if multigraph:
(8)                     yield from ([v] for v, Gv in G.adj.items() if len(Gv.get(v, ())) == 1)
(4)                 else:
(8)                     yield from ([v] for v, Gv in G.adj.items() if v in Gv)
(4)                 if length_bound is not None and length_bound == 1:
(8)                     return
(4)                 if directed:
(8)                     F = nx.DiGraph((u, v) for u, Gu in G.adj.items() if u not in Gu for v
in Gu)
(8)                     B = F.to_undirected(as_view=False)
(4)                 else:
(8)                     F = nx.Graph((u, v) for u, Gu in G.adj.items() if u not in Gu for v in
Gu)
(8)                     B = None
(4)                 if multigraph:
(8)                     if not directed:
(12)                        B = F.copy()
(12)                        visited = set()
(8)                     for u, Gu in G.adj.items():
(12)                        if directed:
(16)                            multiplicity = ((v, len(Guv)) for v, Guv in Gu.items())
(16)                            for v, m in multiplicity:
(20)                                if m > 1:
(24)                                    F.remove_edges_from(((u, v), (v, u)))
(12)                        else:
(16)                            multiplicity = ((v, len(Guv)) for v, Guv in Gu.items() if v in
visited)
(16)                            for v, m in multiplicity:
(20)                                if m == 2:
(24)                                    yield [u, v]
(20)                                if m > 1:
(24)                                    F.remove_edge(u, v)
(16)                            visited.add(u)
(4)                 if directed:
(8)                     for u, Fu in F.adj.items():
(12)                        digons = [[u, v] for v in Fu if F.has_edge(v, u)]
(12)                        yield from digons
(12)                        F.remove_edges_from(digons)
(12)                        F.remove_edges_from(e[::-1] for e in digons)
(4)                 if length_bound is not None and length_bound == 2:
(8)                     return
(4)                 if directed:
(8)                     separate = nx.strongly_connected_components
(8)                     def stems(C, v):
(12)                        for u, w in product(C.pred[v], C.succ[v]):
(16)                            if not G.has_edge(u, w):  # omit stems with acyclic chords
(20)                                yield [u, v, w], F.has_edge(w, u)
```

```
(4)                 else:
(8)                     separate = nx.biconnected_components
(8)                     def stems(C, v):
(12)                        yield from (([u, v, w], F.has_edge(w, u)) for u, w in
combinations(C[v], 2))
(4)                 components = [c for c in separate(F) if len(c) > 2]
(4)                 while components:
(8)                     c = components.pop()
(8)                     v = next(iter(c))
(8)                     Fc = F.subgraph(c)
(8)                     Fcc = Bcc = None
(8)                     for S, is_triangle in stems(Fc, v):
(12)                        if is_triangle:
(16)                            yield S
(12)                        else:
(16)                            if Fcc is None:
(20)                                Fcc = _NeighborhoodCache(Fc)
(20)                                Bcc = Fcc if B is None else
_NeighborhoodCache(B.subgraph(c))
(16)                            yield from _chordless_cycle_search(Fcc, Bcc, S, length_bound)
(8)                     components.extend(c for c in separate(F.subgraph(c - {v})) if len(c) >
2)
(0)             def _chordless_cycle_search(F, B, path, length_bound):
(4)                 """The main loop for chordless cycle enumeration.
(4)                 This algorithm is strongly inspired by that of Dias et al [1]_.  It has
been
(4)                 modified in the following ways:
(8)                     1. Recursion is avoided, per Python's limitations
(8)                     2. The labeling function is not necessary, because the starting paths
(12)                        are chosen (and deleted from the host graph) to prevent multiple
(12)                        occurrences of the same path
(8)                     3. The search is optionally bounded at a specified length
(8)                     4. Support for directed graphs is provided by extending cycles along
(12)                        forward edges, and blocking nodes along forward and reverse edges
(8)                     5. Support for multigraphs is provided by omitting digons from the set
(12)                        of forward edges
(4)                 Parameters
(4)                 ----------
(4)                 F : _NeighborhoodCache
(7)                    A graph of forward edges to follow in constructing cycles
(4)                 B : _NeighborhoodCache
(7)                    A graph of blocking edges to prevent the production of chordless cycles
(4)                 path : list
(7)                    A cycle prefix.  All cycles generated will begin with this prefix.
(4)                 length_bound : int
(7)                    A length bound.  All cycles generated will have length at most
length_bound.
(4)                 Yields
(4)                 ------
(4)                 list of nodes
(7)                    Each cycle is represented by a list of nodes along the cycle.
(4)                 References
(4)                 ----------
(4)                 .. [1] Efficient enumeration of chordless cycles
(7)                    E. Dias and D. Castonguay and H. Longo and W.A.R. Jradi
(7)                    https://arxiv.org/abs/1309.1051
(4)                 """
(4)                 blocked = defaultdict(int)
(4)                 target = path[0]
(4)                 blocked[path[1]] = 1
(4)                 for w in path[1:]:
(8)                     for v in B[w]:
(12)                        blocked[v] += 1
(4)                 stack = [iter(F[path[2]])]
(4)                 while stack:
(8)                     nbrs = stack[-1]
(8)                     for w in nbrs:
(12)                        if blocked[w] == 1 and (length_bound is None or len(path) <
length_bound):
```

```
(16)                            Fw = F[w]
(16)                            if target in Fw:
(20)                                yield path + [w]
(16)                            else:
(20)                                Bw = B[w]
(20)                                if target in Bw:
(24)                                    continue
(20)                                for v in Bw:
(24)                                    blocked[v] += 1
(20)                                path.append(w)
(20)                                stack.append(iter(Fw))
(20)                                break
(8)                     else:
(12)                        stack.pop()
(12)                        for v in B[path.pop()]:
(16)                            blocked[v] -= 1
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable(mutates_input=True)
(0)             def recursive_simple_cycles(G):
(4)                 """Find simple cycles (elementary circuits) of a directed graph.
(4)                 A `simple cycle`, or `elementary circuit`, is a closed path where
(4)                 no node appears twice. Two elementary circuits are distinct if they
(4)                 are not cyclic permutations of each other.
(4)                 This version uses a recursive algorithm to build a list of cycles.
(4)                 You should probably use the iterator version called simple_cycles().
(4)                 Warning: This recursive version uses lots of RAM!
(4)                 It appears in NetworkX for pedagogical value.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX DiGraph
(7)                    A directed graph
(4)                 Returns
(4)                 -------
(4)                 A list of cycles, where each cycle is represented by a list of nodes
(4)                 along the cycle.
(4)                 Example:
(4)                 >>> edges = [(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]
(4)                 >>> G = nx.DiGraph(edges)
(4)                 >>> nx.recursive_simple_cycles(G)
(4)                 [[0], [2], [0, 1, 2], [0, 2], [1, 2]]
(4)                 Notes
(4)                 -----
(4)                 The implementation follows pp. 79-80 in [1]_.
(4)                 The time complexity is $O((n+e)(c+1))$ for $n$ nodes, $e$ edges and $c$
(4)                 elementary circuits.
(4)                 References
(4)                 ----------
(4)                 .. [1] Finding all the elementary circuits of a directed graph.
(7)                    D. B. Johnson, SIAM Journal on Computing 4, no. 1, 77-84, 1975.
(7)                    https://doi.org/10.1137/0204007
(4)                 See Also
(4)                 --------
(4)                 simple_cycles, cycle_basis
(4)                 """
(4)                 def _unblock(thisnode):
(8)                     """Recursively unblock and remove nodes from B[thisnode]."""
(8)                     if blocked[thisnode]:
(12)                        blocked[thisnode] = False
(12)                        while B[thisnode]:
(16)                            _unblock(B[thisnode].pop())
(4)                 def circuit(thisnode, startnode, component):
(8)                     closed = False  # set to True if elementary path is closed
(8)                     path.append(thisnode)
(8)                     blocked[thisnode] = True
(8)                     for nextnode in component[thisnode]:  # direct successors of thisnode
(12)                        if nextnode == startnode:
(16)                            result.append(path[:])
(16)                            closed = True
(12)                        elif not blocked[nextnode]:
```

```
(16)                            if circuit(nextnode, startnode, component):
(20)                                closed = True
(8)                     if closed:
(12)                        _unblock(thisnode)
(8)                     else:
(12)                        for nextnode in component[thisnode]:
(16)                            if thisnode not in B[nextnode]:  # TODO: use set for speedup?
(20)                                B[nextnode].append(thisnode)
(8)                     path.pop()  # remove thisnode from path
(8)                     return closed
(4)                 path = []  # stack of nodes in current path
(4)                 blocked = defaultdict(bool)  # vertex: blocked from search?
(4)                 B = defaultdict(list)  # graph portions that yield no elementary circuit
(4)                 result = []  # list to accumulate the circuits found
(4)                 for v in G:
(8)                     if G.has_edge(v, v):
(12)                        result.append([v])
(12)                        G.remove_edge(v, v)
(4)                 ordering = dict(zip(G, range(len(G))))
(4)                 for s in ordering:
(8)                     subgraph = G.subgraph(node for node in G if ordering[node] >=
ordering[s])
(8)                     strongcomp = nx.strongly_connected_components(subgraph)
(8)                     mincomp = min(strongcomp, key=lambda ns: min(ordering[n] for n in ns))
(8)                     component = G.subgraph(mincomp)
(8)                     if len(component) > 1:
(12)                        startnode = min(component, key=ordering.__getitem__)
(12)                        for node in component:
(16)                            blocked[node] = False
(16)                            B[node][:] = []
(12)                        dummy = circuit(startnode, startnode, component)
(4)                 return result
(0)             @nx._dispatchable
(0)             def find_cycle(G, source=None, orientation=None):
(4)                 """Returns a cycle found via depth-first traversal.
(4)                 The cycle is a list of edges indicating the cyclic path.
(4)                 Orientation of directed edges is controlled by `orientation`.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A directed/undirected graph/multigraph.
(4)                 source : node, list of nodes
(8)                     The node from which the traversal begins. If None, then a source
(8)                     is chosen arbitrarily and repeatedly until all edges from each node in
(8)                     the graph are searched.
(4)                 orientation : None | 'original' | 'reverse' | 'ignore' (default: None)
(8)                     For directed graphs and directed multigraphs, edge traversals need not
(8)                     respect the original orientation of the edges.
(8)                     When set to 'reverse' every edge is traversed in the reverse
direction.
(8)                     When set to 'ignore', every edge is treated as undirected.
(8)                     When set to 'original', every edge is treated as directed.
(8)                     In all three cases, the yielded edge tuples add a last entry to
(8)                     indicate the direction in which that edge was traversed.
(8)                     If orientation is None, the yielded edge has no direction indicated.
(8)                     The direction is respected, but not reported.
(4)                 Returns
(4)                 -------
(4)                 edges : directed edges
(8)                     A list of directed edges indicating the path taken for the loop.
(8)                     If no cycle is found, then an exception is raised.
(8)                     For graphs, an edge is of the form `(u, v)` where `u` and `v`
(8)                     are the tail and head of the edge as determined by the traversal.
(8)                     For multigraphs, an edge is of the form `(u, v, key)`, where `key` is
(8)                     the key of the edge. When the graph is directed, then `u` and `v`
(8)                     are always in the order of the actual directed edge.
(8)                     If orientation is not None then the edge tuple is extended to include
(8)                     the direction of traversal ('forward' or 'reverse') on that edge.
(4)                 Raises
```

```
    ------
    NetworkXNoCycle
        If no cycle was found.
    Examples
    --------
    In this example, we construct a DAG and find, in the first call, that there
    are no directed cycles, and so an exception is raised. In the second call,
    we ignore edge orientations and find that there is an undirected cycle.
    Note that the second call finds a directed cycle while effectively
    traversing an undirected graph, and so, we found an "undirected cycle".
    This means that this DAG structure does not form a directed tree (which
    is also known as a polytree).
    >>> G = nx.DiGraph([(0, 1), (0, 2), (1, 2)])
    >>> nx.find_cycle(G, orientation="original")
    Traceback (most recent call last):
        ...
    networkx.exception.NetworkXNoCycle: No cycle found.
    >>> list(nx.find_cycle(G, orientation="ignore"))
    [(0, 1, 'forward'), (1, 2, 'forward'), (0, 2, 'reverse')]
    See Also
    --------
    simple_cycles
    """
    if not G.is_directed() or orientation in (None, "original"):
        def tailhead(edge):
            return edge[:2]
    elif orientation == "reverse":
        def tailhead(edge):
            return edge[1], edge[0]
    elif orientation == "ignore":
        def tailhead(edge):
            if edge[-1] == "reverse":
                return edge[1], edge[0]
            return edge[:2]
    explored = set()
    cycle = []
    final_node = None
    for start_node in G.nbunch_iter(source):
        if start_node in explored:
            continue
        edges = []
        seen = {start_node}
        active_nodes = {start_node}
        previous_head = None
        for edge in nx.edge_dfs(G, start_node, orientation):
            tail, head = tailhead(edge)
            if head in explored:
                continue
            if previous_head is not None and tail != previous_head:
                while True:
                    try:
                        popped_edge = edges.pop()
                    except IndexError:
                        edges = []
                        active_nodes = {tail}
                        break
                    else:
                        popped_head = tailhead(popped_edge)[1]
                        active_nodes.remove(popped_head)
                    if edges:
                        last_head = tailhead(edges[-1])[1]
                        if tail == last_head:
                            break
            edges.append(edge)
            if head in active_nodes:
                cycle.extend(edges)
                final_node = head
                break
```

```
(12)                        else:
(16)                            seen.add(head)
(16)                            active_nodes.add(head)
(16)                            previous_head = head
(8)                     if cycle:
(12)                        break
(8)                     else:
(12)                        explored.update(seen)
(4)                 else:
(8)                     assert len(cycle) == 0
(8)                     raise nx.exception.NetworkXNoCycle("No cycle found.")
(4)                 for i, edge in enumerate(cycle):
(8)                     tail, head = tailhead(edge)
(8)                     if tail == final_node:
(12)                        break
(4)                 return cycle[i:]
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def minimum_cycle_basis(G, weight=None):
(4)                 """Returns a minimum weight cycle basis for G
(4)                 Minimum weight means a cycle basis for which the total weight
(4)                 (length for unweighted graphs) of all the cycles is minimum.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 weight: string
(8)                     name of the edge attribute to use for edge weights
(4)                 Returns
(4)                 -------
(4)                 A list of cycle lists.  Each cycle list is a list of nodes
(4)                 which forms a cycle (loop) in G. Note that the nodes are not
(4)                 necessarily returned in a order by which they appear in the cycle
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> nx.add_cycle(G, [0, 1, 2, 3])
(4)                 >>> nx.add_cycle(G, [0, 3, 4, 5])
(4)                 >>> nx.minimum_cycle_basis(G)
(4)                 [[5, 4, 3, 0], [3, 2, 1, 0]]
(4)                 References:
(8)                     [1] Kavitha, Telikepalli, et al. "An O(m^2n) Algorithm for
(8)                     Minimum Cycle Basis of Graphs."
(8)                     http://link.springer.com/article/10.1007/s00453-007-9064-z
(8)                     [2] de Pina, J. 1995. Applications of shortest path methods.
(8)                     Ph.D. thesis, University of Amsterdam, Netherlands
(4)                 See Also
(4)                 --------
(4)                 simple_cycles, cycle_basis
(4)                 """
(4)                 return sum(
(8)                     (_min_cycle_basis(G.subgraph(c), weight) for c in
nx.connected_components(G)),
(8)                     [],
(4)                 )
(0)             def _min_cycle_basis(G, weight):
(4)                 cb = []
(4)                 tree_edges = list(nx.minimum_spanning_edges(G, weight=None, data=False))
(4)                 chords = G.edges - tree_edges - {(v, u) for u, v in tree_edges}
(4)                 set_orth = [{edge} for edge in chords]
(4)                 while set_orth:
(8)                     base = set_orth.pop()
(8)                     cycle_edges = _min_cycle(G, base, weight)
(8)                     cb.append([v for u, v in cycle_edges])
(8)                     set_orth = [
(12)                        (
(16)                            {e for e in orth if e not in base if e[::-1] not in base}
(16)                            | {e for e in base if e not in orth if e[::-1] not in orth}
(12)                        )
```

```
848: (12)                        if sum((e in orth or e[::-1] in orth) for e in cycle_edges) % 2
849: (12)                        else orth
850: (12)                        for orth in set_orth
851: (8)                     ]
852: (4)                 return cb
853: (0)             def _min_cycle(G, orth, weight):
854: (4)                 """
855: (4)                 Computes the minimum weight cycle in G,
856: (4)                 orthogonal to the vector orth as per [p. 338, 1]
857: (4)                 Use (u, 1) to indicate the lifted copy of u (denoted u' in paper).
858: (4)                 """
859: (4)                 Gi = nx.Graph()
860: (4)                 for u, v, wt in G.edges(data=weight, default=1):
861: (8)                     if (u, v) in orth or (v, u) in orth:
862: (12)                        Gi.add_edges_from([(u, (v, 1)), ((u, 1), v)], Gi_weight=wt)
863: (8)                     else:
864: (12)                        Gi.add_edges_from([(u, v), ((u, 1), (v, 1))], Gi_weight=wt)
865: (4)                 spl = nx.shortest_path_length
866: (4)                 lift = {n: spl(Gi, source=n, target=(n, 1), weight="Gi_weight") for n in
G}
867: (4)                 start = min(lift, key=lift.get)
868: (4)                 end = (start, 1)
869: (4)                 min_path_i = nx.shortest_path(Gi, source=start, target=end,
weight="Gi_weight")
870: (4)                 min_path = [n if n in G else n[0] for n in min_path_i]
871: (4)                 edgelist = list(pairwise(min_path))
872: (4)                 edgeset = set()
873: (4)                 for e in edgelist:
874: (8)                     if e in edgeset:
875: (12)                        edgeset.remove(e)
876: (8)                     elif e[::-1] in edgeset:
877: (12)                        edgeset.remove(e[::-1])
878: (8)                     else:
879: (12)                        edgeset.add(e)
880: (4)                 min_edgelist = []
881: (4)                 for e in edgelist:
882: (8)                     if e in edgeset:
883: (12)                        min_edgelist.append(e)
884: (12)                        edgeset.remove(e)
885: (8)                     elif e[::-1] in edgeset:
886: (12)                        min_edgelist.append(e[::-1])
887: (12)                        edgeset.remove(e[::-1])
888: (4)                 return min_edgelist
889: (0)             @not_implemented_for("directed")
890: (0)             @not_implemented_for("multigraph")
891: (0)             @nx._dispatchable
892: (0)             def girth(G):
893: (4)                 """Returns the girth of the graph.
894: (4)                 The girth of a graph is the length of its shortest cycle, or infinity if
895: (4)                 the graph is acyclic. The algorithm follows the description given on the
896: (4)                 Wikipedia page [1]_, and runs in time O(mn) on a graph with m edges and n
897: (4)                 nodes.
898: (4)                 Parameters
899: (4)                 ----------
900: (4)                 G : NetworkX Graph
901: (4)                 Returns
902: (4)                 -------
903: (4)                 int or math.inf
904: (4)                 Examples
905: (4)                 --------
906: (4)                 All examples below (except P_5) can easily be checked using Wikipedia,
907: (4)                 which has a page for each of these famous graphs.
908: (4)                 >>> nx.girth(nx.chvatal_graph())
909: (4)                 4
910: (4)                 >>> nx.girth(nx.tutte_graph())
911: (4)                 4
912: (4)                 >>> nx.girth(nx.petersen_graph())
913: (4)                 5
914: (4)                 >>> nx.girth(nx.heawood_graph())
```

```
915: (4)                6
916: (4)                >>> nx.girth(nx.pappus_graph())
917: (4)                6
918: (4)                >>> nx.girth(nx.path_graph(5))
919: (4)                inf
920: (4)                References
921: (4)                ----------
922: (4)                .. [1] `Wikipedia: Girth
<https://en.wikipedia.org/wiki/Girth_(graph_theory)>`_
923: (4)                """
924: (4)                girth = depth_limit = inf
925: (4)                tree_edge = nx.algorithms.traversal.breadth_first_search.TREE_EDGE
926: (4)                level_edge = nx.algorithms.traversal.breadth_first_search.LEVEL_EDGE
927: (4)                for n in G:
928: (8)                    depth = {n: 0}
929: (8)                    for u, v, label in nx.bfs_labeled_edges(G, n):
930: (12)                       du = depth[u]
931: (12)                       if du > depth_limit:
932: (16)                           break
933: (12)                       if label is tree_edge:
934: (16)                           depth[v] = du + 1
935: (12)                       else:
936: (16)                           delta = label is level_edge
937: (16)                           length = du + du + 2 - delta
938: (16)                           if length < girth:
939: (20)                               girth = length
940: (20)                               depth_limit = du - delta
941: (4)                return girth
```

File 22 - d_separation.py:

```
1: (0)              """
2: (0)              Algorithm for testing d-separation in DAGs.
3: (0)              *d-separation* is a test for conditional independence in probability
4: (0)              distributions that can be factorized using DAGs.  It is a purely
5: (0)              graphical test that uses the underlying graph and makes no reference
6: (0)              to the actual distribution parameters.  See [1]_ for a formal
7: (0)              definition.
8: (0)              The implementation is based on the conceptually simple linear time
9: (0)              algorithm presented in [2]_.  Refer to [3]_, [4]_ for a couple of
10: (0)             alternative algorithms.
11: (0)             The functional interface in NetworkX consists of three functions:
12: (0)             - `find_minimal_d_separator` returns a minimal d-separator set ``z``.
13: (2)               That is, removing any node or nodes from it makes it no longer a d-
separator.
14: (0)             - `is_d_separator` checks if a given set is a d-separator.
15: (0)             - `is_minimal_d_separator` checks if a given set is a minimal d-separator.
16: (0)             D-separators
17: (0)             ------------
18: (0)             Here, we provide a brief overview of d-separation and related concepts that
19: (0)             are relevant for understanding it:
20: (0)             The ideas of d-separation and d-connection relate to paths being open or
blocked.
21: (0)             - A "path" is a sequence of nodes connected in order by edges. Unlike for most
22: (2)               graph theory analysis, the direction of the edges is ignored. Thus the path
23: (2)               can be thought of as a traditional path on the undirected version of the
graph.
24: (0)             - A "candidate d-separator" ``z`` is a set of nodes being considered as
25: (2)               possibly blocking all paths between two prescribed sets ``x`` and ``y`` of
nodes.
26: (2)               We refer to each node in the candidate d-separator as "known".
27: (0)             - A "collider" node on a path is a node that is a successor of its two
neighbor
28: (2)               nodes on the path. That is, ``c`` is a collider if the edge directions
29: (2)               along the path look like ``... u -> c <- v ...``.
30: (0)             - If a collider node or any of its descendants are "known", the collider
31: (2)               is called an "open collider". Otherwise it is a "blocking collider".
```

```
(0)             - Any path can be "blocked" in two ways. If the path contains a "known" node
(2)               that is not a collider, the path is blocked. Also, if the path contains a
(2)               collider that is not a "known" node, the path is blocked.
(0)             - A path is "open" if it is not blocked. That is, it is open if every node is
(2)               either an open collider or not a "known". Said another way, every
(2)               "known" in the path is a collider and every collider is open (has a
(2)               "known" as a inclusive descendant). The concept of "open path" is meant to
(2)               demonstrate a probabilistic conditional dependence between two nodes given
(2)               prescribed knowledge ("known" nodes).
(0)             - Two sets ``x`` and ``y`` of nodes are "d-separated" by a set of nodes ``z``
(2)               if all paths between nodes in ``x`` and nodes in ``y`` are blocked. That is,
(2)               if there are no open paths from any node in ``x`` to any node in ``y``.
(2)               Such a set ``z`` is a "d-separator" of ``x`` and ``y``.
(0)             - A "minimal d-separator" is a d-separator ``z`` for which no node or subset
(2)               of nodes can be removed with it still being a d-separator.
(0)             The d-separator blocks some paths between ``x`` and ``y`` but opens others.
(0)             Nodes in the d-separator block paths if the nodes are not colliders.
(0)             But if a collider or its descendant nodes are in the d-separation set, the
(0)             colliders are open, allowing a path through that collider.
(0)             Illustration of D-separation with examples
(0)             ----------------------------------------
(0)             A pair of two nodes, ``u`` and ``v``, are d-connected if there is a path
(0)             from ``u`` to ``v`` that is not blocked. That means, there is an open
(0)             path from ``u`` to ``v``.
(0)             For example, if the d-separating set is the empty set, then the following
paths are
(0)             open between ``u`` and ``v``:
(0)             - u <- n -> v
(0)             - u -> w -> ... -> n -> v
(0)             If  on the other hand, ``n`` is in the d-separating set, then ``n`` blocks
(0)             those paths between ``u`` and ``v``.
(0)             Colliders block a path if they and their descendants are not included
(0)             in the d-separating set. An example of a path that is blocked when the
(0)             d-separating set is empty is:
(0)             - u -> w -> ... -> n <- v
(0)             The node ``n`` is a collider in this path and is not in the d-separating set.
(0)             So ``n`` blocks this path. However, if ``n`` or a descendant of ``n`` is
(0)             included in the d-separating set, then the path through the collider
(0)             at ``n`` (... -> n <- ...) is "open".
(0)             D-separation is concerned with blocking all paths between nodes from ``x`` to
``y``.
(0)             A d-separating set between ``x`` and ``y`` is one where all paths are blocked.
(0)             D-separation and its applications in probability
(0)             ------------------------------------------------
(0)             D-separation is commonly used in probabilistic causal-graph models. D-
separation
(0)             connects the idea of probabilistic "dependence" with separation in a graph. If
(0)             one assumes the causal Markov condition [5]_, (every node is conditionally
(0)             independent of its non-descendants, given its parents) then d-separation
implies
(0)             conditional independence in probability distributions.
(0)             Symmetrically, d-connection implies dependence.
(0)             The intuition is as follows. The edges on a causal graph indicate which nodes
(0)             influence the outcome of other nodes directly. An edge from u to v
(0)             implies that the outcome of event ``u`` influences the probabilities for
(0)             the outcome of event ``v``. Certainly knowing ``u`` changes predictions for
``v``.
(0)             But also knowing ``v`` changes predictions for ``u``. The outcomes are
dependent.
(0)             Furthermore, an edge from ``v`` to ``w`` would mean that ``w`` and ``v`` are
dependent
(0)             and thus that ``u`` could indirectly influence ``w``.
(0)             Without any knowledge about the system (candidate d-separating set is empty)
(0)             a causal graph ``u -> v -> w`` allows all three nodes to be dependent. But
(0)             if we know the outcome of ``v``, the conditional probabilities of outcomes for
(0)             ``u`` and ``w`` are independent of each other. That is, once we know the
outcome
(0)             for ```v`, the probabilities for ``w`` do not depend on the outcome for ``u``.
(0)             This is the idea behind ``v`` blocking the path if it is "known" (in the
```

```
candidate
d-separating set).
The same argument works whether the direction of the edges are both
left-going and when both arrows head out from the middle. Having a "known"
node on a path blocks the collider-free path because those relationships
make the conditional probabilities independent.
The direction of the causal edges does impact dependence precisely in the
case of a collider e.g. ``u -> v <- w``. In that situation, both ``u`` and
``w``
influence ``v```. But they do not directly influence each other. So without
any
knowledge of any outcomes, ``u`` and ``w`` are independent. That is the idea
behind
colliders blocking the path. But, if ``v`` is known, the conditional
probabilities
of ``u`` and ``w`` can be dependent. This is the heart of Berkson's Paradox
[6]_.
For example, suppose ``u`` and ``w`` are boolean events (they either happen or
do not)
and ``v`` represents the outcome "at least one of ``u`` and ``w`` occur". Then
knowing
``v`` is true makes the conditional probabilities of ``u`` and ``w``
dependent.
Essentially, knowing that at least one of them is true raises the probability
of
each. But further knowledge that ``w`` is true (or false) change the
conditional
probability of ``u`` to either the original value or 1. So the conditional
probability of ``u`` depends on the outcome of ``w`` even though there is no
causal relationship between them. When a collider is known, dependence can
occur across paths through that collider. This is the reason open colliders
do not block paths.
Furthermore, even if ``v`` is not "known", if one of its descendants is
"known"
we can use that information to know more about ``v`` which again makes
``u`` and ``w`` potentially dependent. Suppose the chance of ``n`` occurring
is much higher when ``v`` occurs ("at least one of ``u`` and ``w`` occur").
Then if we know ``n`` occurred, it is more likely that ``v`` occurred and that
makes the chance of ``u`` and ``w`` dependent. This is the idea behind why
a collider does no block a path if any descendant of the collider is "known".
When two sets of nodes ``x`` and ``y`` are d-separated by a set ``z``,
it means that given the outcomes of the nodes in ``z``, the probabilities
of outcomes of the nodes in ``x`` are independent of the outcomes of the
nodes in ``y`` and vice versa.
Examples
--------
A Hidden Markov Model with 5 observed states and 5 hidden states
where the hidden states have causal relationships resulting in
a path results in the following causal network. We check that
early states along the path are separated from late state in
the path by the d-separator of the middle hidden state.
Thus if we condition on the middle hidden state, the early
state probabilities are independent of the late state outcomes.
>>> G = nx.DiGraph()
>>> G.add_edges_from(
...     [
...         ("H1", "H2"),
...         ("H2", "H3"),
...         ("H3", "H4"),
...         ("H4", "H5"),
...         ("H1", "O1"),
...         ("H2", "O2"),
...         ("H3", "O3"),
...         ("H4", "O4"),
...         ("H5", "O5"),
...     ]
... )
>>> x, y, z = ({"H1", "O1"}, {"H5", "O5"}, {"H3"})
>>> nx.is_d_separator(G, x, y, z)
```

```
150: (0)            True
151: (0)            >>> nx.is_minimal_d_separator(G, x, y, z)
152: (0)            True
153: (0)            >>> nx.is_minimal_d_separator(G, x, y, z | {"O3"})
154: (0)            False
155: (0)            >>> z = nx.find_minimal_d_separator(G, x | y, {"O2", "O3", "O4"})
156: (0)            >>> z == {"H2", "H4"}
157: (0)            True
158: (0)            If no minimal_d_separator exists, `None` is returned
159: (0)            >>> other_z = nx.find_minimal_d_separator(G, x | y, {"H2", "H3"})
160: (0)            >>> other_z is None
161: (0)            True
162: (0)            References
163: (0)            ----------
164: (0)            .. [1] Pearl, J.  (2009).  Causality.  Cambridge: Cambridge University Press.
165: (0)            .. [2] Darwiche, A.  (2009).  Modeling and reasoning with Bayesian networks.
166: (3)               Cambridge: Cambridge University Press.
167: (0)            .. [3] Shachter, Ross D. "Bayes-ball: The rational pastime (for
168: (3)               determining irrelevance and requisite information in belief networks
169: (3)               and influence diagrams)." In Proceedings of the Fourteenth Conference
170: (3)               on Uncertainty in Artificial Intelligence (UAI), (pp. 480–487). 1998.
171: (0)            .. [4] Koller, D., & Friedman, N. (2009).
172: (3)               Probabilistic graphical models: principles and techniques. The MIT Press.
173: (0)            .. [5] https://en.wikipedia.org/wiki/Causal_Markov_condition
174: (0)            .. [6] https://en.wikipedia.org/wiki/Berkson%27s_paradox
175: (0)            """
176: (0)            from collections import deque
177: (0)            from itertools import chain
178: (0)            import networkx as nx
179: (0)            from networkx.utils import UnionFind, not_implemented_for
180: (0)            __all__ = [
181: (4)                "is_d_separator",
182: (4)                "is_minimal_d_separator",
183: (4)                "find_minimal_d_separator",
184: (4)                "d_separated",
185: (4)                "minimal_d_separator",
186: (0)            ]
187: (0)            @not_implemented_for("undirected")
188: (0)            @nx._dispatchable
189: (0)            def is_d_separator(G, x, y, z):
190: (4)                """Return whether node sets `x` and `y` are d-separated by `z`.
191: (4)                Parameters
192: (4)                ----------
193: (4)                G : nx.DiGraph
194: (8)                    A NetworkX DAG.
195: (4)                x : node or set of nodes
196: (8)                    First node or set of nodes in `G`.
197: (4)                y : node or set of nodes
198: (8)                    Second node or set of nodes in `G`.
199: (4)                z : node or set of nodes
200: (8)                    Potential separator (set of conditioning nodes in `G`). Can be empty
set.
201: (4)                Returns
202: (4)                -------
203: (4)                b : bool
204: (8)                    A boolean that is true if `x` is d-separated from `y` given `z` in
`G`.
205: (4)                Raises
206: (4)                ------
207: (4)                NetworkXError
208: (8)                    The *d-separation* test is commonly used on disjoint sets of
209: (8)                    nodes in acyclic directed graphs.  Accordingly, the algorithm
210: (8)                    raises a :exc:`NetworkXError` if the node sets are not
211: (8)                    disjoint or if the input graph is not a DAG.
212: (4)                NodeNotFound
213: (8)                    If any of the input nodes are not found in the graph,
214: (8)                    a :exc:`NodeNotFound` exception is raised
215: (4)                Notes
216: (4)                -----
```

```
(4)                 A d-separating set in a DAG is a set of nodes that
(4)                 blocks all paths between the two sets. Nodes in `z`
(4)                 block a path if they are part of the path and are not a collider,
(4)                 or a descendant of a collider. Also colliders that are not in `z`
(4)                 block a path. A collider structure along a path
(4)                 is ``... -> c <- ...`` where ``c`` is the collider node.
(4)                 https://en.wikipedia.org/wiki/Bayesian_network#d-separation
(4)                 """
(4)                 try:
(8)                     x = {x} if x in G else x
(8)                     y = {y} if y in G else y
(8)                     z = {z} if z in G else z
(8)                     intersection = x & y or x & z or y & z
(8)                     if intersection:
(12)                        raise nx.NetworkXError(
(16)                            f"The sets are not disjoint, with intersection {intersection}"
(12)                        )
(8)                     set_v = x | y | z
(8)                     if set_v - G.nodes:
(12)                        raise nx.NodeNotFound(f"The node(s) {set_v - G.nodes} are not found in G")
(4)                 except TypeError:
(8)                     raise nx.NodeNotFound("One of x, y, or z is not a node or a set of nodes in G")
(4)                 if not nx.is_directed_acyclic_graph(G):
(8)                     raise nx.NetworkXError("graph should be directed acyclic")
(4)                 forward_deque = deque([])
(4)                 forward_visited = set()
(4)                 backward_deque = deque(x)
(4)                 backward_visited = set()
(4)                 ancestors_or_z = set().union(*[nx.ancestors(G, node) for node in x]) | z | x
(4)                 while forward_deque or backward_deque:
(8)                     if backward_deque:
(12)                        node = backward_deque.popleft()
(12)                        backward_visited.add(node)
(12)                        if node in y:
(16)                            return False
(12)                        if node in z:
(16)                            continue
(12)                        backward_deque.extend(G.pred[node].keys() - backward_visited)
(12)                        forward_deque.extend(G.succ[node].keys() - forward_visited)
(8)                     if forward_deque:
(12)                        node = forward_deque.popleft()
(12)                        forward_visited.add(node)
(12)                        if node in y:
(16)                            return False
(12)                        if node in ancestors_or_z:
(16)                            backward_deque.extend(G.pred[node].keys() - backward_visited)
(12)                        if node not in z:
(16)                            forward_deque.extend(G.succ[node].keys() - forward_visited)
(4)                 return True
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable
(0)             def find_minimal_d_separator(G, x, y, *, included=None, restricted=None):
(4)                 """Returns a minimal d-separating set between `x` and `y` if possible
(4)                 A d-separating set in a DAG is a set of nodes that blocks all
(4)                 paths between the two sets of nodes, `x` and `y`. This function
(4)                 constructs a d-separating set that is "minimal", meaning no nodes can
(4)                 be removed without it losing the d-separating property for `x` and `y`.
(4)                 If no d-separating sets exist for `x` and `y`, this returns `None`.
(4)                 In a DAG there may be more than one minimal d-separator between two
(4)                 sets of nodes. Minimal d-separators are not always unique. This function
(4)                 returns one minimal d-separator, or `None` if no d-separator exists.
(4)                 Uses the algorithm presented in [1]_. The complexity of the algorithm
(4)                 is :math:`O(m)`, where :math:`m` stands for the number of edges in
(4)                 the subgraph of G consisting of only the ancestors of `x` and `y`.
(4)                 For full details, see [1]_.
(4)                 Parameters
```

```
    ----------
    G : graph
        A networkx DAG.
    x : set | node
        A node or set of nodes in the graph.
    y : set | node
        A node or set of nodes in the graph.
    included : set | node | None
        A node or set of nodes which must be included in the found separating set,
        default is None, which means the empty set.
    restricted : set | node | None
        Restricted node or set of nodes to consider. Only these nodes can be in
        the found separating set, default is None meaning all nodes in ``G``.
    Returns
    -------
    z : set | None
        The minimal d-separating set, if at least one d-separating set exists,
        otherwise None.
    Raises
    ------
    NetworkXError
        Raises a :exc:`NetworkXError` if the input graph is not a DAG
        or if node sets `x`, `y`, and `included` are not disjoint.
    NodeNotFound
        If any of the input nodes are not found in the graph,
        a :exc:`NodeNotFound` exception is raised.
    References
    ----------
    .. [1] van der Zander, Benito, and Maciej Liśkiewicz. "Finding
        minimal d-separators in linear time and applications." In
        Uncertainty in Artificial Intelligence, pp. 637-647. PMLR, 2020.
    """
    if not nx.is_directed_acyclic_graph(G):
        raise nx.NetworkXError("graph should be directed acyclic")
    try:
        x = {x} if x in G else x
        y = {y} if y in G else y
        if included is None:
            included = set()
        elif included in G:
            included = {included}
        if restricted is None:
            restricted = set(G)
        elif restricted in G:
            restricted = {restricted}
        set_y = x | y | included | restricted
        if set_y - G.nodes:
            raise nx.NodeNotFound(f"The node(s) {set_y - G.nodes} are not found in G")
    except TypeError:
        raise nx.NodeNotFound(
            "One of x, y, included or restricted is not a node or set of nodes in G"
        )
    if not included <= restricted:
        raise nx.NetworkXError(
            f"Included nodes {included} must be in restricted nodes {restricted}"
        )
    intersection = x & y or x & included or y & included
    if intersection:
        raise nx.NetworkXError(
            f"The sets x, y, included are not disjoint. Overlap: {intersection}"
        )
    nodeset = x | y | included
    ancestors_x_y_included = nodeset.union(*[nx.ancestors(G, node) for node in
```

```
nodeset])
    z_init = restricted & (ancestors_x_y_included - (x | y))
    x_closure = _reachable(G, x, ancestors_x_y_included, z_init)
    if x_closure & y:
        return None
    z_updated = z_init & (x_closure | included)
    y_closure = _reachable(G, y, ancestors_x_y_included, z_updated)
    return z_updated & (y_closure | included)
@not_implemented_for("undirected")
@nx._dispatchable
def is_minimal_d_separator(G, x, y, z, *, included=None, restricted=None):
    """Determine if `z` is a minimal d-separator for `x` and `y`.
    A d-separator, `z`, in a DAG is a set of nodes that blocks
    all paths from nodes in set `x` to nodes in set `y`.
    A minimal d-separator is a d-separator `z` such that removing
    any subset of nodes makes it no longer a d-separator.
    Note: This function checks whether `z` is a d-separator AND is
    minimal. One can use the function `is_d_separator` to only check if
    `z` is a d-separator. See examples below.
    Parameters
    ----------
    G : nx.DiGraph
        A NetworkX DAG.
    x : node | set
        A node or set of nodes in the graph.
    y : node | set
        A node or set of nodes in the graph.
    z : node | set
        The node or set of nodes to check if it is a minimal d-separating set.
        The function :func:`is_d_separator` is called inside this function
        to verify that `z` is in fact a d-separator.
    included : set | node | None
        A node or set of nodes which must be included in the found separating set,
        default is ``None``, which means the empty set.
    restricted : set | node | None
        Restricted node or set of nodes to consider. Only these nodes can be in
        the found separating set, default is ``None`` meaning all nodes in ``G``.
    Returns
    -------
    bool
        Whether or not the set `z` is a minimal d-separator subject to
        `restricted` nodes and `included` node constraints.
    Examples
    --------
    >>> G = nx.path_graph([0, 1, 2, 3], create_using=nx.DiGraph)
    >>> G.add_node(4)
    >>> nx.is_minimal_d_separator(G, 0, 2, {1})
    True
    >>> # since {1} is the minimal d-separator, {1, 3, 4} is not minimal
    >>> nx.is_minimal_d_separator(G, 0, 2, {1, 3, 4})
    False
    >>> # alternatively, if we only want to check that {1, 3, 4} is a d-separator
    >>> nx.is_d_separator(G, 0, 2, {1, 3, 4})
    True
    Raises
    ------
    NetworkXError
        Raises a :exc:`NetworkXError` if the input graph is not a DAG.
    NodeNotFound
        If any of the input nodes are not found in the graph,
        a :exc:`NodeNotFound` exception is raised.
    References
    ----------
    .. [1] van der Zander, Benito, and Maciej Liśkiewicz. "Finding
        minimal d-separators in linear time and applications." In
```

```
410: (8)                    Uncertainty in Artificial Intelligence, pp. 637-647. PMLR, 2020.
411: (4)                Notes
412: (4)                -----
413: (4)                This function works on verifying that a set is minimal and
414: (4)                d-separating between two nodes. Uses criterion (a), (b), (c) on
415: (4)                page 4 of [1]_. a) closure(`x`) and `y` are disjoint. b) `z` contains
416: (4)                all nodes from `included` and is contained in the `restricted`
417: (4)                nodes and in the union of ancestors of `x`, `y`, and `included`.
418: (4)                c) the nodes in `z` not in `included` are contained in both
419: (4)                closure(x) and closure(y). The closure of a set is the set of nodes
420: (4)                connected to the set by a directed path in G.
421: (4)                The complexity is :math:`O(m)`, where :math:`m` stands for the
422: (4)                number of edges in the subgraph of G consisting of only the
423: (4)                ancestors of `x` and `y`.
424: (4)                For full details, see [1]_.
425: (4)                """
426: (4)                if not nx.is_directed_acyclic_graph(G):
427: (8)                    raise nx.NetworkXError("graph should be directed acyclic")
428: (4)                try:
429: (8)                    x = {x} if x in G else x
430: (8)                    y = {y} if y in G else y
431: (8)                    z = {z} if z in G else z
432: (8)                    if included is None:
433: (12)                       included = set()
434: (8)                    elif included in G:
435: (12)                       included = {included}
436: (8)                    if restricted is None:
437: (12)                       restricted = set(G)
438: (8)                    elif restricted in G:
439: (12)                       restricted = {restricted}
440: (8)                    set_y = x | y | included | restricted
441: (8)                    if set_y - G.nodes:
442: (12)                       raise nx.NodeNotFound(f"The node(s) {set_y - G.nodes} are not
found in G")
443: (4)                except TypeError:
444: (8)                    raise nx.NodeNotFound(
445: (12)                       "One of x, y, z, included or restricted is not a node or set of
nodes in G"
446: (8)                    )
447: (4)                if not included <= z:
448: (8)                    raise nx.NetworkXError(
449: (12)                       f"Included nodes {included} must be in proposed separating set z
{x}"
450: (8)                    )
451: (4)                if not z <= restricted:
452: (8)                    raise nx.NetworkXError(
453: (12)                       f"Separating set {z} must be contained in restricted set
{restricted}"
454: (8)                    )
455: (4)                intersection = x.intersection(y) or x.intersection(z) or y.intersection(z)
456: (4)                if intersection:
457: (8)                    raise nx.NetworkXError(
458: (12)                       f"The sets are not disjoint, with intersection {intersection}"
459: (8)                    )
460: (4)                nodeset = x | y | included
461: (4)                ancestors_x_y_included = nodeset.union(*[nx.ancestors(G, n) for n in
nodeset])
462: (4)                x_closure = _reachable(G, x, ancestors_x_y_included, z)
463: (4)                if x_closure & y:
464: (8)                    return False
465: (4)                if not (z <= ancestors_x_y_included):
466: (8)                    return False
467: (4)                y_closure = _reachable(G, y, ancestors_x_y_included, z)
468: (4)                if not ((z - included) <= (x_closure & y_closure)):
469: (8)                    return False
470: (4)                return True
471: (0)            @not_implemented_for("undirected")
472: (0)            def _reachable(G, x, a, z):
473: (4)                """Modified Bayes-Ball algorithm for finding d-connected nodes.
```

```
    Find all nodes in `a` that are d-connected to those in `x` by
    those in `z`. This is an implementation of the function
    `REACHABLE` in [1]_ (which is itself a modification of the
    Bayes-Ball algorithm [2]_) when restricted to DAGs.
    Parameters
    ----------
    G : nx.DiGraph
        A NetworkX DAG.
    x : node | set
        A node in the DAG, or a set of nodes.
    a : node | set
        A (set of) node(s) in the DAG containing the ancestors of `x`.
    z : node | set
        The node or set of nodes conditioned on when checking d-connectedness.
    Returns
    -------
    w : set
        The closure of `x` in `a` with respect to d-connectedness
        given `z`.
    References
    ----------
    .. [1] van der Zander, Benito, and Maciej Liśkiewicz. "Finding
        minimal d-separators in linear time and applications." In
        Uncertainty in Artificial Intelligence, pp. 637-647. PMLR, 2020.
    .. [2] Shachter, Ross D. "Bayes-ball: The rational pastime
       (for determining irrelevance and requisite information in
       belief networks and influence diagrams)." In Proceedings of the
       Fourteenth Conference on Uncertainty in Artificial Intelligence
       (UAI), (pp. 480–487). 1998.
    """
    def _pass(e, v, f, n):
        """Whether a ball entering node `v` along edge `e` passes to `n` along `f`.

        Boolean function defined on page 6 of [1]_.
        Parameters
        ----------
        e : bool
            Directed edge by which the ball got to node `v`; `True` iff directed into `v`.
        v : node
            Node where the ball is.
        f : bool
            Directed edge connecting nodes `v` and `n`; `True` iff directed `n`.
        n : node
            Checking whether the ball passes to this node.
        Returns
        -------
        b : bool
            Whether the ball passes or not.
        References
        ----------
        .. [1] van der Zander, Benito, and Maciej Liśkiewicz. "Finding
           minimal d-separators in linear time and applications." In
           Uncertainty in Artificial Intelligence, pp. 637-647. PMLR, 2020.
        """
        is_element_of_A = n in a
        collider_if_in_Z = v not in z or (e and not f)
        return is_element_of_A and collider_if_in_Z  # and almost_definite_status
    queue = deque([])
    for node in x:
        if bool(G.pred[node]):
            queue.append((True, node))
        if bool(G.succ[node]):
            queue.append((False, node))
    processed = queue.copy()
    while any(queue):
        e, v = queue.popleft()
```

```
539: (8)                    preds = ((False, n) for n in G.pred[v])
540: (8)                    succs = ((True, n) for n in G.succ[v])
541: (8)                    f_n_pairs = chain(preds, succs)
542: (8)                    for f, n in f_n_pairs:
543: (12)                       if (f, n) not in processed and _pass(e, v, f, n):
544: (16)                           queue.append((f, n))
545: (16)                           processed.append((f, n))
546: (4)                return {w for (_, w) in processed}
547: (0)            def d_separated(G, x, y, z):
548: (4)                """Return whether nodes sets ``x`` and ``y`` are d-separated by ``z``.
549: (4)                .. deprecated:: 3.3
550: (8)                    This function is deprecated and will be removed in NetworkX v3.5.
551: (8)                    Please use `is_d_separator(G, x, y, z)`.
552: (4)                """
553: (4)                import warnings
554: (4)                warnings.warn(
555: (8)                    "d_separated is deprecated and will be removed in NetworkX v3.5."
556: (8)                    "Please use `is_d_separator(G, x, y, z)`.",
557: (8)                    category=DeprecationWarning,
558: (8)                    stacklevel=2,
559: (4)                )
560: (4)                return nx.is_d_separator(G, x, y, z)
561: (0)            def minimal_d_separator(G, u, v):
562: (4)                """Returns a minimal_d-separating set between `x` and `y` if possible
563: (4)                .. deprecated:: 3.3
564: (8)                    minimal_d_separator is deprecated and will be removed in NetworkX
v3.5.
565: (8)                    Please use `find_minimal_d_separator(G, x, y)`.
566: (4)                """
567: (4)                import warnings
568: (4)                warnings.warn(
569: (8)                    (
570: (12)                       "This function is deprecated and will be removed in NetworkX
v3.5."
571: (12)                       "Please use `is_d_separator(G, x, y)`."
572: (8)                    ),
573: (8)                    category=DeprecationWarning,
574: (8)                    stacklevel=2,
575: (4)                )
576: (4)                return nx.find_minimal_d_separator(G, u, v)
```

File 23 - dag.py:

```
1: (0)              """Algorithms for directed acyclic graphs (DAGs).
2: (0)              Note that most of these functions are only guaranteed to work for DAGs.
3: (0)              In general, these functions do not check for acyclic-ness, so it is up
4: (0)              to the user to check for that.
5: (0)              """
6: (0)              import heapq
7: (0)              from collections import deque
8: (0)              from functools import partial
9: (0)              from itertools import chain, combinations, product, starmap
10: (0)             from math import gcd
11: (0)             import networkx as nx
12: (0)             from networkx.utils import arbitrary_element, not_implemented_for, pairwise
13: (0)             __all__ = [
14: (4)                 "descendants",
15: (4)                 "ancestors",
16: (4)                 "topological_sort",
17: (4)                 "lexicographical_topological_sort",
18: (4)                 "all_topological_sorts",
19: (4)                 "topological_generations",
20: (4)                 "is_directed_acyclic_graph",
21: (4)                 "is_aperiodic",
22: (4)                 "transitive_closure",
23: (4)                 "transitive_closure_dag",
24: (4)                 "transitive_reduction",
```

```
25: (4)                 "antichains",
26: (4)                 "dag_longest_path",
27: (4)                 "dag_longest_path_length",
28: (4)                 "dag_to_branching",
29: (4)                 "compute_v_structures",
30: (0)             ]
31: (0)             chaini = chain.from_iterable
32: (0)             @nx._dispatchable
33: (0)             def descendants(G, source):
34: (4)                 """Returns all nodes reachable from `source` in `G`.
35: (4)                 Parameters
36: (4)                 ----------
37: (4)                 G : NetworkX Graph
38: (4)                 source : node in `G`
39: (4)                 Returns
40: (4)                 -------
41: (4)                 set()
42: (8)                     The descendants of `source` in `G`
43: (4)                 Raises
44: (4)                 ------
45: (4)                 NetworkXError
46: (8)                     If node `source` is not in `G`.
47: (4)                 Examples
48: (4)                 --------
49: (4)                 >>> DG = nx.path_graph(5, create_using=nx.DiGraph)
50: (4)                 >>> sorted(nx.descendants(DG, 2))
51: (4)                 [3, 4]
52: (4)                 The `source` node is not a descendant of itself, but can be included
manually:
53: (4)                 >>> sorted(nx.descendants(DG, 2) | {2})
54: (4)                 [2, 3, 4]
55: (4)                 See also
56: (4)                 --------
57: (4)                 ancestors
58: (4)                 """
59: (4)                 return {child for parent, child in nx.bfs_edges(G, source)}
60: (0)             @nx._dispatchable
61: (0)             def ancestors(G, source):
62: (4)                 """Returns all nodes having a path to `source` in `G`.
63: (4)                 Parameters
64: (4)                 ----------
65: (4)                 G : NetworkX Graph
66: (4)                 source : node in `G`
67: (4)                 Returns
68: (4)                 -------
69: (4)                 set()
70: (8)                     The ancestors of `source` in `G`
71: (4)                 Raises
72: (4)                 ------
73: (4)                 NetworkXError
74: (8)                     If node `source` is not in `G`.
75: (4)                 Examples
76: (4)                 --------
77: (4)                 >>> DG = nx.path_graph(5, create_using=nx.DiGraph)
78: (4)                 >>> sorted(nx.ancestors(DG, 2))
79: (4)                 [0, 1]
80: (4)                 The `source` node is not an ancestor of itself, but can be included
manually:
81: (4)                 >>> sorted(nx.ancestors(DG, 2) | {2})
82: (4)                 [0, 1, 2]
83: (4)                 See also
84: (4)                 --------
85: (4)                 descendants
86: (4)                 """
87: (4)                 return {child for parent, child in nx.bfs_edges(G, source, reverse=True)}
88: (0)             @nx._dispatchable
89: (0)             def has_cycle(G):
90: (4)                 """Decides whether the directed graph has a cycle."""
91: (4)                 try:
```

```
92: (8)                     deque(topological_sort(G), maxlen=0)
93: (4)                 except nx.NetworkXUnfeasible:
94: (8)                     return True
95: (4)                 else:
96: (8)                     return False
97: (0)             @nx._dispatchable
98: (0)             def is_directed_acyclic_graph(G):
99: (4)                 """Returns True if the graph `G` is a directed acyclic graph (DAG) or
100: (4)                False if not.
101: (4)                Parameters
102: (4)                ----------
103: (4)                G : NetworkX graph
104: (4)                Returns
105: (4)                -------
106: (4)                bool
107: (8)                    True if `G` is a DAG, False otherwise
108: (4)                Examples
109: (4)                --------
110: (4)                Undirected graph::
111: (8)                    >>> G = nx.Graph([(1, 2), (2, 3)])
112: (8)                    >>> nx.is_directed_acyclic_graph(G)
113: (8)                    False
114: (4)                Directed graph with cycle::
115: (8)                    >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
116: (8)                    >>> nx.is_directed_acyclic_graph(G)
117: (8)                    False
118: (4)                Directed acyclic graph::
119: (8)                    >>> G = nx.DiGraph([(1, 2), (2, 3)])
120: (8)                    >>> nx.is_directed_acyclic_graph(G)
121: (8)                    True
122: (4)                See also
123: (4)                --------
124: (4)                topological_sort
125: (4)                """
126: (4)                return G.is_directed() and not has_cycle(G)
127: (0)            @nx._dispatchable
128: (0)            def topological_generations(G):
129: (4)                """Stratifies a DAG into generations.
130: (4)                A topological generation is node collection in which ancestors of a node
in each
131: (4)                generation are guaranteed to be in a previous generation, and any
descendants of
132: (4)                a node are guaranteed to be in a following generation. Nodes are
guaranteed to
133: (4)                be in the earliest possible generation that they can belong to.
134: (4)                Parameters
135: (4)                ----------
136: (4)                G : NetworkX digraph
137: (8)                    A directed acyclic graph (DAG)
138: (4)                Yields
139: (4)                ------
140: (4)                sets of nodes
141: (8)                    Yields sets of nodes representing each generation.
142: (4)                Raises
143: (4)                ------
144: (4)                NetworkXError
145: (8)                    Generations are defined for directed graphs only. If the graph
146: (8)                    `G` is undirected, a :exc:`NetworkXError` is raised.
147: (4)                NetworkXUnfeasible
148: (8)                    If `G` is not a directed acyclic graph (DAG) no topological
generations
149: (8)                    exist and a :exc:`NetworkXUnfeasible` exception is raised.  This can
also
150: (8)                    be raised if `G` is changed while the returned iterator is being
processed
151: (4)                RuntimeError
152: (8)                    If `G` is changed while the returned iterator is being processed.
153: (4)                Examples
154: (4)                --------
```

```
155: (4)                >>> DG = nx.DiGraph([(2, 1), (3, 1)])
156: (4)                >>> [sorted(generation) for generation in nx.topological_generations(DG)]
157: (4)                [[2, 3], [1]]
158: (4)                Notes
159: (4)                -----
160: (4)                The generation in which a node resides can also be determined by taking
the
161: (4)                max-path-distance from the node to the farthest leaf node. That value can
162: (4)                be obtained with this function using
`enumerate(topological_generations(G))`.
163: (4)                See also
164: (4)                --------
165: (4)                topological_sort
166: (4)                """
167: (4)                if not G.is_directed():
168: (8)                    raise nx.NetworkXError("Topological sort not defined on undirected
graphs.")
169: (4)                multigraph = G.is_multigraph()
170: (4)                indegree_map = {v: d for v, d in G.in_degree() if d > 0}
171: (4)                zero_indegree = [v for v, d in G.in_degree() if d == 0]
172: (4)                while zero_indegree:
173: (8)                    this_generation = zero_indegree
174: (8)                    zero_indegree = []
175: (8)                    for node in this_generation:
176: (12)                       if node not in G:
177: (16)                           raise RuntimeError("Graph changed during iteration")
178: (12)                       for child in G.neighbors(node):
179: (16)                           try:
180: (20)                               indegree_map[child] -= len(G[node][child]) if multigraph
else 1
181: (16)                           except KeyError as err:
182: (20)                               raise RuntimeError("Graph changed during iteration") from
err
183: (16)                           if indegree_map[child] == 0:
184: (20)                               zero_indegree.append(child)
185: (20)                               del indegree_map[child]
186: (8)                    yield this_generation
187: (4)                if indegree_map:
188: (8)                    raise nx.NetworkXUnfeasible(
189: (12)                       "Graph contains a cycle or graph changed during iteration"
190: (8)                    )
191: (0)            @nx._dispatchable
192: (0)            def topological_sort(G):
193: (4)                """Returns a generator of nodes in topologically sorted order.
194: (4)                A topological sort is a nonunique permutation of the nodes of a
195: (4)                directed graph such that an edge from u to v implies that u
196: (4)                appears before v in the topological sort order. This ordering is
197: (4)                valid only if the graph has no directed cycles.
198: (4)                Parameters
199: (4)                ----------
200: (4)                G : NetworkX digraph
201: (8)                    A directed acyclic graph (DAG)
202: (4)                Yields
203: (4)                ------
204: (4)                nodes
205: (8)                    Yields the nodes in topological sorted order.
206: (4)                Raises
207: (4)                ------
208: (4)                NetworkXError
209: (8)                    Topological sort is defined for directed graphs only. If the graph `G`
210: (8)                    is undirected, a :exc:`NetworkXError` is raised.
211: (4)                NetworkXUnfeasible
212: (8)                    If `G` is not a directed acyclic graph (DAG) no topological sort
exists
213: (8)                    and a :exc:`NetworkXUnfeasible` exception is raised.  This can also be
214: (8)                    raised if `G` is changed while the returned iterator is being
processed
215: (4)                RuntimeError
216: (8)                    If `G` is changed while the returned iterator is being processed.
```

```
217: (4)                Examples
218: (4)                --------
219: (4)                To get the reverse order of the topological sort:
220: (4)                >>> DG = nx.DiGraph([(1, 2), (2, 3)])
221: (4)                >>> list(reversed(list(nx.topological_sort(DG))))
222: (4)                [3, 2, 1]
223: (4)                If your DiGraph naturally has the edges representing tasks/inputs
224: (4)                and nodes representing people/processes that initiate tasks, then
225: (4)                topological_sort is not quite what you need. You will have to change
226: (4)                the tasks to nodes with dependence reflected by edges. The result is
227: (4)                a kind of topological sort of the edges. This can be done
228: (4)                with :func:`networkx.line_graph` as follows:
229: (4)                >>> list(nx.topological_sort(nx.line_graph(DG)))
230: (4)                [(1, 2), (2, 3)]
231: (4)                Notes
232: (4)                -----
233: (4)                This algorithm is based on a description and proof in
234: (4)                "Introduction to Algorithms: A Creative Approach" [1]_ .
235: (4)                See also
236: (4)                --------
237: (4)                is_directed_acyclic_graph, lexicographical_topological_sort
238: (4)                References
239: (4)                ----------
240: (4)                .. [1] Manber, U. (1989).
241: (7)                   *Introduction to Algorithms - A Creative Approach.* Addison-Wesley.
242: (4)                """
243: (4)                for generation in nx.topological_generations(G):
244: (8)                    yield from generation
245: (0)            @nx._dispatchable
246: (0)            def lexicographical_topological_sort(G, key=None):
247: (4)                """Generate the nodes in the unique lexicographical topological sort
order.
248: (4)                Generates a unique ordering of nodes by first sorting topologically (for
which there are often
249: (4)                multiple valid orderings) and then additionally by sorting
lexicographically.
250: (4)                A topological sort arranges the nodes of a directed graph so that the
251: (4)                upstream node of each directed edge precedes the downstream node.
252: (4)                It is always possible to find a solution for directed graphs that have no
cycles.
253: (4)                There may be more than one valid solution.
254: (4)                Lexicographical sorting is just sorting alphabetically. It is used here to
break ties in the
255: (4)                topological sort and to determine a single, unique ordering.  This can be
useful in comparing
256: (4)                sort results.
257: (4)                The lexicographical order can be customized by providing a function to the
`key=` parameter.
258: (4)                The definition of the key function is the same as used in python's built-
in `sort()`.
259: (4)                The function takes a single argument and returns a key to use for sorting
purposes.
260: (4)                Lexicographical sorting can fail if the node names are un-sortable. See
the example below.
261: (4)                The solution is to provide a function to the `key=` argument that returns
sortable keys.
262: (4)                Parameters
263: (4)                ----------
264: (4)                G : NetworkX digraph
265: (8)                    A directed acyclic graph (DAG)
266: (4)                key : function, optional
267: (8)                    A function of one argument that converts a node name to a comparison
key.
268: (8)                    It defines and resolves ambiguities in the sort order.  Defaults to
the identity function.
269: (4)                Yields
270: (4)                ------
271: (4)                nodes
272: (8)                    Yields the nodes of G in lexicographical topological sort order.
```

```
    Raises
    ------
    NetworkXError
        Topological sort is defined for directed graphs only. If the graph `G`
        is undirected, a :exc:`NetworkXError` is raised.
    NetworkXUnfeasible
        If `G` is not a directed acyclic graph (DAG) no topological sort exists
        and a :exc:`NetworkXUnfeasible` exception is raised.  This can also be
        raised if `G` is changed while the returned iterator is being processed
    RuntimeError
        If `G` is changed while the returned iterator is being processed.
    TypeError
        Results from un-sortable node names.
        Consider using `key=` parameter to resolve ambiguities in the sort order.
    Examples
    --------
    >>> DG = nx.DiGraph([(2, 1), (2, 5), (1, 3), (1, 4), (5, 4)])
    >>> list(nx.lexicographical_topological_sort(DG))
    [2, 1, 3, 5, 4]
    >>> list(nx.lexicographical_topological_sort(DG, key=lambda x: -x))
    [2, 5, 1, 4, 3]
    The sort will fail for any graph with integer and string nodes. Comparison of integer to strings
    is not defined in python.  Is 3 greater or less than 'red'?
    >>> DG = nx.DiGraph([(1, "red"), (3, "red"), (1, "green"), (2, "blue")])
    >>> list(nx.lexicographical_topological_sort(DG))
    Traceback (most recent call last):
    ...
    TypeError: '<' not supported between instances of 'str' and 'int'
    ...
    Incomparable nodes can be resolved using a `key` function. This example function
    allows comparison of integers and strings by returning a tuple where the first
    element is True for `str`, False otherwise. The second element is the node name.
    This groups the strings and integers separately so they can be compared only among themselves.
    >>> key = lambda node: (isinstance(node, str), node)
    >>> list(nx.lexicographical_topological_sort(DG, key=key))
    [1, 2, 3, 'blue', 'green', 'red']
    Notes
    -----
    This algorithm is based on a description and proof in
    "Introduction to Algorithms: A Creative Approach" [1]_ .
    See also
    --------
    topological_sort
    References
    ----------
    .. [1] Manber, U. (1989).
       *Introduction to Algorithms - A Creative Approach.* Addison-Wesley.
    """
    if not G.is_directed():
        msg = "Topological sort not defined on undirected graphs."
        raise nx.NetworkXError(msg)
    if key is None:
        def key(node):
            return node
    nodeid_map = {n: i for i, n in enumerate(G)}
    def create_tuple(node):
        return key(node), nodeid_map[node], node
    indegree_map = {v: d for v, d in G.in_degree() if d > 0}
    zero_indegree = [create_tuple(v) for v, d in G.in_degree() if d == 0]
    heapq.heapify(zero_indegree)
    while zero_indegree:
```

```
(8)                     _, _, node = heapq.heappop(zero_indegree)
(8)                     if node not in G:
(12)                        raise RuntimeError("Graph changed during iteration")
(8)                     for _, child in G.edges(node):
(12)                        try:
(16)                            indegree_map[child] -= 1
(12)                        except KeyError as err:
(16)                            raise RuntimeError("Graph changed during iteration") from err
(12)                        if indegree_map[child] == 0:
(16)                            try:
(20)                                heapq.heappush(zero_indegree, create_tuple(child))
(16)                            except TypeError as err:
(20)                                raise TypeError(
(24)                                    f"{err}\nConsider using `key=` parameter to resolve ambiguities in the sort order."
(20)                                )
(16)                            del indegree_map[child]
(8)                     yield node
(4)                 if indegree_map:
(8)                     msg = "Graph contains a cycle or graph changed during iteration"
(8)                     raise nx.NetworkXUnfeasible(msg)
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable
(0)             def all_topological_sorts(G):
(4)                 """Returns a generator of _all_ topological sorts of the directed graph G.
(4)                 A topological sort is a nonunique permutation of the nodes such that an
(4)                 edge from u to v implies that u appears before v in the topological sort
(4)                 order.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX DiGraph
(8)                     A directed graph
(4)                 Yields
(4)                 ------
(4)                 topological_sort_order : list
(8)                     a list of nodes in `G`, representing one of the topological sort orders
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is not directed
(4)                 NetworkXUnfeasible
(8)                     If `G` is not acyclic
(4)                 Examples
(4)                 --------
(4)                 To enumerate all topological sorts of directed graph:
(4)                 >>> DG = nx.DiGraph([(1, 2), (2, 3), (2, 4)])
(4)                 >>> list(nx.all_topological_sorts(DG))
(4)                 [[1, 2, 4, 3], [1, 2, 3, 4]]
(4)                 Notes
(4)                 -----
(4)                 Implements an iterative version of the algorithm given in [1].
(4)                 References
(4)                 ----------
(4)                 .. [1] Knuth, Donald E., Szwarcfiter, Jayme L. (1974).
(7)                    "A Structured Program to Generate All Topological Sorting Arrangements"
(7)                    Information Processing Letters, Volume 2, Issue 6, 1974, Pages 153-157,
(7)                    ISSN 0020-0190,
(7)                    https://doi.org/10.1016/0020-0190(74)90001-5.
(7)                    Elsevier (North-Holland), Amsterdam
(4)                 """
(4)                 if not G.is_directed():
(8)                     raise nx.NetworkXError("Topological sort not defined on undirected graphs.")
(4)                 count = dict(G.in_degree())
(4)                 D = deque([v for v, d in G.in_degree() if d == 0])
(4)                 bases = []
(4)                 current_sort = []
(4)                 while True:
```

```
(8)                     assert all(count[v] == 0 for v in D)
(8)                     if len(current_sort) == len(G):
(12)                        yield list(current_sort)
(12)                        while len(current_sort) > 0:
(16)                            assert len(bases) == len(current_sort)
(16)                            q = current_sort.pop()
(16)                            for _, j in G.out_edges(q):
(20)                                count[j] += 1
(20)                                assert count[j] >= 0
(16)                            while len(D) > 0 and count[D[-1]] > 0:
(20)                                D.pop()
(16)                            D.appendleft(q)
(16)                            if D[-1] == bases[-1]:
(20)                                bases.pop()
(16)                            else:
(20)                                break
(8)                     else:
(12)                        if len(D) == 0:
(16)                            raise nx.NetworkXUnfeasible("Graph contains a cycle.")
(12)                        q = D.pop()
(12)                        for _, j in G.out_edges(q):
(16)                            count[j] -= 1
(16)                            assert count[j] >= 0
(16)                            if count[j] == 0:
(20)                                D.append(j)
(12)                        current_sort.append(q)
(12)                        if len(bases) < len(current_sort):
(16)                            bases.append(q)
(8)                     if len(bases) == 0:
(12)                        break
(0)             @nx._dispatchable
(0)             def is_aperiodic(G):
(4)                 """Returns True if `G` is aperiodic.
(4)                 A directed graph is aperiodic if there is no integer k > 1 that
(4)                 divides the length of every cycle in the graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX DiGraph
(8)                     A directed graph
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     True if the graph is aperiodic False otherwise
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If `G` is not directed
(4)                 Examples
(4)                 --------
(4)                 A graph consisting of one cycle, the length of which is 2. Therefore ``k =
2``
(4)                 divides the length of every cycle in the graph and thus the graph
(4)                 is *not aperiodic*::
(8)                     >>> DG = nx.DiGraph([(1, 2), (2, 1)])
(8)                     >>> nx.is_aperiodic(DG)
(8)                     False
(4)                 A graph consisting of two cycles: one of length 2 and the other of length
3.
(4)                 The cycle lengths are coprime, so there is no single value of k where ``k
> 1``
(4)                 that divides each cycle length and therefore the graph is *aperiodic*::
(8)                     >>> DG = nx.DiGraph([(1, 2), (2, 3), (3, 1), (1, 4), (4, 1)])
(8)                     >>> nx.is_aperiodic(DG)
(8)                     True
(4)                 A graph consisting of two cycles: one of length 2 and the other of length
4.
(4)                 The lengths of the cycles share a common factor ``k = 2``, and therefore
(4)                 the graph is *not aperiodic*::
(8)                     >>> DG = nx.DiGraph([(1, 2), (2, 1), (3, 4), (4, 5), (5, 6), (6, 3)])
```

```
465: (8)                >>> nx.is_aperiodic(DG)
466: (8)                False
467: (4)            An acyclic graph, therefore the graph is *not aperiodic*::
468: (8)                >>> DG = nx.DiGraph([(1, 2), (2, 3)])
469: (8)                >>> nx.is_aperiodic(DG)
470: (8)                False
471: (4)            Notes
472: (4)            -----
473: (4)            This uses the method outlined in [1]_, which runs in $O(m)$ time
474: (4)            given $m$ edges in `G`. Note that a graph is not aperiodic if it is
475: (4)            acyclic as every integer trivial divides length 0 cycles.
476: (4)            References
477: (4)            ----------
478: (4)            .. [1] Jarvis, J. P.; Shier, D. R. (1996),
479: (7)               "Graph-theoretic analysis of finite Markov chains,"
480: (7)               in Shier, D. R.; Wallenius, K. T., Applied Mathematical Modeling:
481: (7)               A Multidisciplinary Approach, CRC Press.
482: (4)            """
483: (4)            if not G.is_directed():
484: (8)                raise nx.NetworkXError("is_aperiodic not defined for undirected
graphs")
485: (4)            if len(G) == 0:
486: (8)                raise nx.NetworkXPointlessConcept("Graph has no nodes.")
487: (4)            s = arbitrary_element(G)
488: (4)            levels = {s: 0}
489: (4)            this_level = [s]
490: (4)            g = 0
491: (4)            lev = 1
492: (4)            while this_level:
493: (8)                next_level = []
494: (8)                for u in this_level:
495: (12)                   for v in G[u]:
496: (16)                       if v in levels:  # Non-Tree Edge
497: (20)                           g = gcd(g, levels[u] - levels[v] + 1)
498: (16)                       else:  # Tree Edge
499: (20)                           next_level.append(v)
500: (20)                           levels[v] = lev
501: (8)                this_level = next_level
502: (8)                lev += 1
503: (4)            if len(levels) == len(G):  # All nodes in tree
504: (8)                return g == 1
505: (4)            else:
506: (8)                return g == 1 and nx.is_aperiodic(G.subgraph(set(G) - set(levels)))
507: (0)        @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
508: (0)        def transitive_closure(G, reflexive=False):
509: (4)            """Returns transitive closure of a graph
510: (4)            The transitive closure of G = (V,E) is a graph G+ = (V,E+) such that
511: (4)            for all v, w in V there is an edge (v, w) in E+ if and only if there
512: (4)            is a path from v to w in G.
513: (4)            Handling of paths from v to v has some flexibility within this definition.
514: (4)            A reflexive transitive closure creates a self-loop for the path
515: (4)            from v to v of length 0. The usual transitive closure creates a
516: (4)            self-loop only if a cycle exists (a path from v to v with length > 0).
517: (4)            We also allow an option for no self-loops.
518: (4)            Parameters
519: (4)            ----------
520: (4)            G : NetworkX Graph
521: (8)                A directed/undirected graph/multigraph.
522: (4)            reflexive : Bool or None, optional (default: False)
523: (8)                Determines when cycles create self-loops in the Transitive Closure.
524: (8)                If True, trivial cycles (length 0) create self-loops. The result
525: (8)                is a reflexive transitive closure of G.
526: (8)                If False (the default) non-trivial cycles create self-loops.
527: (8)                If None, self-loops are not created.
528: (4)            Returns
529: (4)            -------
530: (4)            NetworkX graph
531: (8)                The transitive closure of `G`
532: (4)            Raises
```

```
    ------
    NetworkXError
        If `reflexive` not in `{None, True, False}`
    Examples
    --------
    The treatment of trivial (i.e. length 0) cycles is controlled by the
    `reflexive` parameter.
    Trivial (i.e. length 0) cycles do not create self-loops when
    ``reflexive=False`` (the default)::
        >>> DG = nx.DiGraph([(1, 2), (2, 3)])
        >>> TC = nx.transitive_closure(DG, reflexive=False)
        >>> TC.edges()
        OutEdgeView([(1, 2), (1, 3), (2, 3)])
    However, nontrivial (i.e. length greater than 0) cycles create self-loops
    when ``reflexive=False`` (the default)::
        >>> DG = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
        >>> TC = nx.transitive_closure(DG, reflexive=False)
        >>> TC.edges()
        OutEdgeView([(1, 2), (1, 3), (1, 1), (2, 3), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3)])
    Trivial cycles (length 0) create self-loops when ``reflexive=True``::
        >>> DG = nx.DiGraph([(1, 2), (2, 3)])
        >>> TC = nx.transitive_closure(DG, reflexive=True)
        >>> TC.edges()
        OutEdgeView([(1, 2), (1, 1), (1, 3), (2, 3), (2, 2), (3, 3)])
    And the third option is not to create self-loops at all when ``reflexive=None``::
        >>> DG = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
        >>> TC = nx.transitive_closure(DG, reflexive=None)
        >>> TC.edges()
        OutEdgeView([(1, 2), (1, 3), (2, 3), (2, 1), (3, 1), (3, 2)])
    References
    ----------
    .. [1] https://www.ics.uci.edu/~eppstein/PADS/PartialOrder.py
    """
    TC = G.copy()
    if reflexive not in {None, True, False}:
        raise nx.NetworkXError("Incorrect value for the parameter `reflexive`")
    for v in G:
        if reflexive is None:
            TC.add_edges_from((v, u) for u in nx.descendants(G, v) if u not in TC[v])
        elif reflexive is True:
            TC.add_edges_from(
                (v, u) for u in nx.descendants(G, v) | {v} if u not in TC[v]
            )
        elif reflexive is False:
            TC.add_edges_from((v, e[1]) for e in nx.edge_bfs(G, v) if e[1] not in TC[v])
    return TC
@not_implemented_for("undirected")
@nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
def transitive_closure_dag(G, topo_order=None):
    """Returns the transitive closure of a directed acyclic graph.
    This function is faster than the function `transitive_closure`, but fails
    if the graph has a cycle.
    The transitive closure of G = (V,E) is a graph G+ = (V,E+) such that
    for all v, w in V there is an edge (v, w) in E+ if and only if there
    is a non-null path from v to w in G.
    Parameters
    ----------
    G : NetworkX DiGraph
        A directed acyclic graph (DAG)
    topo_order: list or tuple, optional
        A topological order for G (if None, the function will compute one)
    Returns
    -------
    NetworkX DiGraph
```

```
(8)                 The transitive closure of `G`
(4)             Raises
(4)             ------
(4)             NetworkXNotImplemented
(8)                 If `G` is not directed
(4)             NetworkXUnfeasible
(8)                 If `G` has a cycle
(4)             Examples
(4)             --------
(4)             >>> DG = nx.DiGraph([(1, 2), (2, 3)])
(4)             >>> TC = nx.transitive_closure_dag(DG)
(4)             >>> TC.edges()
(4)             OutEdgeView([(1, 2), (1, 3), (2, 3)])
(4)             Notes
(4)             -----
(4)             This algorithm is probably simple enough to be well-known but I didn't find
(4)             a mention in the literature.
(4)             """
(4)             if topo_order is None:
(8)                 topo_order = list(topological_sort(G))
(4)             TC = G.copy()
(4)             for v in reversed(topo_order):
(8)                 TC.add_edges_from((v, u) for u in nx.descendants_at_distance(TC, v, 2))
(4)             return TC
(0)         @not_implemented_for("undirected")
(0)         @nx._dispatchable(returns_graph=True)
(0)         def transitive_reduction(G):
(4)             """Returns transitive reduction of a directed graph
(4)             The transitive reduction of G = (V,E) is a graph G- = (V,E-) such that
(4)             for all v,w in V there is an edge (v,w) in E- if and only if (v,w) is
(4)             in E and there is no path from v to w in G with length greater than 1.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX DiGraph
(8)                 A directed acyclic graph (DAG)
(4)             Returns
(4)             -------
(4)             NetworkX DiGraph
(8)                 The transitive reduction of `G`
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If `G` is not a directed acyclic graph (DAG) transitive reduction is
(8)                 not uniquely defined and a :exc:`NetworkXError` exception is raised.
(4)             Examples
(4)             --------
(4)             To perform transitive reduction on a DiGraph:
(4)             >>> DG = nx.DiGraph([(1, 2), (2, 3), (1, 3)])
(4)             >>> TR = nx.transitive_reduction(DG)
(4)             >>> list(TR.edges)
(4)             [(1, 2), (2, 3)]
(4)             To avoid unnecessary data copies, this implementation does not return a
(4)             DiGraph with node/edge data.
(4)             To perform transitive reduction on a DiGraph and transfer node/edge data:
(4)             >>> DG = nx.DiGraph()
(4)             >>> DG.add_edges_from([(1, 2), (2, 3), (1, 3)], color="red")
(4)             >>> TR = nx.transitive_reduction(DG)
(4)             >>> TR.add_nodes_from(DG.nodes(data=True))
(4)             >>> TR.add_edges_from((u, v, DG.edges[u, v]) for u, v in TR.edges)
(4)             >>> list(TR.edges(data=True))
(4)             [(1, 2, {'color': 'red'}), (2, 3, {'color': 'red'})]
(4)             References
(4)             ----------
(4)             https://en.wikipedia.org/wiki/Transitive_reduction
(4)             """
(4)             if not is_directed_acyclic_graph(G):
(8)                 msg = "Directed Acyclic Graph required for transitive_reduction"
```

```
664: (8)                    raise nx.NetworkXError(msg)
665: (4)                TR = nx.DiGraph()
666: (4)                TR.add_nodes_from(G.nodes())
667: (4)                descendants = {}
668: (4)                check_count = dict(G.in_degree)
669: (4)                for u in G:
670: (8)                    u_nbrs = set(G[u])
671: (8)                    for v in G[u]:
672: (12)                       if v in u_nbrs:
673: (16)                           if v not in descendants:
674: (20)                               descendants[v] = {y for x, y in nx.dfs_edges(G, v)}
675: (16)                           u_nbrs -= descendants[v]
676: (12)                       check_count[v] -= 1
677: (12)                       if check_count[v] == 0:
678: (16)                           del descendants[v]
679: (8)                    TR.add_edges_from((u, v) for v in u_nbrs)
680: (4)                return TR
681: (0)            @not_implemented_for("undirected")
682: (0)            @nx._dispatchable
683: (0)            def antichains(G, topo_order=None):
684: (4)                """Generates antichains from a directed acyclic graph (DAG).
685: (4)                An antichain is a subset of a partially ordered set such that any
686: (4)                two elements in the subset are incomparable.
687: (4)                Parameters
688: (4)                ----------
689: (4)                G : NetworkX DiGraph
690: (8)                    A directed acyclic graph (DAG)
691: (4)                topo_order: list or tuple, optional
692: (8)                    A topological order for G (if None, the function will compute one)
693: (4)                Yields
694: (4)                ------
695: (4)                antichain : list
696: (8)                    a list of nodes in `G` representing an antichain
697: (4)                Raises
698: (4)                ------
699: (4)                NetworkXNotImplemented
700: (8)                    If `G` is not directed
701: (4)                NetworkXUnfeasible
702: (8)                    If `G` contains a cycle
703: (4)                Examples
704: (4)                --------
705: (4)                >>> DG = nx.DiGraph([(1, 2), (1, 3)])
706: (4)                >>> list(nx.antichains(DG))
707: (4)                [[], [3], [2], [2, 3], [1]]
708: (4)                Notes
709: (4)                -----
710: (4)                This function was originally developed by Peter Jipsen and Franco Saliola
711: (4)                for the SAGE project. It's included in NetworkX with permission from the
712: (4)                authors. Original SAGE code at:
713: (4)
https://github.com/sagemath/sage/blob/master/src/sage/combinat/posets/hasse_diagram.py
714: (4)                References
715: (4)                ----------
716: (4)                .. [1] Free Lattices, by R. Freese, J. Jezek and J. B. Nation,
717: (7)                   AMS, Vol 42, 1995, p. 226.
718: (4)                """
719: (4)                if topo_order is None:
720: (8)                    topo_order = list(nx.topological_sort(G))
721: (4)                TC = nx.transitive_closure_dag(G, topo_order)
722: (4)                antichains_stacks = [([], list(reversed(topo_order)))]
723: (4)                while antichains_stacks:
724: (8)                    (antichain, stack) = antichains_stacks.pop()
725: (8)                    yield antichain
726: (8)                    while stack:
727: (12)                       x = stack.pop()
728: (12)                       new_antichain = antichain + [x]
729: (12)                       new_stack = [t for t in stack if not ((t in TC[x]) or (x in
TC[t]))]
730: (12)                       antichains_stacks.append((new_antichain, new_stack))
```

```
(0)            @not_implemented_for("undirected")
(0)            @nx._dispatchable(edge_attrs={"weight": "default_weight"})
(0)            def dag_longest_path(G, weight="weight", default_weight=1, topo_order=None):
(4)                """Returns the longest path in a directed acyclic graph (DAG).
(4)                If `G` has edges with `weight` attribute the edge data are used as
(4)                weight values.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX DiGraph
(8)                    A directed acyclic graph (DAG)
(4)                weight : str, optional
(8)                    Edge data key to use for weight
(4)                default_weight : int, optional
(8)                    The weight of edges that do not have a weight attribute
(4)                topo_order: list or tuple, optional
(8)                    A topological order for `G` (if None, the function will compute one)
(4)                Returns
(4)                -------
(4)                list
(8)                    Longest path
(4)                Raises
(4)                ------
(4)                NetworkXNotImplemented
(8)                    If `G` is not directed
(4)                Examples
(4)                --------
(4)                >>> DG = nx.DiGraph([(0, 1, {"cost": 1}), (1, 2, {"cost": 1}), (0, 2,
{"cost": 42})])
(4)                >>> list(nx.all_simple_paths(DG, 0, 2))
(4)                [[0, 1, 2], [0, 2]]
(4)                >>> nx.dag_longest_path(DG)
(4)                [0, 1, 2]
(4)                >>> nx.dag_longest_path(DG, weight="cost")
(4)                [0, 2]
(4)                In the case where multiple valid topological orderings exist, `topo_order`
(4)                can be used to specify a specific ordering:
(4)                >>> DG = nx.DiGraph([(0, 1), (0, 2)])
(4)                >>> sorted(nx.all_topological_sorts(DG))  # Valid topological orderings
(4)                [[0, 1, 2], [0, 2, 1]]
(4)                >>> nx.dag_longest_path(DG, topo_order=[0, 1, 2])
(4)                [0, 1]
(4)                >>> nx.dag_longest_path(DG, topo_order=[0, 2, 1])
(4)                [0, 2]
(4)                See also
(4)                --------
(4)                dag_longest_path_length
(4)                """
(4)                if not G:
(8)                    return []
(4)                if topo_order is None:
(8)                    topo_order = nx.topological_sort(G)
(4)                dist = {}  # stores {v : (length, u)}
(4)                for v in topo_order:
(8)                    us = [
(12)                       (
(16)                           dist[u][0]
(16)                           + (
(20)                               max(data.values(), key=lambda x: x.get(weight,
default_weight))
(20)                               if G.is_multigraph()
(20)                               else data
(16)                           ).get(weight, default_weight),
(16)                           u,
(12)                       )
(12)                       for u, data in G.pred[v].items()
(8)                    ]
(8)                    maxu = max(us, key=lambda x: x[0]) if us else (0, v)
(8)                    dist[v] = maxu if maxu[0] >= 0 else (0, v)
(4)                u = None
```

```
(4)                 v = max(dist, key=lambda x: dist[x][0])
(4)                 path = []
(4)                 while u != v:
(8)                     path.append(v)
(8)                     u = v
(8)                     v = dist[v][1]
(4)                 path.reverse()
(4)                 return path
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable(edge_attrs={"weight": "default_weight"})
(0)             def dag_longest_path_length(G, weight="weight", default_weight=1):
(4)                 """Returns the longest path length in a DAG
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX DiGraph
(8)                     A directed acyclic graph (DAG)
(4)                 weight : string, optional
(8)                     Edge data key to use for weight
(4)                 default_weight : int, optional
(8)                     The weight of edges that do not have a weight attribute
(4)                 Returns
(4)                 -------
(4)                 int
(8)                     Longest path length
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is not directed
(4)                 Examples
(4)                 --------
(4)                 >>> DG = nx.DiGraph([(0, 1, {"cost": 1}), (1, 2, {"cost": 1}), (0, 2,
{"cost": 42})])
(4)                 >>> list(nx.all_simple_paths(DG, 0, 2))
(4)                 [[0, 1, 2], [0, 2]]
(4)                 >>> nx.dag_longest_path_length(DG)
(4)                 2
(4)                 >>> nx.dag_longest_path_length(DG, weight="cost")
(4)                 42
(4)                 See also
(4)                 --------
(4)                 dag_longest_path
(4)                 """
(4)                 path = nx.dag_longest_path(G, weight, default_weight)
(4)                 path_length = 0
(4)                 if G.is_multigraph():
(8)                     for u, v in pairwise(path):
(12)                        i = max(G[u][v], key=lambda x: G[u][v][x].get(weight,
default_weight))
(12)                        path_length += G[u][v][i].get(weight, default_weight)
(4)                 else:
(8)                     for u, v in pairwise(path):
(12)                        path_length += G[u][v].get(weight, default_weight)
(4)                 return path_length
(0)             @nx._dispatchable
(0)             def root_to_leaf_paths(G):
(4)                 """Yields root-to-leaf paths in a directed acyclic graph.
(4)                 `G` must be a directed acyclic graph. If not, the behavior of this
(4)                 function is undefined. A "root" in this graph is a node of in-degree
(4)                 zero and a "leaf" a node of out-degree zero.
(4)                 When invoked, this function iterates over each path from any root to
(4)                 any leaf. A path is a list of nodes.
(4)                 """
(4)                 roots = (v for v, d in G.in_degree() if d == 0)
(4)                 leaves = (v for v, d in G.out_degree() if d == 0)
(4)                 all_paths = partial(nx.all_simple_paths, G)
(4)                 return chaini(starmap(all_paths, product(roots, leaves)))
(0)             @not_implemented_for("multigraph")
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable(returns_graph=True)
```

```
(0)  def dag_to_branching(G):
(4)      """Returns a branching representing all (overlapping) paths from
(4)      root nodes to leaf nodes in the given directed acyclic graph.
(4)      As described in :mod:`networkx.algorithms.tree.recognition`, a
(4)      *branching* is a directed forest in which each node has at most one
(4)      parent. In other words, a branching is a disjoint union of
(4)      *arborescences*. For this function, each node of in-degree zero in
(4)      `G` becomes a root of one of the arborescences, and there will be
(4)      one leaf node for each distinct path from that root to a leaf node
(4)      in `G`.
(4)      Each node `v` in `G` with *k* parents becomes *k* distinct nodes in
(4)      the returned branching, one for each parent, and the sub-DAG rooted
(4)      at `v` is duplicated for each copy. The algorithm then recurses on
(4)      the children of each copy of `v`.
(4)      Parameters
(4)      ----------
(4)      G : NetworkX graph
(8)          A directed acyclic graph.
(4)      Returns
(4)      -------
(4)      DiGraph
(8)          The branching in which there is a bijection between root-to-leaf
(8)          paths in `G` (in which multiple paths may share the same leaf)
(8)          and root-to-leaf paths in the branching (in which there is a
(8)          unique path from a root to a leaf).
(8)          Each node has an attribute 'source' whose value is the original
(8)          node to which this node corresponds. No other graph, node, or
(8)          edge attributes are copied into this new graph.
(4)      Raises
(4)      ------
(4)      NetworkXNotImplemented
(8)          If `G` is not directed, or if `G` is a multigraph.
(4)      HasACycle
(8)          If `G` is not acyclic.
(4)      Examples
(4)      --------
(4)      To examine which nodes in the returned branching were produced by
(4)      which original node in the directed acyclic graph, we can collect
(4)      the mapping from source node to new nodes into a dictionary. For
(4)      example, consider the directed diamond graph::
(8)          >>> from collections import defaultdict
(8)          >>> from operator import itemgetter
(8)          >>>
(8)          >>> G = nx.DiGraph(nx.utils.pairwise("abd"))
(8)          >>> G.add_edges_from(nx.utils.pairwise("acd"))
(8)          >>> B = nx.dag_to_branching(G)
(8)          >>>
(8)          >>> sources = defaultdict(set)
(8)          >>> for v, source in B.nodes(data="source"):
(8)          ...     sources[source].add(v)
(8)          >>> len(sources["a"])
(8)          1
(8)          >>> len(sources["d"])
(8)          2
(4)      To copy node attributes from the original graph to the new graph,
(4)      you can use a dictionary like the one constructed in the above
(4)      example::
(8)          >>> for source, nodes in sources.items():
(8)          ...     for v in nodes:
(8)          ...         B.nodes[v].update(G.nodes[source])
(4)      Notes
(4)      -----
(4)      This function is not idempotent in the sense that the node labels in
(4)      the returned branching may be uniquely generated each time the
(4)      function is invoked. In fact, the node labels may not be integers;
(4)      in order to relabel the nodes to be more readable, you can use the
(4)      :func:`networkx.convert_node_labels_to_integers` function.
(4)      The current implementation of this function uses
(4)      :func:`networkx.prefix_tree`, so it is subject to the limitations of
```

```
934: (4)                that function.
935: (4)                """
936: (4)                if has_cycle(G):
937: (8)                    msg = "dag_to_branching is only defined for acyclic graphs"
938: (8)                    raise nx.HasACycle(msg)
939: (4)                paths = root_to_leaf_paths(G)
940: (4)                B = nx.prefix_tree(paths)
941: (4)                B.remove_node(0)
942: (4)                B.remove_node(-1)
943: (4)                return B
944: (0)            @not_implemented_for("undirected")
945: (0)            @nx._dispatchable
946: (0)            def compute_v_structures(G):
947: (4)                """Iterate through the graph to compute all v-structures.
948: (4)                V-structures are triples in the directed graph where
949: (4)                two parent nodes point to the same child and the two parent nodes
950: (4)                are not adjacent.
951: (4)                Parameters
952: (4)                ----------
953: (4)                G : graph
954: (8)                    A networkx DiGraph.
955: (4)                Returns
956: (4)                -------
957: (4)                vstructs : iterator of tuples
958: (8)                    The v structures within the graph. Each v structure is a 3-tuple with
the
959: (8)                    parent, collider, and other parent.
960: (4)                Examples
961: (4)                --------
962: (4)                >>> G = nx.DiGraph()
963: (4)                >>> G.add_edges_from([(1, 2), (0, 5), (3, 1), (2, 4), (3, 1), (4, 5), (1,
5)])
964: (4)                >>> sorted(nx.compute_v_structures(G))
965: (4)                [(0, 5, 1), (0, 5, 4), (1, 5, 4)]
966: (4)                Notes
967: (4)                -----
968: (4)                `Wikipedia: Collider in causal graphs
<https://en.wikipedia.org/wiki/Collider_(statistics)>`_
969: (4)                """
970: (4)                for collider, preds in G.pred.items():
971: (8)                    for common_parents in combinations(preds, r=2):
972: (12)                       common_parents = sorted(common_parents)
973: (12)                       yield (common_parents[0], collider, common_parents[1])
```

File 24 - distance_measures.py:

```
1: (0)              """Graph diameter, radius, eccentricity and other properties."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for
4: (0)              __all__ = [
5: (4)                  "eccentricity",
6: (4)                  "diameter",
7: (4)                  "radius",
8: (4)                  "periphery",
9: (4)                  "center",
10: (4)                 "barycenter",
11: (4)                 "resistance_distance",
12: (4)                 "kemeny_constant",
13: (4)                 "effective_graph_resistance",
14: (0)             ]
15: (0)             def _extrema_bounding(G, compute="diameter", weight=None):
16: (4)                 """Compute requested extreme distance metric of undirected graph G
17: (4)                 Computation is based on smart lower and upper bounds, and in practice
18: (4)                 linear in the number of nodes, rather than quadratic (except for some
19: (4)                 border cases such as complete graphs or circle shaped graphs).
20: (4)                 Parameters
21: (4)                 ----------
```

```
22: (4)                 G : NetworkX graph
23: (7)                    An undirected graph
24: (4)                 compute : string denoting the requesting metric
25: (7)                    "diameter" for the maximal eccentricity value,
26: (7)                    "radius" for the minimal eccentricity value,
27: (7)                    "periphery" for the set of nodes with eccentricity equal to the
diameter,
28: (7)                    "center" for the set of nodes with eccentricity equal to the radius,
29: (7)                    "eccentricities" for the maximum distance from each node to all other
nodes in G
30: (4)                 weight : string, function, or None
31: (8)                     If this is a string, then edge weights will be accessed via the
32: (8)                     edge attribute with this key (that is, the weight of the edge
33: (8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
34: (8)                     such edge attribute exists, the weight of the edge is assumed to
35: (8)                     be one.
36: (8)                     If this is a function, the weight of an edge is the value
37: (8)                     returned by the function. The function must accept exactly three
38: (8)                     positional arguments: the two endpoints of an edge and the
39: (8)                     dictionary of edge attributes for that edge. The function must
40: (8)                     return a number.
41: (8)                     If this is None, every edge has weight/distance/cost 1.
42: (8)                     Weights stored as floating point values can lead to small round-off
43: (8)                     errors in distances. Use integer weights to avoid this.
44: (8)                     Weights should be positive, since they are distances.
45: (4)                 Returns
46: (4)                 -------
47: (4)                 value : value of the requested metric
48: (7)                    int for "diameter" and "radius" or
49: (7)                    list of nodes for "center" and "periphery" or
50: (7)                    dictionary of eccentricity values keyed by node for "eccentricities"
51: (4)                 Raises
52: (4)                 ------
53: (4)                 NetworkXError
54: (8)                     If the graph consists of multiple components
55: (4)                 ValueError
56: (8)                     If `compute` is not one of "diameter", "radius", "periphery",
"center", or "eccentricities".
57: (4)                 Notes
58: (4)                 -----
59: (4)                 This algorithm was proposed in [1]_ and discussed further in [2]_ and
[3]_.
60: (4)                 References
61: (4)                 ----------
62: (4)                 .. [1] F. W. Takes, W. A. Kosters,
63: (7)                    "Determining the diameter of small world networks."
64: (7)                    Proceedings of the 20th ACM international conference on Information and
knowledge management, 2011
65: (7)                    https://dl.acm.org/doi/abs/10.1145/2063576.2063748
66: (4)                 .. [2] F. W. Takes, W. A. Kosters,
67: (7)                    "Computing the Eccentricity Distribution of Large Graphs."
68: (7)                    Algorithms, 2013
69: (7)                    https://www.mdpi.com/1999-4893/6/1/100
70: (4)                 .. [3] M. Borassi, P. Crescenzi, M. Habib, W. A. Kosters, A. Marino, F. W.
Takes,
71: (7)                    "Fast diameter and radius BFS-based computation in (weakly connected)
real-world graphs: With an application to the six degrees of separation games. "
72: (7)                    Theoretical Computer Science, 2015
73: (7)                    https://www.sciencedirect.com/science/article/pii/S0304397515001644
74: (4)                 """
75: (4)                 degrees = dict(G.degree())  # start with the highest degree node
76: (4)                 minlowernode = max(degrees, key=degrees.get)
77: (4)                 N = len(degrees)  # number of nodes
78: (4)                 high = False
79: (4)                 ecc_lower = dict.fromkeys(G, 0)
80: (4)                 ecc_upper = dict.fromkeys(G, N)
81: (4)                 candidates = set(G)
82: (4)                 minlower = N
83: (4)                 maxlower = 0
```

```
 84: (4)                 minupper = N
 85: (4)                 maxupper = 0
 86: (4)                 while candidates:
 87: (8)                     if high:
 88: (12)                        current = maxuppernode  # select node with largest upper bound
 89: (8)                     else:
 90: (12)                        current = minlowernode  # select node with smallest lower bound
 91: (8)                     high = not high
 92: (8)                     dist = nx.shortest_path_length(G, source=current, weight=weight)
 93: (8)                     if len(dist) != N:
 94: (12)                        msg = "Cannot compute metric because graph is not connected."
 95: (12)                        raise nx.NetworkXError(msg)
 96: (8)                     current_ecc = max(dist.values())
 97: (8)                     maxuppernode = None
 98: (8)                     minlowernode = None
 99: (8)                     for i in candidates:
 100: (12)                       d = dist[i]
 101: (12)                       ecc_lower[i] = low = max(ecc_lower[i], max(d, (current_ecc - d)))
 102: (12)                       ecc_upper[i] = upp = min(ecc_upper[i], current_ecc + d)
 103: (12)                       minlower = min(ecc_lower[i], minlower)
 104: (12)                       maxlower = max(ecc_lower[i], maxlower)
 105: (12)                       minupper = min(ecc_upper[i], minupper)
 106: (12)                       maxupper = max(ecc_upper[i], maxupper)
 107: (8)                    if compute == "diameter":
 108: (12)                       ruled_out = {
 109: (16)                           i
 110: (16)                           for i in candidates
 111: (16)                           if ecc_upper[i] <= maxlower and 2 * ecc_lower[i] >= maxupper
 112: (12)                       }
 113: (8)                    elif compute == "radius":
 114: (12)                       ruled_out = {
 115: (16)                           i
 116: (16)                           for i in candidates
 117: (16)                           if ecc_lower[i] >= minupper and ecc_upper[i] + 1 <= 2 *
 minlower
 118: (12)                       }
 119: (8)                    elif compute == "periphery":
 120: (12)                       ruled_out = {
 121: (16)                           i
 122: (16)                           for i in candidates
 123: (16)                           if ecc_upper[i] < maxlower
 124: (16)                           and (maxlower == maxupper or ecc_lower[i] > maxupper)
 125: (12)                       }
 126: (8)                    elif compute == "center":
 127: (12)                       ruled_out = {
 128: (16)                           i
 129: (16)                           for i in candidates
 130: (16)                           if ecc_lower[i] > minupper
 131: (16)                           and (minlower == minupper or ecc_upper[i] + 1 < 2 * minlower)
 132: (12)                       }
 133: (8)                    elif compute == "eccentricities":
 134: (12)                       ruled_out = set()
 135: (8)                    else:
 136: (12)                       msg = "compute must be one of 'diameter', 'radius', 'periphery',
 'center', 'eccentricities'"
 137: (12)                       raise ValueError(msg)
 138: (8)                    ruled_out.update(i for i in candidates if ecc_lower[i] ==
 ecc_upper[i])
 139: (8)                    candidates -= ruled_out
 140: (8)                    for i in candidates:
 141: (12)                       if (
 142: (16)                           minlowernode is None
 143: (16)                           or (
 144: (20)                               ecc_lower[i] == ecc_lower[minlowernode]
 145: (20)                               and degrees[i] > degrees[minlowernode]
 146: (16)                           )
 147: (16)                           or (ecc_lower[i] < ecc_lower[minlowernode])
 148: (12)                       ):
 149: (16)                           minlowernode = i
```

```
150: (12)                        if (
151: (16)                            maxuppernode is None
152: (16)                            or (
153: (20)                                ecc_upper[i] == ecc_upper[maxuppernode]
154: (20)                                and degrees[i] > degrees[maxuppernode]
155: (16)                            )
156: (16)                            or (ecc_upper[i] > ecc_upper[maxuppernode])
157: (12)                        ):
158: (16)                            maxuppernode = i
159: (4)                if compute == "diameter":
160: (8)                    return maxlower
161: (4)                if compute == "radius":
162: (8)                    return minupper
163: (4)                if compute == "periphery":
164: (8)                    p = [v for v in G if ecc_lower[v] == maxlower]
165: (8)                    return p
166: (4)                if compute == "center":
167: (8)                    c = [v for v in G if ecc_upper[v] == minupper]
168: (8)                    return c
169: (4)                if compute == "eccentricities":
170: (8)                    return ecc_lower
171: (4)                return None
172: (0)            @nx._dispatchable(edge_attrs="weight")
173: (0)            def eccentricity(G, v=None, sp=None, weight=None):
174: (4)                """Returns the eccentricity of nodes in G.
175: (4)                The eccentricity of a node v is the maximum distance from v to
176: (4)                all other nodes in G.
177: (4)                Parameters
178: (4)                ----------
179: (4)                G : NetworkX graph
180: (7)                   A graph
181: (4)                v : node, optional
182: (7)                   Return value of specified node
183: (4)                sp : dict of dicts, optional
184: (7)                   All pairs shortest path lengths as a dictionary of dictionaries
185: (4)                weight : string, function, or None (default=None)
186: (8)                    If this is a string, then edge weights will be accessed via the
187: (8)                    edge attribute with this key (that is, the weight of the edge
188: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
189: (8)                    such edge attribute exists, the weight of the edge is assumed to
190: (8)                    be one.
191: (8)                    If this is a function, the weight of an edge is the value
192: (8)                    returned by the function. The function must accept exactly three
193: (8)                    positional arguments: the two endpoints of an edge and the
194: (8)                    dictionary of edge attributes for that edge. The function must
195: (8)                    return a number.
196: (8)                    If this is None, every edge has weight/distance/cost 1.
197: (8)                    Weights stored as floating point values can lead to small round-off
198: (8)                    errors in distances. Use integer weights to avoid this.
199: (8)                    Weights should be positive, since they are distances.
200: (4)                Returns
201: (4)                -------
202: (4)                ecc : dictionary
203: (7)                   A dictionary of eccentricity values keyed by node.
204: (4)                Examples
205: (4)                --------
206: (4)                >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
207: (4)                >>> dict(nx.eccentricity(G))
208: (4)                {1: 2, 2: 3, 3: 2, 4: 2, 5: 3}
209: (4)                >>> dict(nx.eccentricity(G, v=[1, 5]))  # This returns the eccentricity of
node 1 & 5
210: (4)                {1: 2, 5: 3}
211: (4)                """
212: (4)                order = G.order()
213: (4)                e = {}
214: (4)                for n in G.nbunch_iter(v):
215: (8)                    if sp is None:
216: (12)                       length = nx.shortest_path_length(G, source=n, weight=weight)
217: (12)                       L = len(length)
```

```
218: (8)                    else:
219: (12)                       try:
220: (16)                           length = sp[n]
221: (16)                           L = len(length)
222: (12)                       except TypeError as err:
223: (16)                           raise nx.NetworkXError('Format of "sp" is invalid.') from err
224: (8)                    if L != order:
225: (12)                       if G.is_directed():
226: (16)                           msg = (
227: (20)                               "Found infinite path length because the digraph is not"
228: (20)                               " strongly connected"
229: (16)                           )
230: (12)                       else:
231: (16)                           msg = "Found infinite path length because the graph is not" "
connected"
232: (12)                       raise nx.NetworkXError(msg)
233: (8)                    e[n] = max(length.values())
234: (4)                if v in G:
235: (8)                    return e[v]  # return single value
236: (4)                return e
237: (0)            @nx._dispatchable(edge_attrs="weight")
238: (0)            def diameter(G, e=None, usebounds=False, weight=None):
239: (4)                """Returns the diameter of the graph G.
240: (4)                The diameter is the maximum eccentricity.
241: (4)                Parameters
242: (4)                ----------
243: (4)                G : NetworkX graph
244: (7)                   A graph
245: (4)                e : eccentricity dictionary, optional
246: (6)                  A precomputed dictionary of eccentricities.
247: (4)                weight : string, function, or None
248: (8)                    If this is a string, then edge weights will be accessed via the
249: (8)                    edge attribute with this key (that is, the weight of the edge
250: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
251: (8)                    such edge attribute exists, the weight of the edge is assumed to
252: (8)                    be one.
253: (8)                    If this is a function, the weight of an edge is the value
254: (8)                    returned by the function. The function must accept exactly three
255: (8)                    positional arguments: the two endpoints of an edge and the
256: (8)                    dictionary of edge attributes for that edge. The function must
257: (8)                    return a number.
258: (8)                    If this is None, every edge has weight/distance/cost 1.
259: (8)                    Weights stored as floating point values can lead to small round-off
260: (8)                    errors in distances. Use integer weights to avoid this.
261: (8)                    Weights should be positive, since they are distances.
262: (4)                Returns
263: (4)                -------
264: (4)                d : integer
265: (7)                   Diameter of graph
266: (4)                Examples
267: (4)                --------
268: (4)                >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
269: (4)                >>> nx.diameter(G)
270: (4)                3
271: (4)                See Also
272: (4)                --------
273: (4)                eccentricity
274: (4)                """
275: (4)                if usebounds is True and e is None and not G.is_directed():
276: (8)                    return _extrema_bounding(G, compute="diameter", weight=weight)
277: (4)                if e is None:
278: (8)                    e = eccentricity(G, weight=weight)
279: (4)                return max(e.values())
280: (0)            @nx._dispatchable(edge_attrs="weight")
281: (0)            def periphery(G, e=None, usebounds=False, weight=None):
282: (4)                """Returns the periphery of the graph G.
283: (4)                The periphery is the set of nodes with eccentricity equal to the diameter.
284: (4)                Parameters
285: (4)                ----------
```

```
(4)                 G : NetworkX graph
(7)                    A graph
(4)                 e : eccentricity dictionary, optional
(6)                   A precomputed dictionary of eccentricities.
(4)                 weight : string, function, or None
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number.
(8)                     If this is None, every edge has weight/distance/cost 1.
(8)                     Weights stored as floating point values can lead to small round-off
(8)                     errors in distances. Use integer weights to avoid this.
(8)                     Weights should be positive, since they are distances.
(4)                 Returns
(4)                 -------
(4)                 p : list
(7)                    List of nodes in periphery
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
(4)                 >>> nx.periphery(G)
(4)                 [2, 5]
(4)                 See Also
(4)                 --------
(4)                 barycenter
(4)                 center
(4)                 """
(4)                 if usebounds is True and e is None and not G.is_directed():
(8)                     return _extrema_bounding(G, compute="periphery", weight=weight)
(4)                 if e is None:
(8)                     e = eccentricity(G, weight=weight)
(4)                 diameter = max(e.values())
(4)                 p = [v for v in e if e[v] == diameter]
(4)                 return p
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def radius(G, e=None, usebounds=False, weight=None):
(4)                 """Returns the radius of the graph G.
(4)                 The radius is the minimum eccentricity.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    A graph
(4)                 e : eccentricity dictionary, optional
(6)                   A precomputed dictionary of eccentricities.
(4)                 weight : string, function, or None
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number.
(8)                     If this is None, every edge has weight/distance/cost 1.
(8)                     Weights stored as floating point values can lead to small round-off
(8)                     errors in distances. Use integer weights to avoid this.
(8)                     Weights should be positive, since they are distances.
(4)                 Returns
(4)                 -------
(4)                 r : integer
(7)                    Radius of graph
```

```
355: (4)                Examples
356: (4)                --------
357: (4)                >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
358: (4)                >>> nx.radius(G)
359: (4)                2
360: (4)                """
361: (4)                if usebounds is True and e is None and not G.is_directed():
362: (8)                    return _extrema_bounding(G, compute="radius", weight=weight)
363: (4)                if e is None:
364: (8)                    e = eccentricity(G, weight=weight)
365: (4)                return min(e.values())
366: (0)            @nx._dispatchable(edge_attrs="weight")
367: (0)            def center(G, e=None, usebounds=False, weight=None):
368: (4)                """Returns the center of the graph G.
369: (4)                The center is the set of nodes with eccentricity equal to radius.
370: (4)                Parameters
371: (4)                ----------
372: (4)                G : NetworkX graph
373: (7)                   A graph
374: (4)                e : eccentricity dictionary, optional
375: (6)                  A precomputed dictionary of eccentricities.
376: (4)                weight : string, function, or None
377: (8)                    If this is a string, then edge weights will be accessed via the
378: (8)                    edge attribute with this key (that is, the weight of the edge
379: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
380: (8)                    such edge attribute exists, the weight of the edge is assumed to
381: (8)                    be one.
382: (8)                    If this is a function, the weight of an edge is the value
383: (8)                    returned by the function. The function must accept exactly three
384: (8)                    positional arguments: the two endpoints of an edge and the
385: (8)                    dictionary of edge attributes for that edge. The function must
386: (8)                    return a number.
387: (8)                    If this is None, every edge has weight/distance/cost 1.
388: (8)                    Weights stored as floating point values can lead to small round-off
389: (8)                    errors in distances. Use integer weights to avoid this.
390: (8)                    Weights should be positive, since they are distances.
391: (4)                Returns
392: (4)                -------
393: (4)                c : list
394: (7)                   List of nodes in center
395: (4)                Examples
396: (4)                --------
397: (4)                >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
398: (4)                >>> list(nx.center(G))
399: (4)                [1, 3, 4]
400: (4)                See Also
401: (4)                --------
402: (4)                barycenter
403: (4)                periphery
404: (4)                """
405: (4)                if usebounds is True and e is None and not G.is_directed():
406: (8)                    return _extrema_bounding(G, compute="center", weight=weight)
407: (4)                if e is None:
408: (8)                    e = eccentricity(G, weight=weight)
409: (4)                radius = min(e.values())
410: (4)                p = [v for v in e if e[v] == radius]
411: (4)                return p
412: (0)            @nx._dispatchable(edge_attrs="weight", mutates_input={"attr": 2})
413: (0)            def barycenter(G, weight=None, attr=None, sp=None):
414: (4)                r"""Calculate barycenter of a connected graph, optionally with edge
weights.
415: (4)                The :dfn:`barycenter` a
416: (4)                :func:`connected <networkx.algorithms.components.is_connected>` graph
417: (4)                :math:`G` is the subgraph induced by the set of its nodes :math:`v`
418: (4)                minimizing the objective function
419: (4)                .. math::
420: (8)                    \sum_{u \in V(G)} d_G(u, v),
421: (4)                where :math:`d_G` is the (possibly weighted) :func:`path length
422: (4)                <networkx.algorithms.shortest_paths.generic.shortest_path_length>`.
```

```
    The barycenter is also called the :dfn:`median`. See [West01]_, p. 78.
    Parameters
    ----------
    G : :class:`networkx.Graph`
        The connected graph :math:`G`.
    weight : :class:`str`, optional
        Passed through to
:func:`~networkx.algorithms.shortest_paths.generic.shortest_path_length`.
    attr : :class:`str`, optional
        If given, write the value of the objective function to each node's
        `attr` attribute. Otherwise do not store the value.
    sp : dict of dicts, optional
       All pairs shortest path lengths as a dictionary of dictionaries
    Returns
    -------
    list
        Nodes of `G` that induce the barycenter of `G`.
    Raises
    ------
    NetworkXNoPath
        If `G` is disconnected. `G` may appear disconnected to
        :func:`barycenter` if `sp` is given but is missing shortest path
        lengths for any pairs.
    ValueError
        If `sp` and `weight` are both given.
    Examples
    --------
    >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
    >>> nx.barycenter(G)
    [1, 3, 4]
    See Also
    --------
    center
    periphery
    """
    if sp is None:
        sp = nx.shortest_path_length(G, weight=weight)
    else:
        sp = sp.items()
        if weight is not None:
            raise ValueError("Cannot use both sp, weight arguments together")
    smallest, barycenter_vertices, n = float("inf"), [], len(G)
    for v, dists in sp:
        if len(dists) < n:
            raise nx.NetworkXNoPath(
                f"Input graph {G} is disconnected, so every induced subgraph "
                "has infinite barycentricity."
            )
        barycentricity = sum(dists.values())
        if attr is not None:
            G.nodes[v][attr] = barycentricity
        if barycentricity < smallest:
            smallest = barycentricity
            barycenter_vertices = [v]
        elif barycentricity == smallest:
            barycenter_vertices.append(v)
    if attr is not None:
        nx._clear_cache(G)
    return barycenter_vertices
@not_implemented_for("directed")
@nx._dispatchable(edge_attrs="weight")
def resistance_distance(G, nodeA=None, nodeB=None, weight=None,
invert_weight=True):
    """Returns the resistance distance between pairs of nodes in graph G.
    The resistance distance between two nodes of a graph is akin to treating
    the graph as a grid of resistors with a resistance equal to the provided
    weight [1]_, [2]_.
    If weight is not provided, then a weight of 1 is used for all edges.
```

```
    If two nodes are the same, the resistance distance is zero.
    Parameters
    ----------
    G : NetworkX graph
       A graph
    nodeA : node or None, optional (default=None)
      A node within graph G.
      If None, compute resistance distance using all nodes as source nodes.
    nodeB : node or None, optional (default=None)
      A node within graph G.
      If None, compute resistance distance using all nodes as target nodes.
    weight : string or None, optional (default=None)
       The edge data key used to compute the resistance distance.
       If None, then each edge has weight 1.
    invert_weight : boolean (default=True)
        Proper calculation of resistance distance requires building the
        Laplacian matrix with the reciprocal of the weight. Not required
        if the weight is already inverted. Weight cannot be zero.
    Returns
    -------
    rd : dict or float
       If `nodeA` and `nodeB` are given, resistance distance between `nodeA`
       and `nodeB`. If `nodeA` or `nodeB` is unspecified (the default), a
       dictionary of nodes with resistance distances as the value.
    Raises
    ------
    NetworkXNotImplemented
        If `G` is a directed graph.
    NetworkXError
        If `G` is not connected, or contains no nodes,
        or `nodeA` is not in `G` or `nodeB` is not in `G`.
    Examples
    --------
    >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
    >>> round(nx.resistance_distance(G, 1, 3), 10)
    0.625
    Notes
    -----
    The implementation is based on Theorem A in [2]_. Self-loops are ignored.
    Multi-edges are contracted in one edge with weight equal to the harmonic sum of the weights.
    References
    ----------
    .. [1] Wikipedia
       "Resistance distance."
       https://en.wikipedia.org/wiki/Resistance_distance
    .. [2] D. J. Klein and M. Randic.
        Resistance distance.
        J. of Math. Chem. 12:81-95, 1993.
    """
    import numpy as np
    if len(G) == 0:
        raise nx.NetworkXError("Graph G must contain at least one node.")
    if not nx.is_connected(G):
        raise nx.NetworkXError("Graph G must be strongly connected.")
    if nodeA is not None and nodeA not in G:
        raise nx.NetworkXError("Node A is not in graph G.")
    if nodeB is not None and nodeB not in G:
        raise nx.NetworkXError("Node B is not in graph G.")
    G = G.copy()
    node_list = list(G)
    if invert_weight and weight is not None:
        if G.is_multigraph():
            for u, v, k, d in G.edges(keys=True, data=True):
                d[weight] = 1 / d[weight]
        else:
            for u, v, d in G.edges(data=True):
                d[weight] = 1 / d[weight]
    L = nx.laplacian_matrix(G, weight=weight).todense()
```

```
(4)                 Linv = np.linalg.pinv(L, hermitian=True)
(4)                 if nodeA is not None and nodeB is not None:
(8)                     i = node_list.index(nodeA)
(8)                     j = node_list.index(nodeB)
(8)                     return Linv.item(i, i) + Linv.item(j, j) - Linv.item(i, j) -
Linv.item(j, i)
(4)                 elif nodeA is not None:
(8)                     i = node_list.index(nodeA)
(8)                     d = {}
(8)                     for n in G:
(12)                        j = node_list.index(n)
(12)                        d[n] = Linv.item(i, i) + Linv.item(j, j) - Linv.item(i, j) -
Linv.item(j, i)
(8)                     return d
(4)                 elif nodeB is not None:
(8)                     j = node_list.index(nodeB)
(8)                     d = {}
(8)                     for n in G:
(12)                        i = node_list.index(n)
(12)                        d[n] = Linv.item(i, i) + Linv.item(j, j) - Linv.item(i, j) -
Linv.item(j, i)
(8)                     return d
(4)                 else:
(8)                     d = {}
(8)                     for n in G:
(12)                        i = node_list.index(n)
(12)                        d[n] = {}
(12)                        for n2 in G:
(16)                            j = node_list.index(n2)
(16)                            d[n][n2] = (
(20)                                Linv.item(i, i)
(20)                                + Linv.item(j, j)
(20)                                - Linv.item(i, j)
(20)                                - Linv.item(j, i)
(16)                            )
(8)                     return d
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def effective_graph_resistance(G, weight=None, invert_weight=True):
(4)                 """Returns the Effective graph resistance of G.
(4)                 Also known as the Kirchhoff index.
(4)                 The effective graph resistance is defined as the sum
(4)                 of the resistance distance of every node pair in G [1]_.
(4)                 If weight is not provided, then a weight of 1 is used for all edges.
(4)                 The effective graph resistance of a disconnected graph is infinite.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    A graph
(4)                 weight : string or None, optional (default=None)
(7)                    The edge data key used to compute the effective graph resistance.
(7)                    If None, then each edge has weight 1.
(4)                 invert_weight : boolean (default=True)
(8)                     Proper calculation of resistance distance requires building the
(8)                     Laplacian matrix with the reciprocal of the weight. Not required
(8)                     if the weight is already inverted. Weight cannot be zero.
(4)                 Returns
(4)                 -------
(4)                 RG : float
(8)                     The effective graph resistance of `G`.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is a directed graph.
(4)                 NetworkXError
(8)                     If `G` does not contain any nodes.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
```

```
    >>> round(nx.effective_graph_resistance(G), 10)
    10.25
    Notes
    -----
    The implementation is based on Theorem 2.2 in [2]_. Self-loops are ignored.
    Multi-edges are contracted in one edge with weight equal to the harmonic sum of the weights.
    References
    ----------
    .. [1] Wolfram
       "Kirchhoff Index."
       https://mathworld.wolfram.com/KirchhoffIndex.html
    .. [2] W. Ellens, F. M. Spieksma, P. Van Mieghem, A. Jamakovic, R. E. Kooij.
        Effective graph resistance.
        Lin. Alg. Appl. 435:2491-2506, 2011.
    """
    import numpy as np
    if len(G) == 0:
        raise nx.NetworkXError("Graph G must contain at least one node.")
    if not nx.is_connected(G):
        return float("inf")
    G = G.copy()
    if invert_weight and weight is not None:
        if G.is_multigraph():
            for u, v, k, d in G.edges(keys=True, data=True):
                d[weight] = 1 / d[weight]
        else:
            for u, v, d in G.edges(data=True):
                d[weight] = 1 / d[weight]
    mu = np.sort(nx.laplacian_spectrum(G, weight=weight))
    return float(np.sum(1 / mu[1:]) * G.number_of_nodes())
@nx.utils.not_implemented_for("directed")
@nx._dispatchable(edge_attrs="weight")
def kemeny_constant(G, *, weight=None):
    """Returns the Kemeny constant of the given graph.
    The *Kemeny constant* (or Kemeny's constant) of a graph `G`
    can be computed by regarding the graph as a Markov chain.
    The Kemeny constant is then the expected number of time steps
    to transition from a starting state i to a random destination state
    sampled from the Markov chain's stationary distribution.
    The Kemeny constant is independent of the chosen initial state [1]_.
    The Kemeny constant measures the time needed for spreading
    across a graph. Low values indicate a closely connected graph
    whereas high values indicate a spread-out graph.
    If weight is not provided, then a weight of 1 is used for all edges.
    Since `G` represents a Markov chain, the weights must be positive.
    Parameters
    ----------
    G : NetworkX graph
    weight : string or None, optional (default=None)
       The edge data key used to compute the Kemeny constant.
       If None, then each edge has weight 1.
    Returns
    -------
    float
        The Kemeny constant of the graph `G`.
    Raises
    ------
    NetworkXNotImplemented
        If the graph `G` is directed.
    NetworkXError
        If the graph `G` is not connected, or contains no nodes,
        or has edges with negative weights.
    Examples
    --------
    >>> G = nx.complete_graph(5)
    >>> round(nx.kemeny_constant(G), 10)
```

```
690: (4)                3.2
691: (4)                Notes
692: (4)                -----
693: (4)                The implementation is based on equation (3.3) in [2]_.
694: (4)                Self-loops are allowed and indicate a Markov chain where
695: (4)                the state can remain the same. Multi-edges are contracted
696: (4)                in one edge with weight equal to the sum of the weights.
697: (4)                References
698: (4)                ----------
699: (4)                .. [1] Wikipedia
700: (7)                   "Kemeny's constant."
701: (7)                   https://en.wikipedia.org/wiki/Kemeny%27s_constant
702: (4)                .. [2] Lovász L.
703: (8)                    Random walks on graphs: A survey.
704: (8)                    Paul Erdös is Eighty, vol. 2, Bolyai Society,
705: (8)                    Mathematical Studies, Keszthely, Hungary (1993), pp. 1-46
706: (4)                """
707: (4)                import numpy as np
708: (4)                import scipy as sp
709: (4)                if len(G) == 0:
710: (8)                    raise nx.NetworkXError("Graph G must contain at least one node.")
711: (4)                if not nx.is_connected(G):
712: (8)                    raise nx.NetworkXError("Graph G must be connected.")
713: (4)                if nx.is_negatively_weighted(G, weight=weight):
714: (8)                    raise nx.NetworkXError("The weights of graph G must be nonnegative.")
715: (4)                A = nx.adjacency_matrix(G, weight=weight)
716: (4)                n, m = A.shape
717: (4)                diags = A.sum(axis=1)
718: (4)                with np.errstate(divide="ignore"):
719: (8)                    diags_sqrt = 1.0 / np.sqrt(diags)
720: (4)                diags_sqrt[np.isinf(diags_sqrt)] = 0
721: (4)                DH = sp.sparse.csr_array(sp.sparse.spdiags(diags_sqrt, 0, m, n,
format="csr"))
722: (4)                H = DH @ (A @ DH)
723: (4)                eig = np.sort(sp.linalg.eigvalsh(H.todense()))
724: (4)                return float(np.sum(1 / (1 - eig[:-1])))

----------------------------------------

File 25 - dominance.py:

1: (0)              """
2: (0)              Dominance algorithms.
3: (0)              """
4: (0)              from functools import reduce
5: (0)              import networkx as nx
6: (0)              from networkx.utils import not_implemented_for
7: (0)              __all__ = ["immediate_dominators", "dominance_frontiers"]
8: (0)              @not_implemented_for("undirected")
9: (0)              @nx._dispatchable
10: (0)             def immediate_dominators(G, start):
11: (4)                 """Returns the immediate dominators of all nodes of a directed graph.
12: (4)                 Parameters
13: (4)                 ----------
14: (4)                 G : a DiGraph or MultiDiGraph
15: (8)                     The graph where dominance is to be computed.
16: (4)                 start : node
17: (8)                     The start node of dominance computation.
18: (4)                 Returns
19: (4)                 -------
20: (4)                 idom : dict keyed by nodes
21: (8)                     A dict containing the immediate dominators of each node reachable from
22: (8)                     `start`.
23: (4)                 Raises
24: (4)                 ------
25: (4)                 NetworkXNotImplemented
26: (8)                     If `G` is undirected.
27: (4)                 NetworkXError
28: (8)                     If `start` is not in `G`.
```

```
    Notes
    -----
    Except for `start`, the immediate dominators are the parents of their
    corresponding nodes in the dominator tree.
    Examples
    --------
    >>> G = nx.DiGraph([(1, 2), (1, 3), (2, 5), (3, 4), (4, 5)])
    >>> sorted(nx.immediate_dominators(G, 1).items())
    [(1, 1), (2, 1), (3, 1), (4, 3), (5, 1)]
    References
    ----------
    .. [1] K. D. Cooper, T. J. Harvey, and K. Kennedy.
           A simple, fast dominance algorithm.
           Software Practice & Experience, 4:110, 2001.
    """
    if start not in G:
        raise nx.NetworkXError("start is not in G")
    idom = {start: start}
    order = list(nx.dfs_postorder_nodes(G, start))
    dfn = {u: i for i, u in enumerate(order)}
    order.pop()
    order.reverse()
    def intersect(u, v):
        while u != v:
            while dfn[u] < dfn[v]:
                u = idom[u]
            while dfn[u] > dfn[v]:
                v = idom[v]
        return u
    changed = True
    while changed:
        changed = False
        for u in order:
            new_idom = reduce(intersect, (v for v in G.pred[u] if v in idom))
            if u not in idom or idom[u] != new_idom:
                idom[u] = new_idom
                changed = True
    return idom
@nx._dispatchable
def dominance_frontiers(G, start):
    """Returns the dominance frontiers of all nodes of a directed graph.
    Parameters
    ----------
    G : a DiGraph or MultiDiGraph
        The graph where dominance is to be computed.
    start : node
        The start node of dominance computation.
    Returns
    -------
    df : dict keyed by nodes
        A dict containing the dominance frontiers of each node reachable from
        `start` as lists.
    Raises
    ------
    NetworkXNotImplemented
        If `G` is undirected.
    NetworkXError
        If `start` is not in `G`.
    Examples
    --------
    >>> G = nx.DiGraph([(1, 2), (1, 3), (2, 5), (3, 4), (4, 5)])
    >>> sorted((u, sorted(df)) for u, df in nx.dominance_frontiers(G,
1).items())
    [(1, []), (2, [5]), (3, [5]), (4, [5]), (5, [])]
    References
    ----------
    .. [1] K. D. Cooper, T. J. Harvey, and K. Kennedy.
           A simple, fast dominance algorithm.
           Software Practice & Experience, 4:110, 2001.
```

```
97: (4)                 """
98: (4)                 idom = nx.immediate_dominators(G, start)
99: (4)                 df = {u: set() for u in idom}
100: (4)                for u in idom:
101: (8)                    if len(G.pred[u]) >= 2:
102: (12)                       for v in G.pred[u]:
103: (16)                           if v in idom:
104: (20)                               while v != idom[u]:
105: (24)                                   df[v].add(u)
106: (24)                                   v = idom[v]
107: (4)                return df
```

File 26 - distance_regular.py:

```
1: (0)              """
2: (0)              =======================
3: (0)              Distance-regular graphs
4: (0)              =======================
5: (0)              """
6: (0)              import networkx as nx
7: (0)              from networkx.utils import not_implemented_for
8: (0)              from .distance_measures import diameter
9: (0)              __all__ = [
10: (4)                 "is_distance_regular",
11: (4)                 "is_strongly_regular",
12: (4)                 "intersection_array",
13: (4)                 "global_parameters",
14: (0)             ]
15: (0)             @nx._dispatchable
16: (0)             def is_distance_regular(G):
17: (4)                 """Returns True if the graph is distance regular, False otherwise.
18: (4)                 A connected graph G is distance-regular if for any nodes x,y
19: (4)                 and any integers i,j=0,1,...,d (where d is the graph
20: (4)                 diameter), the number of vertices at distance i from x and
21: (4)                 distance j from y depends only on i,j and the graph distance
22: (4)                 between x and y, independently of the choice of x and y.
23: (4)                 Parameters
24: (4)                 ----------
25: (4)                 G: Networkx graph (undirected)
26: (4)                 Returns
27: (4)                 -------
28: (4)                 bool
29: (6)                   True if the graph is Distance Regular, False otherwise
30: (4)                 Examples
31: (4)                 --------
32: (4)                 >>> G = nx.hypercube_graph(6)
33: (4)                 >>> nx.is_distance_regular(G)
34: (4)                 True
35: (4)                 See Also
36: (4)                 --------
37: (4)                 intersection_array, global_parameters
38: (4)                 Notes
39: (4)                 -----
40: (4)                 For undirected and simple graphs only
41: (4)                 References
42: (4)                 ----------
43: (4)                 .. [1] Brouwer, A. E.; Cohen, A. M.; and Neumaier, A.
44: (8)                     Distance-Regular Graphs. New York: Springer-Verlag, 1989.
45: (4)                 .. [2] Weisstein, Eric W. "Distance-Regular Graph."
46: (8)                     http://mathworld.wolfram.com/Distance-RegularGraph.html
47: (4)                 """
48: (4)                 try:
49: (8)                     intersection_array(G)
50: (8)                     return True
51: (4)                 except nx.NetworkXError:
52: (8)                     return False
53: (0)             def global_parameters(b, c):
```

```
(4)                 """Returns global parameters for a given intersection array.
(4)                 Given a distance-regular graph G with integers b_i, c_i,i = 0,....,d
(4)                 such that for any 2 vertices x,y in G at a distance i=d(x,y), there
(4)                 are exactly c_i neighbors of y at a distance of i-1 from x and b_i
(4)                 neighbors of y at a distance of i+1 from x.
(4)                 Thus, a distance regular graph has the global parameters,
(4)                 [[c_0,a_0,b_0],[c_1,a_1,b_1],......,[c_d,a_d,b_d]] for the
(4)                 intersection array  [b_0,b_1,.....b_{d-1};c_1,c_2,.....c_d]
(4)                 where a_i+b_i+c_i=k , k= degree of every vertex.
(4)                 Parameters
(4)                 ----------
(4)                 b : list
(4)                 c : list
(4)                 Returns
(4)                 -------
(4)                 iterable
(7)                    An iterable over three tuples.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.dodecahedral_graph()
(4)                 >>> b, c = nx.intersection_array(G)
(4)                 >>> list(nx.global_parameters(b, c))
(4)                 [(0, 0, 3), (1, 0, 2), (1, 1, 1), (1, 1, 1), (2, 0, 1), (3, 0, 0)]
(4)                 References
(4)                 ----------
(4)                 .. [1] Weisstein, Eric W. "Global Parameters."
(7)                    From MathWorld--A Wolfram Web Resource.
(7)                    http://mathworld.wolfram.com/GlobalParameters.html
(4)                 See Also
(4)                 --------
(4)                 intersection_array
(4)                 """
(4)                 return ((y, b[0] - x - y, x) for x, y in zip(b + [0], [0] + c))
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable
(0)             def intersection_array(G):
(4)                 """Returns the intersection array of a distance-regular graph.
(4)                 Given a distance-regular graph G with integers b_i, c_i,i = 0,....,d
(4)                 such that for any 2 vertices x,y in G at a distance i=d(x,y), there
(4)                 are exactly c_i neighbors of y at a distance of i-1 from x and b_i
(4)                 neighbors of y at a distance of i+1 from x.
(4)                 A distance regular graph's intersection array is given by,
(4)                 [b_0,b_1,.....b_{d-1};c_1,c_2,.....c_d]
(4)                 Parameters
(4)                 ----------
(4)                 G: Networkx graph (undirected)
(4)                 Returns
(4)                 -------
(4)                 b,c: tuple of lists
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.icosahedral_graph()
(4)                 >>> nx.intersection_array(G)
(4)                 ([5, 2, 1], [1, 2, 5])
(4)                 References
(4)                 ----------
(4)                 .. [1] Weisstein, Eric W. "Intersection Array."
(7)                    From MathWorld--A Wolfram Web Resource.
(7)                    http://mathworld.wolfram.com/IntersectionArray.html
(4)                 See Also
(4)                 --------
(4)                 global_parameters
(4)                 """
(4)                 if len(G) == 0:
(8)                     raise nx.NetworkXPointlessConcept("Graph has no nodes.")
(4)                 degree = iter(G.degree())
(4)                 (_, k) = next(degree)
(4)                 for _, knext in degree:
```

```
123: (8)                    if knext != k:
124: (12)                       raise nx.NetworkXError("Graph is not distance regular.")
125: (8)                    k = knext
126: (4)                path_length = dict(nx.all_pairs_shortest_path_length(G))
127: (4)                diameter = max(max(path_length[n].values()) for n in path_length)
128: (4)                bint = {}  # 'b' intersection array
129: (4)                cint = {}  # 'c' intersection array
130: (4)                for u in G:
131: (8)                    for v in G:
132: (12)                       try:
133: (16)                           i = path_length[u][v]
134: (12)                       except KeyError as err:  # graph must be connected
135: (16)                           raise nx.NetworkXError("Graph is not distance regular.") from
err
136: (12)                       c = len([n for n in G[v] if path_length[n][u] == i - 1])
137: (12)                       b = len([n for n in G[v] if path_length[n][u] == i + 1])
138: (12)                       if cint.get(i, c) != c or bint.get(i, b) != b:
139: (16)                           raise nx.NetworkXError("Graph is not distance regular")
140: (12)                       bint[i] = b
141: (12)                       cint[i] = c
142: (4)                return (
143: (8)                    [bint.get(j, 0) for j in range(diameter)],
144: (8)                    [cint.get(j + 1, 0) for j in range(diameter)],
145: (4)                )
146: (0)            @not_implemented_for("directed")
147: (0)            @not_implemented_for("multigraph")
148: (0)            @nx._dispatchable
149: (0)            def is_strongly_regular(G):
150: (4)                """Returns True if and only if the given graph is strongly
151: (4)                regular.
152: (4)                An undirected graph is *strongly regular* if
153: (4)                * it is regular,
154: (4)                * each pair of adjacent vertices has the same number of neighbors in
155: (6)                  common,
156: (4)                * each pair of nonadjacent vertices has the same number of neighbors
157: (6)                  in common.
158: (4)                Each strongly regular graph is a distance-regular graph.
159: (4)                Conversely, if a distance-regular graph has diameter two, then it is
160: (4)                a strongly regular graph. For more information on distance-regular
161: (4)                graphs, see :func:`is_distance_regular`.
162: (4)                Parameters
163: (4)                ----------
164: (4)                G : NetworkX graph
165: (8)                    An undirected graph.
166: (4)                Returns
167: (4)                -------
168: (4)                bool
169: (8)                    Whether `G` is strongly regular.
170: (4)                Examples
171: (4)                --------
172: (4)                The cycle graph on five vertices is strongly regular. It is
173: (4)                two-regular, each pair of adjacent vertices has no shared neighbors,
174: (4)                and each pair of nonadjacent vertices has one shared neighbor::
175: (8)                    >>> G = nx.cycle_graph(5)
176: (8)                    >>> nx.is_strongly_regular(G)
177: (8)                    True
178: (4)                """
179: (4)                return is_distance_regular(G) and diameter(G) == 2
```

File 27 - dominating.py:

```
1: (0)              """Functions for computing dominating sets in a graph."""
2: (0)              from itertools import chain
3: (0)              import networkx as nx
4: (0)              from networkx.utils import arbitrary_element
5: (0)              __all__ = ["dominating_set", "is_dominating_set"]
6: (0)              @nx._dispatchable
```

```
7: (0)              def dominating_set(G, start_with=None):
8: (4)                  r"""Finds a dominating set for the graph G.
9: (4)                  A *dominating set* for a graph with node set *V* is a subset *D* of
10: (4)                 *V* such that every node not in *D* is adjacent to at least one
11: (4)                 member of *D* [1]_.
12: (4)                 Parameters
13: (4)                 ----------
14: (4)                 G : NetworkX graph
15: (4)                 start_with : node (default=None)
16: (8)                     Node to use as a starting point for the algorithm.
17: (4)                 Returns
18: (4)                 -------
19: (4)                 D : set
20: (8)                     A dominating set for G.
21: (4)                 Notes
22: (4)                 -----
23: (4)                 This function is an implementation of algorithm 7 in [2]_ which
24: (4)                 finds some dominating set, not necessarily the smallest one.
25: (4)                 See also
26: (4)                 --------
27: (4)                 is_dominating_set
28: (4)                 References
29: (4)                 ----------
30: (4)                 .. [1] https://en.wikipedia.org/wiki/Dominating_set
31: (4)                 .. [2] Abdol-Hossein Esfahanian. Connectivity Algorithms.
32: (8)                     http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
33: (4)                 """
34: (4)                 all_nodes = set(G)
35: (4)                 if start_with is None:
36: (8)                     start_with = arbitrary_element(all_nodes)
37: (4)                 if start_with not in G:
38: (8)                     raise nx.NetworkXError(f"node {start_with} is not in G")
39: (4)                 dominating_set = {start_with}
40: (4)                 dominated_nodes = set(G[start_with])
41: (4)                 remaining_nodes = all_nodes - dominated_nodes - dominating_set
42: (4)                 while remaining_nodes:
43: (8)                     v = remaining_nodes.pop()
44: (8)                     undominated_nbrs = set(G[v]) - dominating_set
45: (8)                     dominating_set.add(v)
46: (8)                     dominated_nodes |= undominated_nbrs
47: (8)                     remaining_nodes -= undominated_nbrs
48: (4)                 return dominating_set
49: (0)             @nx._dispatchable
50: (0)             def is_dominating_set(G, nbunch):
51: (4)                 """Checks if `nbunch` is a dominating set for `G`.
52: (4)                 A *dominating set* for a graph with node set *V* is a subset *D* of
53: (4)                 *V* such that every node not in *D* is adjacent to at least one
54: (4)                 member of *D* [1]_.
55: (4)                 Parameters
56: (4)                 ----------
57: (4)                 G : NetworkX graph
58: (4)                 nbunch : iterable
59: (8)                     An iterable of nodes in the graph `G`.
60: (4)                 See also
61: (4)                 --------
62: (4)                 dominating_set
63: (4)                 References
64: (4)                 ----------
65: (4)                 .. [1] https://en.wikipedia.org/wiki/Dominating_set
66: (4)                 """
67: (4)                 testset = {n for n in nbunch if n in G}
68: (4)                 nbrs = set(chain.from_iterable(G[n] for n in testset))
69: (4)                 return len(set(G) - testset - nbrs) == 0
```

File 28 - efficiency_measures.py:

```
1: (0)              """Provides functions for computing the efficiency of nodes and graphs."""
```

```
import networkx as nx
from networkx.exception import NetworkXNoPath
from ..utils import not_implemented_for
__all__ = ["efficiency", "local_efficiency", "global_efficiency"]
@not_implemented_for("directed")
@nx._dispatchable
def efficiency(G, u, v):
    """Returns the efficiency of a pair of nodes in a graph.
    The *efficiency* of a pair of nodes is the multiplicative inverse of the
    shortest path distance between the nodes [1]_. Returns 0 if no path
    between nodes.
    Parameters
    ----------
    G : :class:`networkx.Graph`
        An undirected graph for which to compute the average local efficiency.
    u, v : node
        Nodes in the graph ``G``.
    Returns
    -------
    float
        Multiplicative inverse of the shortest path distance between the
nodes.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
    >>> nx.efficiency(G, 2, 3)  # this gives efficiency for node 2 and 3
    0.5
    Notes
    -----
    Edge weights are ignored when computing the shortest path distances.
    See also
    --------
    local_efficiency
    global_efficiency
    References
    ----------
    .. [1] Latora, Vito, and Massimo Marchiori.
           "Efficient behavior of small-world networks."
           *Physical Review Letters* 87.19 (2001): 198701.
           <https://doi.org/10.1103/PhysRevLett.87.198701>
    """
    try:
        eff = 1 / nx.shortest_path_length(G, u, v)
    except NetworkXNoPath:
        eff = 0
    return eff
@not_implemented_for("directed")
@nx._dispatchable
def global_efficiency(G):
    """Returns the average global efficiency of the graph.
    The *efficiency* of a pair of nodes in a graph is the multiplicative
    inverse of the shortest path distance between the nodes. The *average
    global efficiency* of a graph is the average efficiency of all pairs of
    nodes [1]_.
    Parameters
    ----------
    G : :class:`networkx.Graph`
        An undirected graph for which to compute the average global
efficiency.
    Returns
    -------
    float
        The average global efficiency of the graph.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
    >>> round(nx.global_efficiency(G), 12)
    0.916666666667
    Notes
```

```
69: (4)                 -----
70: (4)                 Edge weights are ignored when computing the shortest path distances.
71: (4)                 See also
72: (4)                 --------
73: (4)                 local_efficiency
74: (4)                 References
75: (4)                 ----------
76: (4)                 .. [1] Latora, Vito, and Massimo Marchiori.
77: (11)                       "Efficient behavior of small-world networks."
78: (11)                       *Physical Review Letters* 87.19 (2001): 198701.
79: (11)                       <https://doi.org/10.1103/PhysRevLett.87.198701>
80: (4)                 """
81: (4)                 n = len(G)
82: (4)                 denom = n * (n - 1)
83: (4)                 if denom != 0:
84: (8)                     lengths = nx.all_pairs_shortest_path_length(G)
85: (8)                     g_eff = 0
86: (8)                     for source, targets in lengths:
87: (12)                        for target, distance in targets.items():
88: (16)                            if distance > 0:
89: (20)                                g_eff += 1 / distance
90: (8)                     g_eff /= denom
91: (4)                 else:
92: (8)                     g_eff = 0
93: (4)                 return g_eff
94: (0)             @not_implemented_for("directed")
95: (0)             @nx._dispatchable
96: (0)             def local_efficiency(G):
97: (4)                 """Returns the average local efficiency of the graph.
98: (4)                 The *efficiency* of a pair of nodes in a graph is the multiplicative
99: (4)                 inverse of the shortest path distance between the nodes. The *local
100: (4)                efficiency* of a node in the graph is the average global efficiency of the
101: (4)                subgraph induced by the neighbors of the node. The *average local
102: (4)                efficiency* is the average of the local efficiencies of each node [1]_.
103: (4)                Parameters
104: (4)                ----------
105: (4)                G : :class:`networkx.Graph`
106: (8)                    An undirected graph for which to compute the average local efficiency.
107: (4)                Returns
108: (4)                -------
109: (4)                float
110: (8)                    The average local efficiency of the graph.
111: (4)                Examples
112: (4)                --------
113: (4)                >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
114: (4)                >>> nx.local_efficiency(G)
115: (4)                0.9166666666666667
116: (4)                Notes
117: (4)                -----
118: (4)                Edge weights are ignored when computing the shortest path distances.
119: (4)                See also
120: (4)                --------
121: (4)                global_efficiency
122: (4)                References
123: (4)                ----------
124: (4)                .. [1] Latora, Vito, and Massimo Marchiori.
125: (11)                      "Efficient behavior of small-world networks."
126: (11)                      *Physical Review Letters* 87.19 (2001): 198701.
127: (11)                      <https://doi.org/10.1103/PhysRevLett.87.198701>
128: (4)                """
129: (4)                efficiency_list = (global_efficiency(G.subgraph(G[v])) for v in G)
130: (4)                return sum(efficiency_list) / len(G)
```

File 29 - euler.py:

```
1: (0)              """
2: (0)              Eulerian circuits and graphs.
```

```
3: (0)              """
4: (0)              from itertools import combinations
5: (0)              import networkx as nx
6: (0)              from ..utils import arbitrary_element, not_implemented_for
7: (0)              __all__ = [
8: (4)                  "is_eulerian",
9: (4)                  "eulerian_circuit",
10: (4)                 "eulerize",
11: (4)                 "is_semieulerian",
12: (4)                 "has_eulerian_path",
13: (4)                 "eulerian_path",
14: (0)             ]
15: (0)             @nx._dispatchable
16: (0)             def is_eulerian(G):
17: (4)                 """Returns True if and only if `G` is Eulerian.
18: (4)                 A graph is *Eulerian* if it has an Eulerian circuit. An *Eulerian
19: (4)                 circuit* is a closed walk that includes each edge of a graph exactly
20: (4)                 once.
21: (4)                 Graphs with isolated vertices (i.e. vertices with zero degree) are not
22: (4)                 considered to have Eulerian circuits. Therefore, if the graph is not
23: (4)                 connected (or not strongly connected, for directed graphs), this function
24: (4)                 returns False.
25: (4)                 Parameters
26: (4)                 ----------
27: (4)                 G : NetworkX graph
28: (7)                    A graph, either directed or undirected.
29: (4)                 Examples
30: (4)                 --------
31: (4)                 >>> nx.is_eulerian(nx.DiGraph({0: [3], 1: [2], 2: [3], 3: [0, 1]}))
32: (4)                 True
33: (4)                 >>> nx.is_eulerian(nx.complete_graph(5))
34: (4)                 True
35: (4)                 >>> nx.is_eulerian(nx.petersen_graph())
36: (4)                 False
37: (4)                 If you prefer to allow graphs with isolated vertices to have Eulerian
circuits,
38: (4)                 you can first remove such vertices and then call `is_eulerian` as below
example shows.
39: (4)                 >>> G = nx.Graph([(0, 1), (1, 2), (0, 2)])
40: (4)                 >>> G.add_node(3)
41: (4)                 >>> nx.is_eulerian(G)
42: (4)                 False
43: (4)                 >>> G.remove_nodes_from(list(nx.isolates(G)))
44: (4)                 >>> nx.is_eulerian(G)
45: (4)                 True
46: (4)                 """
47: (4)                 if G.is_directed():
48: (8)                     return all(
49: (12)                        G.in_degree(n) == G.out_degree(n) for n in G
50: (8)                     ) and nx.is_strongly_connected(G)
51: (4)                 return all(d % 2 == 0 for v, d in G.degree()) and nx.is_connected(G)
52: (0)             @nx._dispatchable
53: (0)             def is_semieulerian(G):
54: (4)                 """Return True iff `G` is semi-Eulerian.
55: (4)                 G is semi-Eulerian if it has an Eulerian path but no Eulerian circuit.
56: (4)                 See Also
57: (4)                 --------
58: (4)                 has_eulerian_path
59: (4)                 is_eulerian
60: (4)                 """
61: (4)                 return has_eulerian_path(G) and not is_eulerian(G)
62: (0)             def _find_path_start(G):
63: (4)                 """Return a suitable starting vertex for an Eulerian path.
64: (4)                 If no path exists, return None.
65: (4)                 """
66: (4)                 if not has_eulerian_path(G):
67: (8)                     return None
68: (4)                 if is_eulerian(G):
69: (8)                     return arbitrary_element(G)
```

```
(4)                 if G.is_directed():
(8)                     v1, v2 = (v for v in G if G.in_degree(v) != G.out_degree(v))
(8)                     if G.out_degree(v1) > G.in_degree(v1):
(12)                        return v1
(8)                     else:
(12)                        return v2
(4)                 else:
(8)                     start = [v for v in G if G.degree(v) % 2 != 0][0]
(8)                     return start
(0)             def _simplegraph_eulerian_circuit(G, source):
(4)                 if G.is_directed():
(8)                     degree = G.out_degree
(8)                     edges = G.out_edges
(4)                 else:
(8)                     degree = G.degree
(8)                     edges = G.edges
(4)                 vertex_stack = [source]
(4)                 last_vertex = None
(4)                 while vertex_stack:
(8)                     current_vertex = vertex_stack[-1]
(8)                     if degree(current_vertex) == 0:
(12)                        if last_vertex is not None:
(16)                            yield (last_vertex, current_vertex)
(12)                        last_vertex = current_vertex
(12)                        vertex_stack.pop()
(8)                     else:
(12)                        _, next_vertex = arbitrary_element(edges(current_vertex))
(12)                        vertex_stack.append(next_vertex)
(12)                        G.remove_edge(current_vertex, next_vertex)
(0)             def _multigraph_eulerian_circuit(G, source):
(4)                 if G.is_directed():
(8)                     degree = G.out_degree
(8)                     edges = G.out_edges
(4)                 else:
(8)                     degree = G.degree
(8)                     edges = G.edges
(4)                 vertex_stack = [(source, None)]
(4)                 last_vertex = None
(4)                 last_key = None
(4)                 while vertex_stack:
(8)                     current_vertex, current_key = vertex_stack[-1]
(8)                     if degree(current_vertex) == 0:
(12)                        if last_vertex is not None:
(16)                            yield (last_vertex, current_vertex, last_key)
(12)                        last_vertex, last_key = current_vertex, current_key
(12)                        vertex_stack.pop()
(8)                     else:
(12)                        triple = arbitrary_element(edges(current_vertex, keys=True))
(12)                        _, next_vertex, next_key = triple
(12)                        vertex_stack.append((next_vertex, next_key))
(12)                        G.remove_edge(current_vertex, next_vertex, next_key)
(0)             @nx._dispatchable
(0)             def eulerian_circuit(G, source=None, keys=False):
(4)                 """Returns an iterator over the edges of an Eulerian circuit in `G`.
(4)                 An *Eulerian circuit* is a closed walk that includes each edge of a
(4)                 graph exactly once.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    A graph, either directed or undirected.
(4)                 source : node, optional
(7)                    Starting node for circuit.
(4)                 keys : bool
(7)                    If False, edges generated by this function will be of the form
(7)                    ``(u, v)``. Otherwise, edges will be of the form ``(u, v, k)``.
(7)                    This option is ignored unless `G` is a multigraph.
(4)                 Returns
(4)                 -------
(4)                 edges : iterator
```

```
(7)                 An iterator over edges in the Eulerian circuit.
(4)             Raises
(4)             ------
(4)             NetworkXError
(7)                 If the graph is not Eulerian.
(4)             See Also
(4)             --------
(4)             is_eulerian
(4)             Notes
(4)             -----
(4)             This is a linear time implementation of an algorithm adapted from [1]_.
(4)             For general information about Euler tours, see [2]_.
(4)             References
(4)             ----------
(4)             .. [1] J. Edmonds, E. L. Johnson.
(7)                Matching, Euler tours and the Chinese postman.
(7)                Mathematical programming, Volume 5, Issue 1 (1973), 111-114.
(4)             .. [2] https://en.wikipedia.org/wiki/Eulerian_path
(4)             Examples
(4)             --------
(4)             To get an Eulerian circuit in an undirected graph::
(8)                 >>> G = nx.complete_graph(3)
(8)                 >>> list(nx.eulerian_circuit(G))
(8)                 [(0, 2), (2, 1), (1, 0)]
(8)                 >>> list(nx.eulerian_circuit(G, source=1))
(8)                 [(1, 2), (2, 0), (0, 1)]
(4)             To get the sequence of vertices in an Eulerian circuit::
(8)                 >>> [u for u, v in nx.eulerian_circuit(G)]
(8)                 [0, 2, 1]
(4)             """
(4)             if not is_eulerian(G):
(8)                 raise nx.NetworkXError("G is not Eulerian.")
(4)             if G.is_directed():
(8)                 G = G.reverse()
(4)             else:
(8)                 G = G.copy()
(4)             if source is None:
(8)                 source = arbitrary_element(G)
(4)             if G.is_multigraph():
(8)                 for u, v, k in _multigraph_eulerian_circuit(G, source):
(12)                    if keys:
(16)                        yield u, v, k
(12)                    else:
(16)                        yield u, v
(4)             else:
(8)                 yield from _simplegraph_eulerian_circuit(G, source)
(0)         @nx._dispatchable
(0)         def has_eulerian_path(G, source=None):
(4)             """Return True iff `G` has an Eulerian path.
(4)             An Eulerian path is a path in a graph which uses each edge of a graph
(4)             exactly once. If `source` is specified, then this function checks
(4)             whether an Eulerian path that starts at node `source` exists.
(4)             A directed graph has an Eulerian path iff:
(8)                 - at most one vertex has out_degree - in_degree = 1,
(8)                 - at most one vertex has in_degree - out_degree = 1,
(8)                 - every other vertex has equal in_degree and out_degree,
(8)                 - and all of its vertices belong to a single connected
(10)                  component of the underlying undirected graph.
(4)             If `source` is not None, an Eulerian path starting at `source` exists if
no
(4)             other node has out_degree - in_degree = 1. This is equivalent to either
(4)             there exists an Eulerian circuit or `source` has out_degree - in_degree =
1
(4)             and the conditions above hold.
(4)             An undirected graph has an Eulerian path iff:
(8)                 - exactly zero or two vertices have odd degree,
(8)                 - and all of its vertices belong to a single connected component.
(4)             If `source` is not None, an Eulerian path starting at `source` exists if
(4)             either there exists an Eulerian circuit or `source` has an odd degree and
```

```
the
(4)                 conditions above hold.
(4)                 Graphs with isolated vertices (i.e. vertices with zero degree) are not
considered
(4)                 to have an Eulerian path. Therefore, if the graph is not connected (or not
strongly
(4)                 connected, for directed graphs), this function returns False.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(8)                     The graph to find an euler path in.
(4)                 source : node, optional
(8)                     Starting node for path.
(4)                 Returns
(4)                 -------
(4)                 Bool : True if G has an Eulerian path.
(4)                 Examples
(4)                 --------
(4)                 If you prefer to allow graphs with isolated vertices to have Eulerian
path,
(4)                 you can first remove such vertices and then call `has_eulerian_path` as
below example shows.
(4)                 >>> G = nx.Graph([(0, 1), (1, 2), (0, 2)])
(4)                 >>> G.add_node(3)
(4)                 >>> nx.has_eulerian_path(G)
(4)                 False
(4)                 >>> G.remove_nodes_from(list(nx.isolates(G)))
(4)                 >>> nx.has_eulerian_path(G)
(4)                 True
(4)                 See Also
(4)                 --------
(4)                 is_eulerian
(4)                 eulerian_path
(4)                 """
(4)                 if nx.is_eulerian(G):
(8)                     return True
(4)                 if G.is_directed():
(8)                     ins = G.in_degree
(8)                     outs = G.out_degree
(8)                     if source is not None and outs[source] - ins[source] != 1:
(12)                        return False
(8)                     unbalanced_ins = 0
(8)                     unbalanced_outs = 0
(8)                     for v in G:
(12)                        if ins[v] - outs[v] == 1:
(16)                            unbalanced_ins += 1
(12)                        elif outs[v] - ins[v] == 1:
(16)                            unbalanced_outs += 1
(12)                        elif ins[v] != outs[v]:
(16)                            return False
(8)                     return (
(12)                        unbalanced_ins <= 1 and unbalanced_outs <= 1 and
nx.is_weakly_connected(G)
(8)                     )
(4)                 else:
(8)                     if source is not None and G.degree[source] % 2 != 1:
(12)                        return False
(8)                     return sum(d % 2 == 1 for v, d in G.degree()) == 2 and
nx.is_connected(G)
(0)             @nx._dispatchable
(0)             def eulerian_path(G, source=None, keys=False):
(4)                 """Return an iterator over the edges of an Eulerian path in `G`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(8)                     The graph in which to look for an eulerian path.
(4)                 source : node or None (default: None)
(8)                     The node at which to start the search. None means search over all
(8)                     starting nodes.
```

```
    keys : Bool (default: False)
        Indicates whether to yield edge 3-tuples (u, v, edge_key).
        The default yields edge 2-tuples
    Yields
    ------
    Edge tuples along the eulerian path.
    Warning: If `source` provided is not the start node of an Euler path
    will raise error even if an Euler Path exists.
    """
    if not has_eulerian_path(G, source):
        raise nx.NetworkXError("Graph has no Eulerian paths.")
    if G.is_directed():
        G = G.reverse()
        if source is None or nx.is_eulerian(G) is False:
            source = _find_path_start(G)
        if G.is_multigraph():
            for u, v, k in _multigraph_eulerian_circuit(G, source):
                if keys:
                    yield u, v, k
                else:
                    yield u, v
        else:
            yield from _simplegraph_eulerian_circuit(G, source)
    else:
        G = G.copy()
        if source is None:
            source = _find_path_start(G)
        if G.is_multigraph():
            if keys:
                yield from reversed(
                    [(v, u, k) for u, v, k in _multigraph_eulerian_circuit(G, source)]
                )
            else:
                yield from reversed(
                    [(v, u) for u, v, k in _multigraph_eulerian_circuit(G, source)]
                )
        else:
            yield from reversed(
                [(v, u) for u, v in _simplegraph_eulerian_circuit(G, source)]
            )
@not_implemented_for("directed")
@nx._dispatchable(returns_graph=True)
def eulerize(G):
    """Transforms a graph into an Eulerian graph.
    If `G` is Eulerian the result is `G` as a MultiGraph, otherwise the result is a smallest
    (in terms of the number of edges) multigraph whose underlying simple graph is `G`.
    Parameters
    ----------
    G : NetworkX graph
       An undirected graph
    Returns
    -------
    G : NetworkX multigraph
    Raises
    ------
    NetworkXError
       If the graph is not connected.
    See Also
    --------
    is_eulerian
    eulerian_circuit
    References
    ----------
    .. [1] J. Edmonds, E. L. Johnson.
       Matching, Euler tours and the Chinese postman.
```

```
333: (7)                   Mathematical programming, Volume 5, Issue 1 (1973), 111-114.
334: (4)                .. [2] https://en.wikipedia.org/wiki/Eulerian_path
335: (4)                .. [3] http://web.math.princeton.edu/math_alive/5/Notes1.pdf
336: (4)                Examples
337: (4)                --------
338: (8)                    >>> G = nx.complete_graph(10)
339: (8)                    >>> H = nx.eulerize(G)
340: (8)                    >>> nx.is_eulerian(H)
341: (8)                    True
342: (4)                """
343: (4)                if G.order() == 0:
344: (8)                    raise nx.NetworkXPointlessConcept("Cannot Eulerize null graph")
345: (4)                if not nx.is_connected(G):
346: (8)                    raise nx.NetworkXError("G is not connected")
347: (4)                odd_degree_nodes = [n for n, d in G.degree() if d % 2 == 1]
348: (4)                G = nx.MultiGraph(G)
349: (4)                if len(odd_degree_nodes) == 0:
350: (8)                    return G
351: (4)                odd_deg_pairs_paths = [
352: (8)                    (m, {n: nx.shortest_path(G, source=m, target=n)})
353: (8)                    for m, n in combinations(odd_degree_nodes, 2)
354: (4)                ]
355: (4)                upper_bound_on_max_path_length = len(G) + 1
356: (4)                Gp = nx.Graph()
357: (4)                for n, Ps in odd_deg_pairs_paths:
358: (8)                    for m, P in Ps.items():
359: (12)                       if n != m:
360: (16)                           Gp.add_edge(
361: (20)                               m, n, weight=upper_bound_on_max_path_length - len(P),
path=P
362: (16)                           )
363: (4)                best_matching = nx.Graph(list(nx.max_weight_matching(Gp)))
364: (4)                for m, n in best_matching.edges():
365: (8)                    path = Gp[m][n]["path"]
366: (8)                    G.add_edges_from(nx.utils.pairwise(path))
367: (4)                return G
```

File 30 - graph_hashing.py:

```
1: (0)              """
2: (0)              Functions for hashing graphs to strings.
3: (0)              Isomorphic graphs should be assigned identical hashes.
4: (0)              For now, only Weisfeiler-Lehman hashing is implemented.
5: (0)              """
6: (0)              from collections import Counter, defaultdict
7: (0)              from hashlib import blake2b
8: (0)              import networkx as nx
9: (0)              __all__ = ["weisfeiler_lehman_graph_hash",
"weisfeiler_lehman_subgraph_hashes"]
10: (0)             def _hash_label(label, digest_size):
11: (4)                 return blake2b(label.encode("ascii"), digest_size=digest_size).hexdigest()
12: (0)             def _init_node_labels(G, edge_attr, node_attr):
13: (4)                 if node_attr:
14: (8)                     return {u: str(dd[node_attr]) for u, dd in G.nodes(data=True)}
15: (4)                 elif edge_attr:
16: (8)                     return {u: "" for u in G}
17: (4)                 else:
18: (8)                     return {u: str(deg) for u, deg in G.degree()}
19: (0)             def _neighborhood_aggregate(G, node, node_labels, edge_attr=None):
20: (4)                 """
21: (4)                 Compute new labels for given node by aggregating
22: (4)                 the labels of each node's neighbors.
23: (4)                 """
24: (4)                 label_list = []
25: (4)                 for nbr in G.neighbors(node):
26: (8)                     prefix = "" if edge_attr is None else str(G[node][nbr][edge_attr])
27: (8)                     label_list.append(prefix + node_labels[nbr])
```

```
 28: (4)                 return node_labels[node] + "".join(sorted(label_list))
 29: (0)             @nx._dispatchable(edge_attrs={"edge_attr": None}, node_attrs="node_attr")
 30: (0)             def weisfeiler_lehman_graph_hash(
 31: (4)                 G, edge_attr=None, node_attr=None, iterations=3, digest_size=16
 32: (0)             ):
 33: (4)                 """Return Weisfeiler Lehman (WL) graph hash.
 34: (4)                 The function iteratively aggregates and hashes neighborhoods of each node.
 35: (4)                 After each node's neighbors are hashed to obtain updated node labels,
 36: (4)                 a hashed histogram of resulting labels is returned as the final hash.
 37: (4)                 Hashes are identical for isomorphic graphs and strong guarantees that
 38: (4)                 non-isomorphic graphs will get different hashes. See [1]_ for details.
 39: (4)                 If no node or edge attributes are provided, the degree of each node
 40: (4)                 is used as its initial label.
 41: (4)                 Otherwise, node and/or edge labels are used to compute the hash.
 42: (4)                 Parameters
 43: (4)                 ----------
 44: (4)                 G : graph
 45: (8)                     The graph to be hashed.
 46: (8)                     Can have node and/or edge attributes. Can also have no attributes.
 47: (4)                 edge_attr : string, optional (default=None)
 48: (8)                     The key in edge attribute dictionary to be used for hashing.
 49: (8)                     If None, edge labels are ignored.
 50: (4)                 node_attr: string, optional (default=None)
 51: (8)                     The key in node attribute dictionary to be used for hashing.
 52: (8)                     If None, and no edge_attr given, use the degrees of the nodes as
labels.
 53: (4)                 iterations: int, optional (default=3)
 54: (8)                     Number of neighbor aggregations to perform.
 55: (8)                     Should be larger for larger graphs.
 56: (4)                 digest_size: int, optional (default=16)
 57: (8)                     Size (in bits) of blake2b hash digest to use for hashing node labels.
 58: (4)                 Returns
 59: (4)                 -------
 60: (4)                 h : string
 61: (8)                     Hexadecimal string corresponding to hash of the input graph.
 62: (4)                 Examples
 63: (4)                 --------
 64: (4)                 Two graphs with edge attributes that are isomorphic, except for
 65: (4)                 differences in the edge labels.
 66: (4)                 >>> G1 = nx.Graph()
 67: (4)                 >>> G1.add_edges_from(
 68: (4)                 ...     [
 69: (4)                 ...         (1, 2, {"label": "A"}),
 70: (4)                 ...         (2, 3, {"label": "A"}),
 71: (4)                 ...         (3, 1, {"label": "A"}),
 72: (4)                 ...         (1, 4, {"label": "B"}),
 73: (4)                 ...     ]
 74: (4)                 ... )
 75: (4)                 >>> G2 = nx.Graph()
 76: (4)                 >>> G2.add_edges_from(
 77: (4)                 ...     [
 78: (4)                 ...         (5, 6, {"label": "B"}),
 79: (4)                 ...         (6, 7, {"label": "A"}),
 80: (4)                 ...         (7, 5, {"label": "A"}),
 81: (4)                 ...         (7, 8, {"label": "A"}),
 82: (4)                 ...     ]
 83: (4)                 ... )
 84: (4)                 Omitting the `edge_attr` option, results in identical hashes.
 85: (4)                 >>> nx.weisfeiler_lehman_graph_hash(G1)
 86: (4)                 '7bc4dde9a09d0b94c5097b219891d81a'
 87: (4)                 >>> nx.weisfeiler_lehman_graph_hash(G2)
 88: (4)                 '7bc4dde9a09d0b94c5097b219891d81a'
 89: (4)                 With edge labels, the graphs are no longer assigned
 90: (4)                 the same hash digest.
 91: (4)                 >>> nx.weisfeiler_lehman_graph_hash(G1, edge_attr="label")
 92: (4)                 'c653d85538bcf041d88c011f4f905f10'
 93: (4)                 >>> nx.weisfeiler_lehman_graph_hash(G2, edge_attr="label")
 94: (4)                 '3dcd84af1ca855d0eff3c978d88e7ec7'
 95: (4)                 Notes
```

```
96: (4)                 -----
97: (4)                 To return the WL hashes of each subgraph of a graph, use
98: (4)                 `weisfeiler_lehman_subgraph_hashes`
99: (4)                 Similarity between hashes does not imply similarity between graphs.
100: (4)                References
101: (4)                ----------
102: (4)                .. [1] Shervashidze, Nino, Pascal Schweitzer, Erik Jan Van Leeuwen,
103: (7)                   Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler Lehman
104: (7)                   Graph Kernels. Journal of Machine Learning Research. 2011.
105: (7)                   http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf
106: (4)                See also
107: (4)                --------
108: (4)                weisfeiler_lehman_subgraph_hashes
109: (4)                """
110: (4)                def weisfeiler_lehman_step(G, labels, edge_attr=None):
111: (8)                    """
112: (8)                    Apply neighborhood aggregation to each node
113: (8)                    in the graph.
114: (8)                    Computes a dictionary with labels for each node.
115: (8)                    """
116: (8)                    new_labels = {}
117: (8)                    for node in G.nodes():
118: (12)                       label = _neighborhood_aggregate(G, node, labels,
edge_attr=edge_attr)
119: (12)                       new_labels[node] = _hash_label(label, digest_size)
120: (8)                    return new_labels
121: (4)                node_labels = _init_node_labels(G, edge_attr, node_attr)
122: (4)                subgraph_hash_counts = []
123: (4)                for _ in range(iterations):
124: (8)                    node_labels = weisfeiler_lehman_step(G, node_labels,
edge_attr=edge_attr)
125: (8)                    counter = Counter(node_labels.values())
126: (8)                    subgraph_hash_counts.extend(sorted(counter.items(), key=lambda x:
x[0]))
127: (4)                return _hash_label(str(tuple(subgraph_hash_counts)), digest_size)
128: (0)            @nx._dispatchable(edge_attrs={"edge_attr": None}, node_attrs="node_attr")
129: (0)            def weisfeiler_lehman_subgraph_hashes(
130: (4)                G,
131: (4)                edge_attr=None,
132: (4)                node_attr=None,
133: (4)                iterations=3,
134: (4)                digest_size=16,
135: (4)                include_initial_labels=False,
136: (0)            ):
137: (4)                """
138: (4)                Return a dictionary of subgraph hashes by node.
139: (4)                Dictionary keys are nodes in `G`, and values are a list of hashes.
140: (4)                Each hash corresponds to a subgraph rooted at a given node u in `G`.
141: (4)                Lists of subgraph hashes are sorted in increasing order of depth from
142: (4)                their root node, with the hash at index i corresponding to a subgraph
143: (4)                of nodes at most i edges distance from u. Thus, each list will contain
144: (4)                `iterations` elements - a hash for a subgraph at each depth. If
145: (4)                `include_initial_labels` is set to `True`, each list will additionally
146: (4)                have contain a hash of the initial node label (or equivalently a
147: (4)                subgraph of depth 0) prepended, totalling ``iterations + 1`` elements.
148: (4)                The function iteratively aggregates and hashes neighborhoods of each node.
149: (4)                This is achieved for each step by replacing for each node its label from
150: (4)                the previous iteration with its hashed 1-hop neighborhood aggregate.
151: (4)                The new node label is then appended to a list of node labels for each
152: (4)                node.
153: (4)                To aggregate neighborhoods for a node $u$ at each step, all labels of
154: (4)                nodes adjacent to $u$ are concatenated. If the `edge_attr` parameter is
set,
155: (4)                labels for each neighboring node are prefixed with the value of this
attribute
156: (4)                along the connecting edge from this neighbor to node $u$. The resulting
string
157: (4)                is then hashed to compress this information into a fixed digest size.
158: (4)                Thus, at the $i$-th iteration, nodes within $i$ hops influence any given
```

```
hashed node label. We can therefore say that at depth $i$ for node $u$
we have a hash for a subgraph induced by the $i$-hop neighborhood of $u$.
The output can be used to to create general Weisfeiler-Lehman graph kernels,
or generate features for graphs or nodes - for example to generate 'words' in
a graph as seen in the 'graph2vec' algorithm.
See [1]_ & [2]_ respectively for details.
Hashes are identical for isomorphic subgraphs and there exist strong
guarantees that non-isomorphic graphs will get different hashes.
See [1]_ for details.
If no node or edge attributes are provided, the degree of each node
is used as its initial label.
Otherwise, node and/or edge labels are used to compute the hash.
Parameters
----------
G : graph
    The graph to be hashed.
    Can have node and/or edge attributes. Can also have no attributes.
edge_attr : string, optional (default=None)
    The key in edge attribute dictionary to be used for hashing.
    If None, edge labels are ignored.
node_attr : string, optional (default=None)
    The key in node attribute dictionary to be used for hashing.
    If None, and no edge_attr given, use the degrees of the nodes as labels.

    If None, and edge_attr is given, each node starts with an identical label.
iterations : int, optional (default=3)
    Number of neighbor aggregations to perform.
    Should be larger for larger graphs.
digest_size : int, optional (default=16)
    Size (in bits) of blake2b hash digest to use for hashing node labels.
    The default size is 16 bits.
include_initial_labels : bool, optional (default=False)
    If True, include the hashed initial node label as the first subgraph
    hash for each node.
Returns
-------
node_subgraph_hashes : dict
    A dictionary with each key given by a node in G, and each value given
    by the subgraph hashes in order of depth from the key node.
Examples
--------
Finding similar nodes in different graphs:
>>> G1 = nx.Graph()
>>> G1.add_edges_from([(1, 2), (2, 3), (2, 4), (3, 5), (4, 6), (5, 7), (6, 7)])
>>> G2 = nx.Graph()
>>> G2.add_edges_from([(1, 3), (2, 3), (1, 6), (1, 5), (4, 6)])
>>> g1_hashes = nx.weisfeiler_lehman_subgraph_hashes(G1, iterations=3, digest_size=8)
>>> g2_hashes = nx.weisfeiler_lehman_subgraph_hashes(G2, iterations=3, digest_size=8)
Even though G1 and G2 are not isomorphic (they have different numbers of edges),
the hash sequence of depth 3 for node 1 in G1 and node 5 in G2 are similar:
>>> g1_hashes[1]
['a93b64973cfc8897', 'db1b43ae35a1878f', '57872a7d2059c1c0']
>>> g2_hashes[5]
['a93b64973cfc8897', 'db1b43ae35a1878f', '1716d2a4012fa4bc']
The first 2 WL subgraph hashes match. From this we can conclude that it's very
likely the neighborhood of 2 hops around these nodes are isomorphic.
However the 3-hop neighborhoods of ``G1`` and ``G2`` are not isomorphic since the
3rd hashes in the lists above are not equal.
These nodes may be candidates to be classified together since their local
```

```
topology
217: (4)                is similar.
218: (4)                Notes
219: (4)                -----
220: (4)                To hash the full graph when subgraph hashes are not needed, use
221: (4)                `weisfeiler_lehman_graph_hash` for efficiency.
222: (4)                Similarity between hashes does not imply similarity between graphs.
223: (4)                References
224: (4)                ----------
225: (4)                .. [1] Shervashidze, Nino, Pascal Schweitzer, Erik Jan Van Leeuwen,
226: (7)                   Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler Lehman
227: (7)                   Graph Kernels. Journal of Machine Learning Research. 2011.
228: (7)                   http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf
229: (4)                .. [2] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan,
230: (7)                   Lihui Chen, Yang Liu and Shantanu Jaiswa. graph2vec: Learning
231: (7)                   Distributed Representations of Graphs. arXiv. 2017
232: (7)                   https://arxiv.org/pdf/1707.05005.pdf
233: (4)                See also
234: (4)                --------
235: (4)                weisfeiler_lehman_graph_hash
236: (4)                """
237: (4)                def weisfeiler_lehman_step(G, labels, node_subgraph_hashes,
edge_attr=None):
238: (8)                    """
239: (8)                    Apply neighborhood aggregation to each node
240: (8)                    in the graph.
241: (8)                    Computes a dictionary with labels for each node.
242: (8)                    Appends the new hashed label to the dictionary of subgraph hashes
243: (8)                    originating from and indexed by each node in G
244: (8)                    """
245: (8)                    new_labels = {}
246: (8)                    for node in G.nodes():
247: (12)                       label = _neighborhood_aggregate(G, node, labels,
edge_attr=edge_attr)
248: (12)                       hashed_label = _hash_label(label, digest_size)
249: (12)                       new_labels[node] = hashed_label
250: (12)                       node_subgraph_hashes[node].append(hashed_label)
251: (8)                    return new_labels
252: (4)                node_labels = _init_node_labels(G, edge_attr, node_attr)
253: (4)                if include_initial_labels:
254: (8)                    node_subgraph_hashes = {
255: (12)                       k: [_hash_label(v, digest_size)] for k, v in node_labels.items()
256: (8)                    }
257: (4)                else:
258: (8)                    node_subgraph_hashes = defaultdict(list)
259: (4)                for _ in range(iterations):
260: (8)                    node_labels = weisfeiler_lehman_step(
261: (12)                       G, node_labels, node_subgraph_hashes, edge_attr
262: (8)                    )
263: (4)                return dict(node_subgraph_hashes)

----------------------------------------

File 31 - graphical.py:

1: (0)              """Test sequences for graphiness.
2: (0)              """
3: (0)              import heapq
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "is_graphical",
7: (4)                  "is_multigraphical",
8: (4)                  "is_pseudographical",
9: (4)                  "is_digraphical",
10: (4)                 "is_valid_degree_sequence_erdos_gallai",
11: (4)                 "is_valid_degree_sequence_havel_hakimi",
12: (0)             ]
13: (0)             @nx._dispatchable(graphs=None)
14: (0)             def is_graphical(sequence, method="eg"):
```

```
(4)                 """Returns True if sequence is a valid degree sequence.
(4)                 A degree sequence is valid if some graph can realize it.
(4)                 Parameters
(4)                 ----------
(4)                 sequence : list or iterable container
(8)                     A sequence of integer node degrees
(4)                 method : "eg" | "hh"  (default: 'eg')
(8)                     The method used to validate the degree sequence.
(8)                     "eg" corresponds to the Erdős-Gallai algorithm
(8)                     [EG1960]_, [choudum1986]_, and
(8)                     "hh" to the Havel-Hakimi algorithm
(8)                     [havel1955]_, [hakimi1962]_, [CL1996]_.
(4)                 Returns
(4)                 -------
(4)                 valid : bool
(8)                     True if the sequence is a valid degree sequence and False if not.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> sequence = (d for n, d in G.degree())
(4)                 >>> nx.is_graphical(sequence)
(4)                 True
(4)                 To test a non-graphical sequence:
(4)                 >>> sequence_list = [d for n, d in G.degree()]
(4)                 >>> sequence_list[-1] += 1
(4)                 >>> nx.is_graphical(sequence_list)
(4)                 False
(4)                 References
(4)                 ----------
(4)                 .. [EG1960] Erdős and Gallai, Mat. Lapok 11 264, 1960.
(4)                 .. [choudum1986] S.A. Choudum. "A simple proof of the Erdős-Gallai theorem
on
(7)                    graph sequences." Bulletin of the Australian Mathematical Society, 33,
(7)                    pp 67-70, 1986. https://doi.org/10.1017/S0004972700002872
(4)                 .. [havel1955] Havel, V. "A Remark on the Existence of Finite Graphs"
(7)                    Casopis Pest. Mat. 80, 477-480, 1955.
(4)                 .. [hakimi1962] Hakimi, S. "On the Realizability of a Set of Integers as
(7)                    Degrees of the Vertices of a Graph." SIAM J. Appl. Math. 10, 496-506,
1962.
(4)                 .. [CL1996] G. Chartrand and L. Lesniak, "Graphs and Digraphs",
(7)                    Chapman and Hall/CRC, 1996.
(4)                 """
(4)                 if method == "eg":
(8)                     valid = is_valid_degree_sequence_erdos_gallai(list(sequence))
(4)                 elif method == "hh":
(8)                     valid = is_valid_degree_sequence_havel_hakimi(list(sequence))
(4)                 else:
(8)                     msg = "`method` must be 'eg' or 'hh'"
(8)                     raise nx.NetworkXException(msg)
(4)                 return valid
(0)             def _basic_graphical_tests(deg_sequence):
(4)                 deg_sequence = nx.utils.make_list_of_ints(deg_sequence)
(4)                 p = len(deg_sequence)
(4)                 num_degs = [0] * p
(4)                 dmax, dmin, dsum, n = 0, p, 0, 0
(4)                 for d in deg_sequence:
(8)                     if d < 0 or d >= p:
(12)                        raise nx.NetworkXUnfeasible
(8)                     elif d > 0:
(12)                        dmax, dmin, dsum, n = max(dmax, d), min(dmin, d), dsum + d, n + 1
(12)                        num_degs[d] += 1
(4)                 if dsum % 2 or dsum > n * (n - 1):
(8)                     raise nx.NetworkXUnfeasible
(4)                 return dmax, dmin, dsum, n, num_degs
(0)             @nx._dispatchable(graphs=None)
(0)             def is_valid_degree_sequence_havel_hakimi(deg_sequence):
(4)                 r"""Returns True if deg_sequence can be realized by a simple graph.
(4)                 The validation proceeds using the Havel-Hakimi theorem
(4)                 [havel1955]_, [hakimi1962]_, [CL1996]_.
```

```
    Worst-case run time is $O(s)$ where $s$ is the sum of the sequence.
    Parameters
    ----------
    deg_sequence : list
        A list of integers where each element specifies the degree of a node
        in a graph.
    Returns
    -------
    valid : bool
        True if deg_sequence is graphical and False if not.
    Examples
    --------
    >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (3, 4), (4, 2), (5, 1), (5, 4)])
    >>> sequence = (d for _, d in G.degree())
    >>> nx.is_valid_degree_sequence_havel_hakimi(sequence)
    True
    To test a non-valid sequence:
    >>> sequence_list = [d for _, d in G.degree()]
    >>> sequence_list[-1] += 1
    >>> nx.is_valid_degree_sequence_havel_hakimi(sequence_list)
    False
    Notes
    -----
    The ZZ condition says that for the sequence d if
    .. math::
        |d| >= \frac{(\max(d) + \min(d) + 1)^2}{4*\min(d)}
    then d is graphical.  This was shown in Theorem 6 in [1]_.
    References
    ----------
    .. [1] I.E. Zverovich and V.E. Zverovich. "Contributions to the theory
       of graphic sequences", Discrete Mathematics, 105, pp. 292-303 (1992).
    .. [havel1955] Havel, V. "A Remark on the Existence of Finite Graphs"
       Casopis Pest. Mat. 80, 477-480, 1955.
    .. [hakimi1962] Hakimi, S. "On the Realizability of a Set of Integers as
       Degrees of the Vertices of a Graph." SIAM J. Appl. Math. 10, 496-506,
1962.
    .. [CL1996] G. Chartrand and L. Lesniak, "Graphs and Digraphs",
       Chapman and Hall/CRC, 1996.
    """
    try:
        dmax, dmin, dsum, n, num_degs = _basic_graphical_tests(deg_sequence)
    except nx.NetworkXUnfeasible:
        return False
    if n == 0 or 4 * dmin * n >= (dmax + dmin + 1) * (dmax + dmin + 1):
        return True
    modstubs = [0] * (dmax + 1)
    while n > 0:
        while num_degs[dmax] == 0:
            dmax -= 1
        if dmax > n - 1:
            return False
        num_degs[dmax], n = num_degs[dmax] - 1, n - 1
        mslen = 0
        k = dmax
        for i in range(dmax):
            while num_degs[k] == 0:
                k -= 1
            num_degs[k], n = num_degs[k] - 1, n - 1
            if k > 1:
                modstubs[mslen] = k - 1
                mslen += 1
        for i in range(mslen):
            stub = modstubs[i]
            num_degs[stub], n = num_degs[stub] + 1, n + 1
    return True
@nx._dispatchable(graphs=None)
def is_valid_degree_sequence_erdos_gallai(deg_sequence):
    r"""Returns True if deg_sequence can be realized by a simple graph.
    The validation is done using the Erdős-Gallai theorem [EG1960]_.
```

```
150: (4)                Parameters
151: (4)                ----------
152: (4)                deg_sequence : list
153: (8)                    A list of integers
154: (4)                Returns
155: (4)                -------
156: (4)                valid : bool
157: (8)                    True if deg_sequence is graphical and False if not.
158: (4)                Examples
159: (4)                --------
160: (4)                >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (3, 4), (4, 2), (5, 1), (5, 4)])
161: (4)                >>> sequence = (d for _, d in G.degree())
162: (4)                >>> nx.is_valid_degree_sequence_erdos_gallai(sequence)
163: (4)                True
164: (4)                To test a non-valid sequence:
165: (4)                >>> sequence_list = [d for _, d in G.degree()]
166: (4)                >>> sequence_list[-1] += 1
167: (4)                >>> nx.is_valid_degree_sequence_erdos_gallai(sequence_list)
168: (4)                False
169: (4)                Notes
170: (4)                -----
171: (4)                This implementation uses an equivalent form of the Erdős-Gallai criterion.
172: (4)                Worst-case run time is $O(n)$ where $n$ is the length of the sequence.
173: (4)                Specifically, a sequence d is graphical if and only if the
174: (4)                sum of the sequence is even and for all strong indices k in the sequence,
175: (5)                 .. math::
176: (7)                   \sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{j=k+1}^{n} \min(d_i,k)
177: (13)                        = k(n-1) - ( k \sum_{j=0}^{k-1} n_j - \sum_{j=0}^{k-1} j n_j )
178: (4)                A strong index k is any index where d_k >= k and the value n_j is the
179: (4)                number of occurrences of j in d.  The maximal strong index is called the
180: (4)                Durfee index.
181: (4)                This particular rearrangement comes from the proof of Theorem 3 in [2]_.
182: (4)                The ZZ condition says that for the sequence d if
183: (4)                .. math::
184: (8)                    |d| >= \frac{(\max(d) + \min(d) + 1)^2}{4*\min(d)}
185: (4)                then d is graphical.  This was shown in Theorem 6 in [2]_.
186: (4)                References
187: (4)                ----------
188: (4)                .. [1] A. Tripathi and S. Vijay. "A note on a theorem of Erdős & Gallai",
189: (7)                   Discrete Mathematics, 265, pp. 417-420 (2003).
190: (4)                .. [2] I.E. Zverovich and V.E. Zverovich. "Contributions to the theory
191: (7)                   of graphic sequences", Discrete Mathematics, 105, pp. 292-303 (1992).
192: (4)                .. [EG1960] Erdős and Gallai, Mat. Lapok 11 264, 1960.
193: (4)                """
194: (4)                try:
195: (8)                    dmax, dmin, dsum, n, num_degs = _basic_graphical_tests(deg_sequence)
196: (4)                except nx.NetworkXUnfeasible:
197: (8)                    return False
198: (4)                if n == 0 or 4 * dmin * n >= (dmax + dmin + 1) * (dmax + dmin + 1):
199: (8)                    return True
200: (4)                k, sum_deg, sum_nj, sum_jnj = 0, 0, 0, 0
201: (4)                for dk in range(dmax, dmin - 1, -1):
202: (8)                    if dk < k + 1:  # Check if already past Durfee index
203: (12)                       return True
204: (8)                    if num_degs[dk] > 0:
205: (12)                       run_size = num_degs[dk]  # Process a run of identical-valued
degrees
206: (12)                       if dk < k + run_size:  # Check if end of run is past Durfee index
207: (16)                           run_size = dk - k  # Adjust back to Durfee index
208: (12)                       sum_deg += run_size * dk
209: (12)                       for v in range(run_size):
210: (16)                           sum_nj += num_degs[k + v]
211: (16)                           sum_jnj += (k + v) * num_degs[k + v]
212: (12)                       k += run_size
213: (12)                       if sum_deg > k * (n - 1) - k * sum_nj + sum_jnj:
214: (16)                           return False
215: (4)                return True
216: (0)            @nx._dispatchable(graphs=None)
217: (0)            def is_multigraphical(sequence):
```

```
    """Returns True if some multigraph can realize the sequence.
    Parameters
    ----------
    sequence : list
        A list of integers
    Returns
    -------
    valid : bool
        True if deg_sequence is a multigraphic degree sequence and False if not.
    Examples
    --------
    >>> G = nx.MultiGraph([(1, 2), (1, 3), (2, 3), (3, 4), (4, 2), (5, 1), (5, 4)])
    >>> sequence = (d for _, d in G.degree())
    >>> nx.is_multigraphical(sequence)
    True
    To test a non-multigraphical sequence:
    >>> sequence_list = [d for _, d in G.degree()]
    >>> sequence_list[-1] += 1
    >>> nx.is_multigraphical(sequence_list)
    False
    Notes
    -----
    The worst-case run time is $O(n)$ where $n$ is the length of the sequence.
    References
    ----------
    .. [1] S. L. Hakimi. "On the realizability of a set of integers as
       degrees of the vertices of a linear graph", J. SIAM, 10, pp. 496-506
       (1962).
    """
    try:
        deg_sequence = nx.utils.make_list_of_ints(sequence)
    except nx.NetworkXError:
        return False
    dsum, dmax = 0, 0
    for d in deg_sequence:
        if d < 0:
            return False
        dsum, dmax = dsum + d, max(dmax, d)
    if dsum % 2 or dsum < 2 * dmax:
        return False
    return True
@nx._dispatchable(graphs=None)
def is_pseudographical(sequence):
    """Returns True if some pseudograph can realize the sequence.
    Every nonnegative integer sequence with an even sum is pseudographical
    (see [1]_).
    Parameters
    ----------
    sequence : list or iterable container
        A sequence of integer node degrees
    Returns
    -------
    valid : bool
      True if the sequence is a pseudographic degree sequence and False if not.
    Examples
    --------
    >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (3, 4), (4, 2), (5, 1), (5, 4)])
    >>> sequence = (d for _, d in G.degree())
    >>> nx.is_pseudographical(sequence)
    True
    To test a non-pseudographical sequence:
    >>> sequence_list = [d for _, d in G.degree()]
    >>> sequence_list[-1] += 1
    >>> nx.is_pseudographical(sequence_list)
    False
    Notes
```

```
    -----
    The worst-case run time is $O(n)$ where n is the length of the sequence.
    References
    ----------
    .. [1] F. Boesch and F. Harary. "Line removal algorithms for graphs
       and their degree lists", IEEE Trans. Circuits and Systems, CAS-23(12),
       pp. 778-782 (1976).
    """
    try:
        deg_sequence = nx.utils.make_list_of_ints(sequence)
    except nx.NetworkXError:
        return False
    return sum(deg_sequence) % 2 == 0 and min(deg_sequence) >= 0
@nx._dispatchable(graphs=None)
def is_digraphical(in_sequence, out_sequence):
    r"""Returns True if some directed graph can realize the in- and out-degree
    sequences.
    Parameters
    ----------
    in_sequence : list or iterable container
        A sequence of integer node in-degrees
    out_sequence : list or iterable container
        A sequence of integer node out-degrees
    Returns
    -------
    valid : bool
      True if in and out-sequences are digraphic False if not.
    Examples
    --------
    >>> G = nx.DiGraph([(1, 2), (1, 3), (2, 3), (3, 4), (4, 2), (5, 1), (5, 4)])
    >>> in_seq = (d for n, d in G.in_degree())
    >>> out_seq = (d for n, d in G.out_degree())
    >>> nx.is_digraphical(in_seq, out_seq)
    True
    To test a non-digraphical scenario:
    >>> in_seq_list = [d for n, d in G.in_degree()]
    >>> in_seq_list[-1] += 1
    >>> nx.is_digraphical(in_seq_list, out_seq)
    False
    Notes
    -----
    This algorithm is from Kleitman and Wang [1]_.
    The worst case runtime is $O(s \times \log n)$ where $s$ and $n$ are the
    sum and length of the sequences respectively.
    References
    ----------
    .. [1] D.J. Kleitman and D.L. Wang
       Algorithms for Constructing Graphs and Digraphs with Given Valences
       and Factors, Discrete Mathematics, 6(1), pp. 79-88 (1973)
    """
    try:
        in_deg_sequence = nx.utils.make_list_of_ints(in_sequence)
        out_deg_sequence = nx.utils.make_list_of_ints(out_sequence)
    except nx.NetworkXError:
        return False
    sumin, sumout, nin, nout = 0, 0, len(in_deg_sequence), len(out_deg_sequence)
    maxn = max(nin, nout)
    maxin = 0
    if maxn == 0:
        return True
    stubheap, zeroheap = [], []
    for n in range(maxn):
        in_deg, out_deg = 0, 0
        if n < nout:
            out_deg = out_deg_sequence[n]
        if n < nin:
            in_deg = in_deg_sequence[n]
```

```
351: (8)                    if in_deg < 0 or out_deg < 0:
352: (12)                       return False
353: (8)                    sumin, sumout, maxin = sumin + in_deg, sumout + out_deg, max(maxin,
in_deg)
354: (8)                    if in_deg > 0:
355: (12)                       stubheap.append((-1 * out_deg, -1 * in_deg))
356: (8)                    elif out_deg > 0:
357: (12)                       zeroheap.append(-1 * out_deg)
358: (4)                if sumin != sumout:
359: (8)                    return False
360: (4)                heapq.heapify(stubheap)
361: (4)                heapq.heapify(zeroheap)
362: (4)                modstubs = [(0, 0)] * (maxin + 1)
363: (4)                while stubheap:
364: (8)                    (freeout, freein) = heapq.heappop(stubheap)
365: (8)                    freein *= -1
366: (8)                    if freein > len(stubheap) + len(zeroheap):
367: (12)                       return False
368: (8)                    mslen = 0
369: (8)                    for i in range(freein):
370: (12)                       if zeroheap and (not stubheap or stubheap[0][0] > zeroheap[0]):
371: (16)                           stubout = heapq.heappop(zeroheap)
372: (16)                           stubin = 0
373: (12)                       else:
374: (16)                           (stubout, stubin) = heapq.heappop(stubheap)
375: (12)                       if stubout == 0:
376: (16)                           return False
377: (12)                       if stubout + 1 < 0 or stubin < 0:
378: (16)                           modstubs[mslen] = (stubout + 1, stubin)
379: (16)                           mslen += 1
380: (8)                    for i in range(mslen):
381: (12)                       stub = modstubs[i]
382: (12)                       if stub[1] < 0:
383: (16)                           heapq.heappush(stubheap, stub)
384: (12)                       else:
385: (16)                           heapq.heappush(zeroheap, stub[0])
386: (8)                    if freeout < 0:
387: (12)                       heapq.heappush(zeroheap, freeout)
388: (4)                return True
```

File 32 - hierarchy.py:

```
1: (0)              """
2: (0)              Flow Hierarchy.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = ["flow_hierarchy"]
6: (0)              @nx._dispatchable(edge_attrs="weight")
7: (0)              def flow_hierarchy(G, weight=None):
8: (4)                  """Returns the flow hierarchy of a directed network.
9: (4)                  Flow hierarchy is defined as the fraction of edges not participating
10: (4)                 in cycles in a directed graph [1]_.
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 G : DiGraph or MultiDiGraph
14: (7)                    A directed graph
15: (4)                 weight : string, optional (default=None)
16: (7)                    Attribute to use for edge weights. If None the weight defaults to 1.
17: (4)                 Returns
18: (4)                 -------
19: (4)                 h : float
20: (7)                    Flow hierarchy value
21: (4)                 Notes
22: (4)                 -----
23: (4)                 The algorithm described in [1]_ computes the flow hierarchy through
24: (4)                 exponentiation of the adjacency matrix.  This function implements an
25: (4)                 alternative approach that finds strongly connected components.
```

```
26: (4)                 An edge is in a cycle if and only if it is in a strongly connected
27: (4)                 component, which can be found in $O(m)$ time using Tarjan's algorithm.
28: (4)                 References
29: (4)                 ----------
30: (4)                 .. [1] Luo, J.; Magee, C.L. (2011),
31: (7)                    Detecting evolving patterns of self-organizing networks by flow
32: (7)                    hierarchy measurement, Complexity, Volume 16 Issue 6 53-61.
33: (7)                    DOI: 10.1002/cplx.20368
34: (7)                    http://web.mit.edu/~cmagee/www/documents/28-
DetectingEvolvingPatterns_FlowHierarchy.pdf
35: (4)                 """
36: (4)                 if not G.is_directed():
37: (8)                     raise nx.NetworkXError("G must be a digraph in flow_hierarchy")
38: (4)                 scc = nx.strongly_connected_components(G)
39: (4)                 return 1 - sum(G.subgraph(c).size(weight) for c in scc) / G.size(weight)
```

File 33 - hybrid.py:

```
1: (0)              """
2: (0)              Provides functions for finding and testing for locally `(k, l)`-connected
3: (0)              graphs.
4: (0)              """
5: (0)              import copy
6: (0)              import networkx as nx
7: (0)              __all__ = ["kl_connected_subgraph", "is_kl_connected"]
8: (0)              @nx._dispatchable(returns_graph=True)
9: (0)              def kl_connected_subgraph(G, k, l, low_memory=False, same_as_graph=False):
10: (4)                 """Returns the maximum locally `(k, l)`-connected subgraph of `G`.
11: (4)                 A graph is locally `(k, l)`-connected if for each edge `(u, v)` in the
12: (4)                 graph there are at least `l` edge-disjoint paths of length at most `k`
13: (4)                 joining `u` to `v`.
14: (4)                 Parameters
15: (4)                 ----------
16: (4)                 G : NetworkX graph
17: (8)                     The graph in which to find a maximum locally `(k, l)`-connected
18: (8)                     subgraph.
19: (4)                 k : integer
20: (8)                     The maximum length of paths to consider. A higher number means a
looser
21: (8)                     connectivity requirement.
22: (4)                 l : integer
23: (8)                     The number of edge-disjoint paths. A higher number means a stricter
24: (8)                     connectivity requirement.
25: (4)                 low_memory : bool
26: (8)                     If this is True, this function uses an algorithm that uses slightly
27: (8)                     more time but less memory.
28: (4)                 same_as_graph : bool
29: (8)                     If True then return a tuple of the form `(H, is_same)`,
30: (8)                     where `H` is the maximum locally `(k, l)`-connected subgraph and
31: (8)                     `is_same` is a Boolean representing whether `G` is locally `(k,
32: (8)                     l)`-connected (and hence, whether `H` is simply a copy of the input
33: (8)                     graph `G`).
34: (4)                 Returns
35: (4)                 -------
36: (4)                 NetworkX graph or two-tuple
37: (8)                     If `same_as_graph` is True, then this function returns a
38: (8)                     two-tuple as described above. Otherwise, it returns only the maximum
39: (8)                     locally `(k, l)`-connected subgraph.
40: (4)                 See also
41: (4)                 --------
42: (4)                 is_kl_connected
43: (4)                 References
44: (4)                 ----------
45: (4)                 .. [1] Chung, Fan and Linyuan Lu. "The Small World Phenomenon in Hybrid
46: (11)                       Power Law Graphs." *Complex Networks*. Springer Berlin Heidelberg,
47: (11)                       2004. 89--104.
48: (4)                 """
```

```
 49: (4)                H = copy.deepcopy(G)  # subgraph we construct by removing from G
 50: (4)                graphOK = True
 51: (4)                deleted_some = True  # hack to start off the while loop
 52: (4)                while deleted_some:
 53: (8)                    deleted_some = False
 54: (8)                    for edge in list(H.edges()):
 55: (12)                       (u, v) = edge
 56: (12)                       if low_memory:
 57: (16)                           verts = {u, v}
 58: (16)                           for i in range(k):
 59: (20)                               for w in verts.copy():
 60: (24)                                   verts.update(G[w])
 61: (16)                           G2 = G.subgraph(verts).copy()
 62: (12)                       else:
 63: (16)                           G2 = copy.deepcopy(G)
 64: (12)                       path = [u, v]
 65: (12)                       cnt = 0
 66: (12)                       accept = 0
 67: (12)                       while path:
 68: (16)                           cnt += 1  # Found a path
 69: (16)                           if cnt >= l:
 70: (20)                               accept = 1
 71: (20)                               break
 72: (16)                           prev = u
 73: (16)                           for w in path:
 74: (20)                               if prev != w:
 75: (24)                                   G2.remove_edge(prev, w)
 76: (24)                                   prev = w
 77: (16)                           try:
 78: (20)                               path = nx.shortest_path(G2, u, v)  # ??? should "Cutoff"
be k+1?
 79: (16)                           except nx.NetworkXNoPath:
 80: (20)                               path = False
 81: (12)                       if accept == 0:
 82: (16)                           H.remove_edge(u, v)
 83: (16)                           deleted_some = True
 84: (16)                           if graphOK:
 85: (20)                               graphOK = False
 86: (4)                if same_as_graph:
 87: (8)                    return (H, graphOK)
 88: (4)                return H
 89: (0)            @nx._dispatchable
 90: (0)            def is_kl_connected(G, k, l, low_memory=False):
 91: (4)                """Returns True if and only if `G` is locally `(k, l)`-connected.
 92: (4)                A graph is locally `(k, l)`-connected if for each edge `(u, v)` in the
 93: (4)                graph there are at least `l` edge-disjoint paths of length at most `k`
 94: (4)                joining `u` to `v`.
 95: (4)                Parameters
 96: (4)                ----------
 97: (4)                G : NetworkX graph
 98: (8)                    The graph to test for local `(k, l)`-connectedness.
 99: (4)                k : integer
100: (8)                    The maximum length of paths to consider. A higher number means a
looser
101: (8)                    connectivity requirement.
102: (4)                l : integer
103: (8)                    The number of edge-disjoint paths. A higher number means a stricter
104: (8)                    connectivity requirement.
105: (4)                low_memory : bool
106: (8)                    If this is True, this function uses an algorithm that uses slightly
107: (8)                    more time but less memory.
108: (4)                Returns
109: (4)                -------
110: (4)                bool
111: (8)                    Whether the graph is locally `(k, l)`-connected subgraph.
112: (4)                See also
113: (4)                --------
114: (4)                kl_connected_subgraph
115: (4)                References
```

```
116: (4)                ----------
117: (4)                .. [1] Chung, Fan and Linyuan Lu. "The Small World Phenomenon in Hybrid
118: (11)                      Power Law Graphs." *Complex Networks*. Springer Berlin Heidelberg,
119: (11)                      2004. 89--104.
120: (4)                """
121: (4)                graphOK = True
122: (4)                for edge in G.edges():
123: (8)                    (u, v) = edge
124: (8)                    if low_memory:
125: (12)                       verts = {u, v}
126: (12)                       for i in range(k):
127: (16)                           [verts.update(G.neighbors(w)) for w in verts.copy()]
128: (12)                       G2 = G.subgraph(verts)
129: (8)                    else:
130: (12)                       G2 = copy.deepcopy(G)
131: (8)                    path = [u, v]
132: (8)                    cnt = 0
133: (8)                    accept = 0
134: (8)                    while path:
135: (12)                       cnt += 1  # Found a path
136: (12)                       if cnt >= l:
137: (16)                           accept = 1
138: (16)                           break
139: (12)                       prev = u
140: (12)                       for w in path:
141: (16)                           if w != prev:
142: (20)                               G2.remove_edge(prev, w)
143: (20)                               prev = w
144: (12)                       try:
145: (16)                           path = nx.shortest_path(G2, u, v)  # ??? should "Cutoff" be
k+1?
146: (12)                       except nx.NetworkXNoPath:
147: (16)                           path = False
148: (8)                    if accept == 0:
149: (12)                       graphOK = False
150: (12)                       break
151: (4)                return graphOK

----------------------------------------

File 34 - isolate.py:

1: (0)              """
2: (0)              Functions for identifying isolate (degree zero) nodes.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = ["is_isolate", "isolates", "number_of_isolates"]
6: (0)              @nx._dispatchable
7: (0)              def is_isolate(G, n):
8: (4)                  """Determines whether a node is an isolate.
9: (4)                  An *isolate* is a node with no neighbors (that is, with degree
10: (4)                 zero). For directed graphs, this means no in-neighbors and no
11: (4)                 out-neighbors.
12: (4)                 Parameters
13: (4)                 ----------
14: (4)                 G : NetworkX graph
15: (4)                 n : node
16: (8)                     A node in `G`.
17: (4)                 Returns
18: (4)                 -------
19: (4)                 is_isolate : bool
20: (7)                    True if and only if `n` has no neighbors.
21: (4)                 Examples
22: (4)                 --------
23: (4)                 >>> G = nx.Graph()
24: (4)                 >>> G.add_edge(1, 2)
25: (4)                 >>> G.add_node(3)
26: (4)                 >>> nx.is_isolate(G, 2)
27: (4)                 False
```

```
28: (4)                 >>> nx.is_isolate(G, 3)
29: (4)                 True
30: (4)                 """
31: (4)                 return G.degree(n) == 0
32: (0)             @nx._dispatchable
33: (0)             def isolates(G):
34: (4)                 """Iterator over isolates in the graph.
35: (4)                 An *isolate* is a node with no neighbors (that is, with degree
36: (4)                 zero). For directed graphs, this means no in-neighbors and no
37: (4)                 out-neighbors.
38: (4)                 Parameters
39: (4)                 ----------
40: (4)                 G : NetworkX graph
41: (4)                 Returns
42: (4)                 -------
43: (4)                 iterator
44: (8)                     An iterator over the isolates of `G`.
45: (4)                 Examples
46: (4)                 --------
47: (4)                 To get a list of all isolates of a graph, use the :class:`list`
48: (4)                 constructor::
49: (8)                     >>> G = nx.Graph()
50: (8)                     >>> G.add_edge(1, 2)
51: (8)                     >>> G.add_node(3)
52: (8)                     >>> list(nx.isolates(G))
53: (8)                     [3]
54: (4)                 To remove all isolates in the graph, first create a list of the
55: (4)                 isolates, then use :meth:`Graph.remove_nodes_from`::
56: (8)                     >>> G.remove_nodes_from(list(nx.isolates(G)))
57: (8)                     >>> list(G)
58: (8)                     [1, 2]
59: (4)                 For digraphs, isolates have zero in-degree and zero out_degre::
60: (8)                     >>> G = nx.DiGraph([(0, 1), (1, 2)])
61: (8)                     >>> G.add_node(3)
62: (8)                     >>> list(nx.isolates(G))
63: (8)                     [3]
64: (4)                 """
65: (4)                 return (n for n, d in G.degree() if d == 0)
66: (0)             @nx._dispatchable
67: (0)             def number_of_isolates(G):
68: (4)                 """Returns the number of isolates in the graph.
69: (4)                 An *isolate* is a node with no neighbors (that is, with degree
70: (4)                 zero). For directed graphs, this means no in-neighbors and no
71: (4)                 out-neighbors.
72: (4)                 Parameters
73: (4)                 ----------
74: (4)                 G : NetworkX graph
75: (4)                 Returns
76: (4)                 -------
77: (4)                 int
78: (8)                     The number of degree zero nodes in the graph `G`.
79: (4)                 """
80: (4)                 return sum(1 for v in isolates(G))
```

File 35 - link_prediction.py:

```
1: (0)              """
2: (0)              Link prediction algorithms.
3: (0)              """
4: (0)              from math import log
5: (0)              import networkx as nx
6: (0)              from networkx.utils import not_implemented_for
7: (0)              __all__ = [
8: (4)                  "resource_allocation_index",
9: (4)                  "jaccard_coefficient",
10: (4)                 "adamic_adar_index",
11: (4)                 "preferential_attachment",
```

```
(4)                 "cn_soundarajan_hopcroft",
(4)                 "ra_index_soundarajan_hopcroft",
(4)                 "within_inter_cluster",
(4)                 "common_neighbor_centrality",
(0)             ]
(0)             def _apply_prediction(G, func, ebunch=None):
(4)                 """Applies the given function to each edge in the specified iterable
(4)                 of edges.
(4)                 `G` is an instance of :class:`networkx.Graph`.
(4)                 `func` is a function on two inputs, each of which is a node in the
(4)                 graph. The function can return anything, but it should return a
(4)                 value representing a prediction of the likelihood of a "link"
(4)                 joining the two nodes.
(4)                 `ebunch` is an iterable of pairs of nodes. If not specified, all
(4)                 non-edges in the graph `G` will be used.
(4)                 """
(4)                 if ebunch is None:
(8)                     ebunch = nx.non_edges(G)
(4)                 else:
(8)                     for u, v in ebunch:
(12)                        if u not in G:
(16)                            raise nx.NodeNotFound(f"Node {u} not in G.")
(12)                        if v not in G:
(16)                            raise nx.NodeNotFound(f"Node {v} not in G.")
(4)                 return ((u, v, func(u, v)) for u, v in ebunch)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable
(0)             def resource_allocation_index(G, ebunch=None):
(4)                 r"""Compute the resource allocation index of all node pairs in ebunch.
(4)                 Resource allocation index of `u` and `v` is defined as
(4)                 .. math::
(8)                     \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{|\Gamma(w)|}
(4)                 where $\Gamma(u)$ denotes the set of neighbors of $u$.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX undirected graph.
(4)                 ebunch : iterable of node pairs, optional (default = None)
(8)                     Resource allocation index will be computed for each pair of
(8)                     nodes given in the iterable. The pairs must be given as
(8)                     2-tuples (u, v) where u and v are nodes in the graph. If ebunch
(8)                     is None then all nonexistent edges in the graph will be used.
(8)                     Default value: None.
(4)                 Returns
(4)                 -------
(4)                 piter : iterator
(8)                     An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
(8)                     pair of nodes and p is their resource allocation index.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
(4)                 NodeNotFound
(8)                     If `ebunch` has a node that is not in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.complete_graph(5)
(4)                 >>> preds = nx.resource_allocation_index(G, [(0, 1), (2, 3)])
(4)                 >>> for u, v, p in preds:
(4)                 ...     print(f"({u}, {v}) -> {p:.8f}")
(4)                 (0, 1) -> 0.75000000
(4)                 (2, 3) -> 0.75000000
(4)                 References
(4)                 ----------
(4)                 .. [1] T. Zhou, L. Lu, Y.-C. Zhang.
(7)                    Predicting missing links via local information.
(7)                    Eur. Phys. J. B 71 (2009) 623.
(7)                    https://arxiv.org/pdf/0901.0553.pdf
```

```
    """
    def predict(u, v):
        return sum(1 / G.degree(w) for w in nx.common_neighbors(G, u, v))
    return _apply_prediction(G, predict, ebunch)
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable
def jaccard_coefficient(G, ebunch=None):
    r"""Compute the Jaccard coefficient of all node pairs in ebunch.
    Jaccard coefficient of nodes `u` and `v` is defined as
    .. math::
        \frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|}
    where $\Gamma(u)$ denotes the set of neighbors of $u$.
    Parameters
    ----------
    G : graph
        A NetworkX undirected graph.
    ebunch : iterable of node pairs, optional (default = None)
        Jaccard coefficient will be computed for each pair of nodes
        given in the iterable. The pairs must be given as 2-tuples
        (u, v) where u and v are nodes in the graph. If ebunch is None
        then all nonexistent edges in the graph will be used.
        Default value: None.
    Returns
    -------
    piter : iterator
        An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
        pair of nodes and p is their Jaccard coefficient.
    Raises
    ------
    NetworkXNotImplemented
        If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
    NodeNotFound
        If `ebunch` has a node that is not in `G`.
    Examples
    --------
    >>> G = nx.complete_graph(5)
    >>> preds = nx.jaccard_coefficient(G, [(0, 1), (2, 3)])
    >>> for u, v, p in preds:
    ...     print(f"({u}, {v}) -> {p:.8f}")
    (0, 1) -> 0.60000000
    (2, 3) -> 0.60000000
    References
    ----------
    .. [1] D. Liben-Nowell, J. Kleinberg.
           The Link Prediction Problem for Social Networks (2004).
           http://www.cs.cornell.edu/home/kleinber/link-pred.pdf
    """
    def predict(u, v):
        union_size = len(set(G[u]) | set(G[v]))
        if union_size == 0:
            return 0
        return len(nx.common_neighbors(G, u, v)) / union_size
    return _apply_prediction(G, predict, ebunch)
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable
def adamic_adar_index(G, ebunch=None):
    r"""Compute the Adamic-Adar index of all node pairs in ebunch.
    Adamic-Adar index of `u` and `v` is defined as
    .. math::
        \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(w)|}
    where $\Gamma(u)$ denotes the set of neighbors of $u$.
    This index leads to zero-division for nodes only connected via self-loops.
    It is intended to be used when no self-loops are present.
    Parameters
    ----------
    G : graph
        NetworkX undirected graph.
```

```
    ebunch : iterable of node pairs, optional (default = None)
        Adamic-Adar index will be computed for each pair of nodes given
        in the iterable. The pairs must be given as 2-tuples (u, v)
        where u and v are nodes in the graph. If ebunch is None then all
        nonexistent edges in the graph will be used.
        Default value: None.
    Returns
    -------
    piter : iterator
        An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
        pair of nodes and p is their Adamic-Adar index.
    Raises
    ------
    NetworkXNotImplemented
        If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
    NodeNotFound
        If `ebunch` has a node that is not in `G`.
    Examples
    --------
    >>> G = nx.complete_graph(5)
    >>> preds = nx.adamic_adar_index(G, [(0, 1), (2, 3)])
    >>> for u, v, p in preds:
    ...     print(f"({u}, {v}) -> {p:.8f}")
    (0, 1) -> 2.16404256
    (2, 3) -> 2.16404256
    References
    ----------
    .. [1] D. Liben-Nowell, J. Kleinberg.
           The Link Prediction Problem for Social Networks (2004).
           http://www.cs.cornell.edu/home/kleinber/link-pred.pdf
    """
    def predict(u, v):
        return sum(1 / log(G.degree(w)) for w in nx.common_neighbors(G, u, v))
    return _apply_prediction(G, predict, ebunch)
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable
def common_neighbor_centrality(G, ebunch=None, alpha=0.8):
    r"""Return the CCPA score for each pair of nodes.
    Compute the Common Neighbor and Centrality based Parameterized Algorithm(CCPA)
    score of all node pairs in ebunch.
    CCPA score of `u` and `v` is defined as
    .. math::
        \alpha \cdot (|\Gamma (u){\cap }^{}\Gamma (v)|)+(1-\alpha )\cdot \frac{N}{{d}_{uv}}
    where $\Gamma(u)$ denotes the set of neighbors of $u$, $\Gamma(v)$ denotes the
    set of neighbors of $v$, $\alpha$ is  parameter varies between [0,1], $N$ denotes
    total number of nodes in the Graph and ${d}_{uv}$ denotes shortest distance
    between $u$ and $v$.
    This algorithm is based on two vital properties of nodes, namely the number
    of common neighbors and their centrality. Common neighbor refers to the common
    nodes between two nodes. Centrality refers to the prestige that a node enjoys
    in a network.
    .. seealso::
        :func:`common_neighbors`
    Parameters
    ----------
    G : graph
        NetworkX undirected graph.
    ebunch : iterable of node pairs, optional (default = None)
        Preferential attachment score will be computed for each pair of
        nodes given in the iterable. The pairs must be given as
```

```
(8)                 2-tuples (u, v) where u and v are nodes in the graph. If ebunch
(8)                 is None then all nonexistent edges in the graph will be used.
(8)                 Default value: None.
(4)             alpha : Parameter defined for participation of Common Neighbor
(12)                    and Centrality Algorithm share. Values for alpha should
(12)                    normally be between 0 and 1. Default value set to 0.8
(12)                    because author found better performance at 0.8 for all the
(12)                    dataset.
(12)                    Default value: 0.8
(4)             Returns
(4)             -------
(4)             piter : iterator
(8)                 An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
(8)                 pair of nodes and p is their Common Neighbor and Centrality based
(8)                 Parameterized Algorithm(CCPA) score.
(4)             Raises
(4)             ------
(4)             NetworkXNotImplemented
(8)                 If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
(4)             NetworkXAlgorithmError
(8)                 If self loops exsists in `ebunch` or in `G` (if `ebunch` is `None`).
(4)             NodeNotFound
(8)                 If `ebunch` has a node that is not in `G`.
(4)             Examples
(4)             --------
(4)             >>> G = nx.complete_graph(5)
(4)             >>> preds = nx.common_neighbor_centrality(G, [(0, 1), (2, 3)])
(4)             >>> for u, v, p in preds:
(4)             ...     print(f"({u}, {v}) -> {p}")
(4)             (0, 1) -> 3.4000000000000004
(4)             (2, 3) -> 3.4000000000000004
(4)             References
(4)             ----------
(4)             .. [1] Ahmad, I., Akhtar, M.U., Noor, S. et al.
(11)                   Missing Link Prediction using Common Neighbor and Centrality based
Parameterized Algorithm.
(11)                   Sci Rep 10, 364 (2020).
(11)                   https://doi.org/10.1038/s41598-019-57304-y
(4)             """
(4)             if alpha == 1:
(8)                 def predict(u, v):
(12)                    if u == v:
(16)                        raise nx.NetworkXAlgorithmError("Self loops are not
supported")
(12)                    return len(nx.common_neighbors(G, u, v))
(4)             else:
(8)                 spl = dict(nx.shortest_path_length(G))
(8)                 inf = float("inf")
(8)                 def predict(u, v):
(12)                    if u == v:
(16)                        raise nx.NetworkXAlgorithmError("Self loops are not
supported")
(12)                    path_len = spl[u].get(v, inf)
(12)                    n_nbrs = len(nx.common_neighbors(G, u, v))
(12)                    return alpha * n_nbrs + (1 - alpha) * len(G) / path_len
(4)             return _apply_prediction(G, predict, ebunch)
(0)         @not_implemented_for("directed")
(0)         @not_implemented_for("multigraph")
(0)         @nx._dispatchable
(0)         def preferential_attachment(G, ebunch=None):
(4)             r"""Compute the preferential attachment score of all node pairs in ebunch.
(4)             Preferential attachment score of `u` and `v` is defined as
(4)             .. math::
(8)                 |\Gamma(u)| |\Gamma(v)|
(4)             where $\Gamma(u)$ denotes the set of neighbors of $u$.
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 NetworkX undirected graph.
```

```
(4)                 ebunch : iterable of node pairs, optional (default = None)
(8)                     Preferential attachment score will be computed for each pair of
(8)                     nodes given in the iterable. The pairs must be given as
(8)                     2-tuples (u, v) where u and v are nodes in the graph. If ebunch
(8)                     is None then all nonexistent edges in the graph will be used.
(8)                     Default value: None.
(4)                 Returns
(4)                 -------
(4)                 piter : iterator
(8)                     An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
(8)                     pair of nodes and p is their preferential attachment score.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
(4)                 NodeNotFound
(8)                     If `ebunch` has a node that is not in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.complete_graph(5)
(4)                 >>> preds = nx.preferential_attachment(G, [(0, 1), (2, 3)])
(4)                 >>> for u, v, p in preds:
(4)                 ...     print(f"({u}, {v}) -> {p}")
(4)                 (0, 1) -> 16
(4)                 (2, 3) -> 16
(4)                 References
(4)                 ----------
(4)                 .. [1] D. Liben-Nowell, J. Kleinberg.
(11)                       The Link Prediction Problem for Social Networks (2004).
(11)                       http://www.cs.cornell.edu/home/kleinber/link-pred.pdf
(4)                 """
(4)                 def predict(u, v):
(8)                     return G.degree(u) * G.degree(v)
(4)                 return _apply_prediction(G, predict, ebunch)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(node_attrs="community")
(0)             def cn_soundarajan_hopcroft(G, ebunch=None, community="community"):
(4)                 r"""Count the number of common neighbors of all node pairs in ebunch
(8)                     using community information.
(4)                 For two nodes $u$ and $v$, this function computes the number of
(4)                 common neighbors and bonus one for each common neighbor belonging to
(4)                 the same community as $u$ and $v$. Mathematically,
(4)                 .. math::
(8)                     |\Gamma(u) \cap \Gamma(v)| + \sum_{w \in \Gamma(u) \cap \Gamma(v)}
f(w)
(4)                 where $f(w)$ equals 1 if $w$ belongs to the same community as $u$
(4)                 and $v$ or 0 otherwise and $\Gamma(u)$ denotes the set of
(4)                 neighbors of $u$.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX undirected graph.
(4)                 ebunch : iterable of node pairs, optional (default = None)
(8)                     The score will be computed for each pair of nodes given in the
(8)                     iterable. The pairs must be given as 2-tuples (u, v) where u
(8)                     and v are nodes in the graph. If ebunch is None then all
(8)                     nonexistent edges in the graph will be used.
(8)                     Default value: None.
(4)                 community : string, optional (default = 'community')
(8)                     Nodes attribute name containing the community information.
(8)                     G[u][community] identifies which community u belongs to. Each
(8)                     node belongs to at most one community. Default value: 'community'.
(4)                 Returns
(4)                 -------
(4)                 piter : iterator
(8)                     An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
(8)                     pair of nodes and p is their score.
(4)                 Raises
```

```
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
(4)                 NetworkXAlgorithmError
(8)                     If no community information is available for a node in `ebunch` or in `G` (if `ebunch` is `None`).
(4)                 NodeNotFound
(8)                     If `ebunch` has a node that is not in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(3)
(4)                 >>> G.nodes[0]["community"] = 0
(4)                 >>> G.nodes[1]["community"] = 0
(4)                 >>> G.nodes[2]["community"] = 0
(4)                 >>> preds = nx.cn_soundarajan_hopcroft(G, [(0, 2)])
(4)                 >>> for u, v, p in preds:
(4)                 ...     print(f"({u}, {v}) -> {p}")
(4)                 (0, 2) -> 2
(4)                 References
(4)                 ----------
(4)                 .. [1] Sucheta Soundarajan and John Hopcroft.
(7)                    Using community information to improve the precision of link
(7)                    prediction methods.
(7)                    In Proceedings of the 21st international conference companion on
(7)                    World Wide Web (WWW '12 Companion). ACM, New York, NY, USA, 607-608.
(7)                    http://doi.acm.org/10.1145/2187980.2188150
(4)                 """
(4)                 def predict(u, v):
(8)                     Cu = _community(G, u, community)
(8)                     Cv = _community(G, v, community)
(8)                     cnbors = nx.common_neighbors(G, u, v)
(8)                     neighbors = (
(12)                        sum(_community(G, w, community) == Cu for w in cnbors) if Cu == Cv else 0
(8)                     )
(8)                     return len(cnbors) + neighbors
(4)                 return _apply_prediction(G, predict, ebunch)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(node_attrs="community")
(0)             def ra_index_soundarajan_hopcroft(G, ebunch=None, community="community"):
(4)                 r"""Compute the resource allocation index of all node pairs in
(4)                 ebunch using community information.
(4)                 For two nodes $u$ and $v$, this function computes the resource
(4)                 allocation index considering only common neighbors belonging to the
(4)                 same community as $u$ and $v$. Mathematically,
(4)                 .. math::
(8)                     \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{f(w)}{|\Gamma(w)|}
(4)                 where $f(w)$ equals 1 if $w$ belongs to the same community as $u$
(4)                 and $v$ or 0 otherwise and $\Gamma(u)$ denotes the set of
(4)                 neighbors of $u$.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX undirected graph.
(4)                 ebunch : iterable of node pairs, optional (default = None)
(8)                     The score will be computed for each pair of nodes given in the
(8)                     iterable. The pairs must be given as 2-tuples (u, v) where u
(8)                     and v are nodes in the graph. If ebunch is None then all
(8)                     nonexistent edges in the graph will be used.
(8)                     Default value: None.
(4)                 community : string, optional (default = 'community')
(8)                     Nodes attribute name containing the community information.
(8)                     G[u][community] identifies which community u belongs to. Each
(8)                     node belongs to at most one community. Default value: 'community'.
(4)                 Returns
(4)                 -------
(4)                 piter : iterator
(8)                     An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
```

```
(8)                     pair of nodes and p is their score.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
(4)                 NetworkXAlgorithmError
(8)                     If no community information is available for a node in `ebunch` or in
`G` (if `ebunch` is `None`).
(4)                 NodeNotFound
(8)                     If `ebunch` has a node that is not in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(4)                 >>> G.nodes[0]["community"] = 0
(4)                 >>> G.nodes[1]["community"] = 0
(4)                 >>> G.nodes[2]["community"] = 1
(4)                 >>> G.nodes[3]["community"] = 0
(4)                 >>> preds = nx.ra_index_soundarajan_hopcroft(G, [(0, 3)])
(4)                 >>> for u, v, p in preds:
(4)                 ...     print(f"({u}, {v}) -> {p:.8f}")
(4)                 (0, 3) -> 0.50000000
(4)                 References
(4)                 ----------
(4)                 .. [1] Sucheta Soundarajan and John Hopcroft.
(7)                    Using community information to improve the precision of link
(7)                    prediction methods.
(7)                    In Proceedings of the 21st international conference companion on
(7)                    World Wide Web (WWW '12 Companion). ACM, New York, NY, USA, 607-608.
(7)                    http://doi.acm.org/10.1145/2187980.2188150
(4)                 """
(4)                 def predict(u, v):
(8)                     Cu = _community(G, u, community)
(8)                     Cv = _community(G, v, community)
(8)                     if Cu != Cv:
(12)                        return 0
(8)                     cnbors = nx.common_neighbors(G, u, v)
(8)                     return sum(1 / G.degree(w) for w in cnbors if _community(G, w,
community) == Cu)
(4)                 return _apply_prediction(G, predict, ebunch)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(node_attrs="community")
(0)             def within_inter_cluster(G, ebunch=None, delta=0.001, community="community"):
(4)                 """Compute the ratio of within- and inter-cluster common neighbors
(4)                 of all node pairs in ebunch.
(4)                 For two nodes `u` and `v`, if a common neighbor `w` belongs to the
(4)                 same community as them, `w` is considered as within-cluster common
(4)                 neighbor of `u` and `v`. Otherwise, it is considered as
(4)                 inter-cluster common neighbor of `u` and `v`. The ratio between the
(4)                 size of the set of within- and inter-cluster common neighbors is
(4)                 defined as the WIC measure. [1]_
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX undirected graph.
(4)                 ebunch : iterable of node pairs, optional (default = None)
(8)                     The WIC measure will be computed for each pair of nodes given in
(8)                     the iterable. The pairs must be given as 2-tuples (u, v) where
(8)                     u and v are nodes in the graph. If ebunch is None then all
(8)                     nonexistent edges in the graph will be used.
(8)                     Default value: None.
(4)                 delta : float, optional (default = 0.001)
(8)                     Value to prevent division by zero in case there is no
(8)                     inter-cluster common neighbor between two nodes. See [1]_ for
(8)                     details. Default value: 0.001.
(4)                 community : string, optional (default = 'community')
(8)                     Nodes attribute name containing the community information.
(8)                     G[u][community] identifies which community u belongs to. Each
```

```
479: (8)                    node belongs to at most one community. Default value: 'community'.
480: (4)                Returns
481: (4)                -------
482: (4)                piter : iterator
483: (8)                    An iterator of 3-tuples in the form (u, v, p) where (u, v) is a
484: (8)                    pair of nodes and p is their WIC measure.
485: (4)                Raises
486: (4)                ------
487: (4)                NetworkXNotImplemented
488: (8)                    If `G` is a `DiGraph`, a `Multigraph` or a `MultiDiGraph`.
489: (4)                NetworkXAlgorithmError
490: (8)                    - If `delta` is less than or equal to zero.
491: (8)                    - If no community information is available for a node in `ebunch` or
in `G` (if `ebunch` is `None`).
492: (4)                NodeNotFound
493: (8)                    If `ebunch` has a node that is not in `G`.
494: (4)                Examples
495: (4)                --------
496: (4)                >>> G = nx.Graph()
497: (4)                >>> G.add_edges_from([(0, 1), (0, 2), (0, 3), (1, 4), (2, 4), (3, 4)])
498: (4)                >>> G.nodes[0]["community"] = 0
499: (4)                >>> G.nodes[1]["community"] = 1
500: (4)                >>> G.nodes[2]["community"] = 0
501: (4)                >>> G.nodes[3]["community"] = 0
502: (4)                >>> G.nodes[4]["community"] = 0
503: (4)                >>> preds = nx.within_inter_cluster(G, [(0, 4)])
504: (4)                >>> for u, v, p in preds:
505: (4)                ...     print(f"({u}, {v}) -> {p:.8f}")
506: (4)                (0, 4) -> 1.99800200
507: (4)                >>> preds = nx.within_inter_cluster(G, [(0, 4)], delta=0.5)
508: (4)                >>> for u, v, p in preds:
509: (4)                ...     print(f"({u}, {v}) -> {p:.8f}")
510: (4)                (0, 4) -> 1.33333333
511: (4)                References
512: (4)                ----------
513: (4)                .. [1] Jorge Carlos Valverde-Rebaza and Alneu de Andrade Lopes.
514: (7)                   Link prediction in complex networks based on cluster information.
515: (7)                   In Proceedings of the 21st Brazilian conference on Advances in
516: (7)                   Artificial Intelligence (SBIA'12)
517: (7)                   https://doi.org/10.1007/978-3-642-34459-6_10
518: (4)                """
519: (4)                if delta <= 0:
520: (8)                    raise nx.NetworkXAlgorithmError("Delta must be greater than zero")
521: (4)                def predict(u, v):
522: (8)                    Cu = _community(G, u, community)
523: (8)                    Cv = _community(G, v, community)
524: (8)                    if Cu != Cv:
525: (12)                       return 0
526: (8)                    cnbors = nx.common_neighbors(G, u, v)
527: (8)                    within = {w for w in cnbors if _community(G, w, community) == Cu}
528: (8)                    inter = cnbors - within
529: (8)                    return len(within) / (len(inter) + delta)
530: (4)                return _apply_prediction(G, predict, ebunch)
531: (0)            def _community(G, u, community):
532: (4)                """Get the community of the given node."""
533: (4)                node_u = G.nodes[u]
534: (4)                try:
535: (8)                    return node_u[community]
536: (4)                except KeyError as err:
537: (8)                    raise nx.NetworkXAlgorithmError(
538: (12)                       f"No community information available for Node {u}"
539: (8)                    ) from err
```

File 36 - lowest_common_ancestors.py:

```
1: (0)              """Algorithms for finding the lowest common ancestor of trees and DAGs."""
2: (0)              from collections import defaultdict
```

```
3: (0)              from collections.abc import Mapping, Set
4: (0)              from itertools import combinations_with_replacement
5: (0)              import networkx as nx
6: (0)              from networkx.utils import UnionFind, arbitrary_element, not_implemented_for
7: (0)              __all__ = [
8: (4)                  "all_pairs_lowest_common_ancestor",
9: (4)                  "tree_all_pairs_lowest_common_ancestor",
10: (4)                 "lowest_common_ancestor",
11: (0)             ]
12: (0)             @not_implemented_for("undirected")
13: (0)             @nx._dispatchable
14: (0)             def all_pairs_lowest_common_ancestor(G, pairs=None):
15: (4)                 """Return the lowest common ancestor of all pairs or the provided pairs
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G : NetworkX directed graph
19: (4)                 pairs : iterable of pairs of nodes, optional (default: all pairs)
20: (8)                     The pairs of nodes of interest.
21: (8)                     If None, will find the LCA of all pairs of nodes.
22: (4)                 Yields
23: (4)                 ------
24: (4)                 ((node1, node2), lca) : 2-tuple
25: (8)                     Where lca is least common ancestor of node1 and node2.
26: (8)                     Note that for the default case, the order of the node pair is not
considered,
27: (8)                     e.g. you will not get both ``(a, b)`` and ``(b, a)``
28: (4)                 Raises
29: (4)                 ------
30: (4)                 NetworkXPointlessConcept
31: (8)                     If `G` is null.
32: (4)                 NetworkXError
33: (8)                     If `G` is not a DAG.
34: (4)                 Examples
35: (4)                 --------
36: (4)                 The default behavior is to yield the lowest common ancestor for all
37: (4)                 possible combinations of nodes in `G`, including self-pairings:
38: (4)                 >>> G = nx.DiGraph([(0, 1), (0, 3), (1, 2)])
39: (4)                 >>> dict(nx.all_pairs_lowest_common_ancestor(G))
40: (4)                 {(0, 0): 0, (0, 1): 0, (0, 3): 0, (0, 2): 0, (1, 1): 1, (1, 3): 0, (1, 2):
1, (3, 3): 3, (3, 2): 0, (2, 2): 2}
41: (4)                 The pairs argument can be used to limit the output to only the
42: (4)                 specified node pairings:
43: (4)                 >>> dict(nx.all_pairs_lowest_common_ancestor(G, pairs=[(1, 2), (2, 3)]))
44: (4)                 {(1, 2): 1, (2, 3): 0}
45: (4)                 Notes
46: (4)                 -----
47: (4)                 Only defined on non-null directed acyclic graphs.
48: (4)                 See Also
49: (4)                 --------
50: (4)                 lowest_common_ancestor
51: (4)                 """
52: (4)                 if not nx.is_directed_acyclic_graph(G):
53: (8)                     raise nx.NetworkXError("LCA only defined on directed acyclic graphs.")
54: (4)                 if len(G) == 0:
55: (8)                     raise nx.NetworkXPointlessConcept("LCA meaningless on null graphs.")
56: (4)                 if pairs is None:
57: (8)                     pairs = combinations_with_replacement(G, 2)
58: (4)                 else:
59: (8)                     pairs = dict.fromkeys(pairs)
60: (8)                     nodeset = set(G)
61: (8)                     for pair in pairs:
62: (12)                        if set(pair) - nodeset:
63: (16)                            raise nx.NodeNotFound(
64: (20)                                f"Node(s) {set(pair) - nodeset} from pair {pair} not in
G."
65: (16)                            )
66: (4)                 def generate_lca_from_pairs(G, pairs):
67: (8)                     ancestor_cache = {}
68: (8)                     for v, w in pairs:
```

```
69: (12)                        if v not in ancestor_cache:
70: (16)                            ancestor_cache[v] = nx.ancestors(G, v)
71: (16)                            ancestor_cache[v].add(v)
72: (12)                        if w not in ancestor_cache:
73: (16)                            ancestor_cache[w] = nx.ancestors(G, w)
74: (16)                            ancestor_cache[w].add(w)
75: (12)                        common_ancestors = ancestor_cache[v] & ancestor_cache[w]
76: (12)                        if common_ancestors:
77: (16)                            common_ancestor = next(iter(common_ancestors))
78: (16)                            while True:
79: (20)                                successor = None
80: (20)                                for lower_ancestor in G.successors(common_ancestor):
81: (24)                                    if lower_ancestor in common_ancestors:
82: (28)                                        successor = lower_ancestor
83: (28)                                        break
84: (20)                                if successor is None:
85: (24)                                    break
86: (20)                                common_ancestor = successor
87: (16)                            yield ((v, w), common_ancestor)
88: (4)                 return generate_lca_from_pairs(G, pairs)
89: (0)             @not_implemented_for("undirected")
90: (0)             @nx._dispatchable
91: (0)             def lowest_common_ancestor(G, node1, node2, default=None):
92: (4)                 """Compute the lowest common ancestor of the given pair of nodes.
93: (4)                 Parameters
94: (4)                 ----------
95: (4)                 G : NetworkX directed graph
96: (4)                 node1, node2 : nodes in the graph.
97: (4)                 default : object
98: (8)                     Returned if no common ancestor between `node1` and `node2`
99: (4)                 Returns
100: (4)                -------
101: (4)                The lowest common ancestor of node1 and node2,
102: (4)                or default if they have no common ancestors.
103: (4)                Examples
104: (4)                --------
105: (4)                >>> G = nx.DiGraph()
106: (4)                >>> nx.add_path(G, (0, 1, 2, 3))
107: (4)                >>> nx.add_path(G, (0, 4, 3))
108: (4)                >>> nx.lowest_common_ancestor(G, 2, 4)
109: (4)                0
110: (4)                See Also
111: (4)                --------
112: (4)                all_pairs_lowest_common_ancestor"""
113: (4)                ans = list(all_pairs_lowest_common_ancestor(G, pairs=[(node1, node2)]))
114: (4)                if ans:
115: (8)                    assert len(ans) == 1
116: (8)                    return ans[0][1]
117: (4)                return default
118: (0)            @not_implemented_for("undirected")
119: (0)            @nx._dispatchable
120: (0)            def tree_all_pairs_lowest_common_ancestor(G, root=None, pairs=None):
121: (4)                r"""Yield the lowest common ancestor for sets of pairs in a tree.
122: (4)                Parameters
123: (4)                ----------
124: (4)                G : NetworkX directed graph (must be a tree)
125: (4)                root : node, optional (default: None)
126: (8)                    The root of the subtree to operate on.
127: (8)                    If None, assume the entire graph has exactly one source and use that.
128: (4)                pairs : iterable or iterator of pairs of nodes, optional (default: None)
129: (8)                    The pairs of interest. If None, Defaults to all pairs of nodes
130: (8)                    under `root` that have a lowest common ancestor.
131: (4)                Returns
132: (4)                -------
133: (4)                lcas : generator of tuples `((u, v), lca)` where `u` and `v` are nodes
134: (8)                    in `pairs` and `lca` is their lowest common ancestor.
135: (4)                Examples
136: (4)                --------
137: (4)                >>> import pprint
```

```
    >>> G = nx.DiGraph([(1, 3), (2, 4), (1, 2)])
    >>> pprint.pprint(dict(nx.tree_all_pairs_lowest_common_ancestor(G)))
    {(1, 1): 1,
     (2, 1): 1,
     (2, 2): 2,
     (3, 1): 1,
     (3, 2): 1,
     (3, 3): 3,
     (3, 4): 1,
     (4, 1): 1,
     (4, 2): 2,
     (4, 4): 4}
    We can also use `pairs` argument to specify the pairs of nodes for which we
    want to compute lowest common ancestors. Here is an example:
    >>> dict(nx.tree_all_pairs_lowest_common_ancestor(G, pairs=[(1, 4), (2, 3)]))
    {(2, 3): 1, (1, 4): 1}
    Notes
    -----
    Only defined on non-null trees represented with directed edges from
    parents to children. Uses Tarjan's off-line lowest-common-ancestors
    algorithm. Runs in time $O(4 \times (V + E + P))$ time, where 4 is the largest
    value of the inverse Ackermann function likely to ever come up in actual
    use, and $P$ is the number of pairs requested (or $V^2$ if all are needed).
    Tarjan, R. E. (1979), "Applications of path compression on balanced trees",
    Journal of the ACM 26 (4): 690-715, doi:10.1145/322154.322161.
    See Also
    --------
    all_pairs_lowest_common_ancestor: similar routine for general DAGs
    lowest_common_ancestor: just a single pair for general DAGs
    """
    if len(G) == 0:
        raise nx.NetworkXPointlessConcept("LCA meaningless on null graphs.")
    if pairs is not None:
        pair_dict = defaultdict(set)
        if not isinstance(pairs, Mapping | Set):
            pairs = set(pairs)
        for u, v in pairs:
            for n in (u, v):
                if n not in G:
                    msg = f"The node {str(n)} is not in the digraph."
                    raise nx.NodeNotFound(msg)
            pair_dict[u].add(v)
            pair_dict[v].add(u)
    if root is None:
        for n, deg in G.in_degree:
            if deg == 0:
                if root is not None:
                    msg = "No root specified and tree has multiple sources."
                    raise nx.NetworkXError(msg)
                root = n
            elif deg > 1 and len(G.pred[n]) > 1:
                msg = "Tree LCA only defined on trees; use DAG routine."
                raise nx.NetworkXError(msg)
    if root is None:
        raise nx.NetworkXError("Graph contains a cycle.")
    uf = UnionFind()
    ancestors = {}
    for node in G:
        ancestors[node] = uf[node]
    colors = defaultdict(bool)
    for node in nx.dfs_postorder_nodes(G, root):
        colors[node] = True
        for v in pair_dict[node] if pairs is not None else G:
            if colors[v]:
```

```
202: (16)                            if pairs is not None and (node, v) in pairs:
203: (20)                                yield (node, v), ancestors[uf[v]]
204: (16)                            if pairs is None or (v, node) in pairs:
205: (20)                                yield (v, node), ancestors[uf[v]]
206: (8)                     if node != root:
207: (12)                        parent = arbitrary_element(G.pred[node])
208: (12)                        uf.union(parent, node)
209: (12)                        ancestors[uf[parent]] = parent
```

File 37 - matching.py:

```
1: (0)              """Functions for computing and verifying matchings in a graph."""
2: (0)              from collections import Counter
3: (0)              from itertools import combinations, repeat
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = [
7: (4)                  "is_matching",
8: (4)                  "is_maximal_matching",
9: (4)                  "is_perfect_matching",
10: (4)                 "max_weight_matching",
11: (4)                 "min_weight_matching",
12: (4)                 "maximal_matching",
13: (0)             ]
14: (0)             @not_implemented_for("multigraph")
15: (0)             @not_implemented_for("directed")
16: (0)             @nx._dispatchable
17: (0)             def maximal_matching(G):
18: (4)                 r"""Find a maximal matching in the graph.
19: (4)                 A matching is a subset of edges in which no node occurs more than once.
20: (4)                 A maximal matching cannot add more edges and still be a matching.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : NetworkX graph
24: (8)                     Undirected graph
25: (4)                 Returns
26: (4)                 -------
27: (4)                 matching : set
28: (8)                     A maximal matching of the graph.
29: (4)                 Examples
30: (4)                 --------
31: (4)                 >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)])
32: (4)                 >>> sorted(nx.maximal_matching(G))
33: (4)                 [(1, 2), (3, 5)]
34: (4)                 Notes
35: (4)                 -----
36: (4)                 The algorithm greedily selects a maximal matching M of the graph G
37: (4)                 (i.e. no superset of M exists). It runs in $O(|E|)$ time.
38: (4)                 """
39: (4)                 matching = set()
40: (4)                 nodes = set()
41: (4)                 for edge in G.edges():
42: (8)                     u, v = edge
43: (8)                     if u not in nodes and v not in nodes and u != v:
44: (12)                        matching.add(edge)
45: (12)                        nodes.update(edge)
46: (4)                 return matching
47: (0)             def matching_dict_to_set(matching):
48: (4)                 """Converts matching dict format to matching set format
49: (4)                 Converts a dictionary representing a matching (as returned by
50: (4)                 :func:`max_weight_matching`) to a set representing a matching (as
51: (4)                 returned by :func:`maximal_matching`).
52: (4)                 In the definition of maximal matching adopted by NetworkX,
53: (4)                 self-loops are not allowed, so the provided dictionary is expected
54: (4)                 to never have any mapping from a key to itself. However, the
55: (4)                 dictionary is expected to have mirrored key/value pairs, for
56: (4)                 example, key ``u`` with value ``v`` and key ``v`` with value ``u``.
```

```
 57: (4)                    """
 58: (4)                    edges = set()
 59: (4)                    for edge in matching.items():
 60: (8)                        u, v = edge
 61: (8)                        if (v, u) in edges or edge in edges:
 62: (12)                           continue
 63: (8)                        if u == v:
 64: (12)                           raise nx.NetworkXError(f"Selfloops cannot appear in matchings
{edge}")
 65: (8)                        edges.add(edge)
 66: (4)                    return edges
 67: (0)                @nx._dispatchable
 68: (0)                def is_matching(G, matching):
 69: (4)                    """Return True if ``matching`` is a valid matching of ``G``
 70: (4)                    A *matching* in a graph is a set of edges in which no two distinct
 71: (4)                    edges share a common endpoint. Each node is incident to at most one
 72: (4)                    edge in the matching. The edges are said to be independent.
 73: (4)                    Parameters
 74: (4)                    ----------
 75: (4)                    G : NetworkX graph
 76: (4)                    matching : dict or set
 77: (8)                        A dictionary or set representing a matching. If a dictionary, it
 78: (8)                        must have ``matching[u] == v`` and ``matching[v] == u`` for each
 79: (8)                        edge ``(u, v)`` in the matching. If a set, it must have elements
 80: (8)                        of the form ``(u, v)``, where ``(u, v)`` is an edge in the
 81: (8)                        matching.
 82: (4)                    Returns
 83: (4)                    -------
 84: (4)                    bool
 85: (8)                        Whether the given set or dictionary represents a valid matching
 86: (8)                        in the graph.
 87: (4)                    Raises
 88: (4)                    ------
 89: (4)                    NetworkXError
 90: (8)                        If the proposed matching has an edge to a node not in G.
 91: (8)                        Or if the matching is not a collection of 2-tuple edges.
 92: (4)                    Examples
 93: (4)                    --------
 94: (4)                    >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)])
 95: (4)                    >>> nx.is_maximal_matching(G, {1: 3, 2: 4})  # using dict to represent
matching
 96: (4)                    True
 97: (4)                    >>> nx.is_matching(G, {(1, 3), (2, 4)})  # using set to represent matching
 98: (4)                    True
 99: (4)                    """
 100: (4)                   if isinstance(matching, dict):
 101: (8)                       matching = matching_dict_to_set(matching)
 102: (4)                   nodes = set()
 103: (4)                   for edge in matching:
 104: (8)                       if len(edge) != 2:
 105: (12)                          raise nx.NetworkXError(f"matching has non-2-tuple edge {edge}")
 106: (8)                       u, v = edge
 107: (8)                       if u not in G or v not in G:
 108: (12)                          raise nx.NetworkXError(f"matching contains edge {edge} with node
not in G")
 109: (8)                       if u == v:
 110: (12)                          return False
 111: (8)                       if not G.has_edge(u, v):
 112: (12)                          return False
 113: (8)                       if u in nodes or v in nodes:
 114: (12)                          return False
 115: (8)                       nodes.update(edge)
 116: (4)                   return True
 117: (0)               @nx._dispatchable
 118: (0)               def is_maximal_matching(G, matching):
 119: (4)                   """Return True if ``matching`` is a maximal matching of ``G``
 120: (4)                   A *maximal matching* in a graph is a matching in which adding any
 121: (4)                   edge would cause the set to no longer be a valid matching.
 122: (4)                   Parameters
```

```
    ----------
    G : NetworkX graph
    matching : dict or set
        A dictionary or set representing a matching. If a dictionary, it
        must have ``matching[u] == v`` and ``matching[v] == u`` for each
        edge ``(u, v)`` in the matching. If a set, it must have elements
        of the form ``(u, v)``, where ``(u, v)`` is an edge in the
        matching.
    Returns
    -------
    bool
        Whether the given set or dictionary represents a valid maximal
        matching in the graph.
    Examples
    --------
    >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (3, 4), (3, 5)])
    >>> nx.is_maximal_matching(G, {(1, 2), (3, 4)})
    True
    """
    if isinstance(matching, dict):
        matching = matching_dict_to_set(matching)
    edges = set()
    nodes = set()
    for edge in matching:
        if len(edge) != 2:
            raise nx.NetworkXError(f"matching has non-2-tuple edge {edge}")
        u, v = edge
        if u not in G or v not in G:
            raise nx.NetworkXError(f"matching contains edge {edge} with node not in G")
        if u == v:
            return False
        if not G.has_edge(u, v):
            return False
        if u in nodes or v in nodes:
            return False
        nodes.update(edge)
        edges.add(edge)
        edges.add((v, u))
    for u, v in G.edges:
        if (u, v) not in edges:
            if u not in nodes and v not in nodes and u != v:
                return False
    return True
@nx._dispatchable
def is_perfect_matching(G, matching):
    """Return True if ``matching`` is a perfect matching for ``G``
    A *perfect matching* in a graph is a matching in which exactly one edge
    is incident upon each vertex.
    Parameters
    ----------
    G : NetworkX graph
    matching : dict or set
        A dictionary or set representing a matching. If a dictionary, it
        must have ``matching[u] == v`` and ``matching[v] == u`` for each
        edge ``(u, v)`` in the matching. If a set, it must have elements
        of the form ``(u, v)``, where ``(u, v)`` is an edge in the
        matching.
    Returns
    -------
    bool
        Whether the given set or dictionary represents a valid perfect
        matching in the graph.
    Examples
    --------
    >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5), (4, 6)])
    >>> my_match = {1: 2, 3: 5, 4: 6}
    >>> nx.is_perfect_matching(G, my_match)
    True
```

```
191: (4)                """
192: (4)                if isinstance(matching, dict):
193: (8)                    matching = matching_dict_to_set(matching)
194: (4)                nodes = set()
195: (4)                for edge in matching:
196: (8)                    if len(edge) != 2:
197: (12)                       raise nx.NetworkXError(f"matching has non-2-tuple edge {edge}")
198: (8)                    u, v = edge
199: (8)                    if u not in G or v not in G:
200: (12)                       raise nx.NetworkXError(f"matching contains edge {edge} with node
not in G")
201: (8)                    if u == v:
202: (12)                       return False
203: (8)                    if not G.has_edge(u, v):
204: (12)                       return False
205: (8)                    if u in nodes or v in nodes:
206: (12)                       return False
207: (8)                    nodes.update(edge)
208: (4)                return len(nodes) == len(G)
209: (0)            @not_implemented_for("multigraph")
210: (0)            @not_implemented_for("directed")
211: (0)            @nx._dispatchable(edge_attrs="weight")
212: (0)            def min_weight_matching(G, weight="weight"):
213: (4)                """Computing a minimum-weight maximal matching of G.
214: (4)                Use the maximum-weight algorithm with edge weights subtracted
215: (4)                from the maximum weight of all edges.
216: (4)                A matching is a subset of edges in which no node occurs more than once.
217: (4)                The weight of a matching is the sum of the weights of its edges.
218: (4)                A maximal matching cannot add more edges and still be a matching.
219: (4)                The cardinality of a matching is the number of matched edges.
220: (4)                This method replaces the edge weights with 1 plus the maximum edge weight
221: (4)                minus the original edge weight.
222: (4)                new_weight = (max_weight + 1) - edge_weight
223: (4)                then runs :func:`max_weight_matching` with the new weights.
224: (4)                The max weight matching with these new weights corresponds
225: (4)                to the min weight matching using the original weights.
226: (4)                Adding 1 to the max edge weight keeps all edge weights positive
227: (4)                and as integers if they started as integers.
228: (4)                You might worry that adding 1 to each weight would make the algorithm
229: (4)                favor matchings with more edges. But we use the parameter
230: (4)                `maxcardinality=True` in `max_weight_matching` to ensure that the
231: (4)                number of edges in the competing matchings are the same and thus
232: (4)                the optimum does not change due to changes in the number of edges.
233: (4)                Read the documentation of `max_weight_matching` for more information.
234: (4)                Parameters
235: (4)                ----------
236: (4)                G : NetworkX graph
237: (6)                  Undirected graph
238: (4)                weight: string, optional (default='weight')
239: (7)                   Edge data key corresponding to the edge weight.
240: (7)                   If key not found, uses 1 as weight.
241: (4)                Returns
242: (4)                -------
243: (4)                matching : set
244: (8)                    A minimal weight matching of the graph.
245: (4)                See Also
246: (4)                --------
247: (4)                max_weight_matching
248: (4)                """
249: (4)                if len(G.edges) == 0:
250: (8)                    return max_weight_matching(G, maxcardinality=True, weight=weight)
251: (4)                G_edges = G.edges(data=weight, default=1)
252: (4)                max_weight = 1 + max(w for _, _, w in G_edges)
253: (4)                InvG = nx.Graph()
254: (4)                edges = ((u, v, max_weight - w) for u, v, w in G_edges)
255: (4)                InvG.add_weighted_edges_from(edges, weight=weight)
256: (4)                return max_weight_matching(InvG, maxcardinality=True, weight=weight)
257: (0)            @not_implemented_for("multigraph")
258: (0)            @not_implemented_for("directed")
```

```
(0)            @nx._dispatchable(edge_attrs="weight")
(0)            def max_weight_matching(G, maxcardinality=False, weight="weight"):
(4)                """Compute a maximum-weighted matching of G.
(4)                A matching is a subset of edges in which no node occurs more than once.
(4)                The weight of a matching is the sum of the weights of its edges.
(4)                A maximal matching cannot add more edges and still be a matching.
(4)                The cardinality of a matching is the number of matched edges.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX graph
(6)                  Undirected graph
(4)                maxcardinality: bool, optional (default=False)
(7)                   If maxcardinality is True, compute the maximum-cardinality matching
(7)                   with maximum weight among all maximum-cardinality matchings.
(4)                weight: string, optional (default='weight')
(7)                   Edge data key corresponding to the edge weight.
(7)                   If key not found, uses 1 as weight.
(4)                Returns
(4)                -------
(4)                matching : set
(8)                    A maximal matching of the graph.
(5)                 Examples
(4)                --------
(4)                >>> G = nx.Graph()
(4)                >>> edges = [(1, 2, 6), (1, 3, 2), (2, 3, 1), (2, 4, 7), (3, 5, 9), (4, 5,
3)]
(4)                >>> G.add_weighted_edges_from(edges)
(4)                >>> sorted(nx.max_weight_matching(G))
(4)                [(2, 4), (5, 3)]
(4)                Notes
(4)                -----
(4)                If G has edges with weight attributes the edge data are used as
(4)                weight values else the weights are assumed to be 1.
(4)                This function takes time O(number_of_nodes ** 3).
(4)                If all edge weights are integers, the algorithm uses only integer
(4)                computations.  If floating point weights are used, the algorithm
(4)                could return a slightly suboptimal matching due to numeric
(4)                precision errors.
(4)                This method is based on the "blossom" method for finding augmenting
(4)                paths and the "primal-dual" method for finding a matching of maximum
(4)                weight, both methods invented by Jack Edmonds [1]_.
(4)                Bipartite graphs can also be matched using the functions present in
(4)                :mod:`networkx.algorithms.bipartite.matching`.
(4)                References
(4)                ----------
(4)                .. [1] "Efficient Algorithms for Finding Maximum Matching in Graphs",
(7)                   Zvi Galil, ACM Computing Surveys, 1986.
(4)                """
(4)                class NoNode:
(8)                    """Dummy value which is different from any node."""
(4)                class Blossom:
(8)                    """Representation of a non-trivial blossom or sub-blossom."""
(8)                    __slots__ = ["childs", "edges", "mybestedges"]
(8)                    def leaves(self):
(12)                       stack = [*self.childs]
(12)                       while stack:
(16)                           t = stack.pop()
(16)                           if isinstance(t, Blossom):
(20)                               stack.extend(t.childs)
(16)                           else:
(20)                               yield t
(4)                gnodes = list(G)
(4)                if not gnodes:
(8)                    return set()  # don't bother with empty graphs
(4)                maxweight = 0
(4)                allinteger = True
(4)                for i, j, d in G.edges(data=True):
(8)                    wt = d.get(weight, 1)
(8)                    if i != j and wt > maxweight:
```

```
(12)                        maxweight = wt
(8)                     allinteger = allinteger and (str(type(wt)).split("'")[1] in ("int", "long"))
(4)                 mate = {}
(4)                 label = {}
(4)                 labeledge = {}
(4)                 inblossom = dict(zip(gnodes, gnodes))
(4)                 blossomparent = dict(zip(gnodes, repeat(None)))
(4)                 blossombase = dict(zip(gnodes, gnodes))
(4)                 bestedge = {}
(4)                 dualvar = dict(zip(gnodes, repeat(maxweight)))
(4)                 blossomdual = {}
(4)                 allowedge = {}
(4)                 queue = []
(4)                 def slack(v, w):
(8)                     return dualvar[v] + dualvar[w] - 2 * G[v][w].get(weight, 1)
(4)                 def assignLabel(w, t, v):
(8)                     b = inblossom[w]
(8)                     assert label.get(w) is None and label.get(b) is None
(8)                     label[w] = label[b] = t
(8)                     if v is not None:
(12)                        labeledge[w] = labeledge[b] = (v, w)
(8)                     else:
(12)                        labeledge[w] = labeledge[b] = None
(8)                     bestedge[w] = bestedge[b] = None
(8)                     if t == 1:
(12)                        if isinstance(b, Blossom):
(16)                            queue.extend(b.leaves())
(12)                        else:
(16)                            queue.append(b)
(8)                     elif t == 2:
(12)                        base = blossombase[b]
(12)                        assignLabel(mate[base], 1, base)
(4)                 def scanBlossom(v, w):
(8)                     path = []
(8)                     base = NoNode
(8)                     while v is not NoNode:
(12)                        b = inblossom[v]
(12)                        if label[b] & 4:
(16)                            base = blossombase[b]
(16)                            break
(12)                        assert label[b] == 1
(12)                        path.append(b)
(12)                        label[b] = 5
(12)                        if labeledge[b] is None:
(16)                            assert blossombase[b] not in mate
(16)                            v = NoNode
(12)                        else:
(16)                            assert labeledge[b][0] == mate[blossombase[b]]
(16)                            v = labeledge[b][0]
(16)                            b = inblossom[v]
(16)                            assert label[b] == 2
(16)                            v = labeledge[b][0]
(12)                        if w is not NoNode:
(16)                            v, w = w, v
(8)                     for b in path:
(12)                        label[b] = 1
(8)                     return base
(4)                 def addBlossom(base, v, w):
(8)                     bb = inblossom[base]
(8)                     bv = inblossom[v]
(8)                     bw = inblossom[w]
(8)                     b = Blossom()
(8)                     blossombase[b] = base
(8)                     blossomparent[b] = None
(8)                     blossomparent[bb] = b
(8)                     b.childs = path = []
(8)                     b.edges = edgs = [(v, w)]
(8)                     while bv != bb:
```

```
395: (12)                        blossomparent[bv] = b
396: (12)                        path.append(bv)
397: (12)                        edgs.append(labeledge[bv])
398: (12)                        assert label[bv] == 2 or (
399: (16)                            label[bv] == 1 and labeledge[bv][0] == mate[blossombase[bv]]
400: (12)                        )
401: (12)                        v = labeledge[bv][0]
402: (12)                        bv = inblossom[v]
403: (8)                    path.append(bb)
404: (8)                    path.reverse()
405: (8)                    edgs.reverse()
406: (8)                    while bw != bb:
407: (12)                        blossomparent[bw] = b
408: (12)                        path.append(bw)
409: (12)                        edgs.append((labeledge[bw][1], labeledge[bw][0]))
410: (12)                        assert label[bw] == 2 or (
411: (16)                            label[bw] == 1 and labeledge[bw][0] == mate[blossombase[bw]]
412: (12)                        )
413: (12)                        w = labeledge[bw][0]
414: (12)                        bw = inblossom[w]
415: (8)                    assert label[bb] == 1
416: (8)                    label[b] = 1
417: (8)                    labeledge[b] = labeledge[bb]
418: (8)                    blossomdual[b] = 0
419: (8)                    for v in b.leaves():
420: (12)                        if label[inblossom[v]] == 2:
421: (16)                            queue.append(v)
422: (12)                        inblossom[v] = b
423: (8)                    bestedgeto = {}
424: (8)                    for bv in path:
425: (12)                        if isinstance(bv, Blossom):
426: (16)                            if bv.mybestedges is not None:
427: (20)                                nblist = bv.mybestedges
428: (20)                                bv.mybestedges = None
429: (16)                            else:
430: (20)                                nblist = [
431: (24)                                    (v, w) for v in bv.leaves() for w in G.neighbors(v) if
v != w
432: (20)                                ]
433: (12)                        else:
434: (16)                            nblist = [(bv, w) for w in G.neighbors(bv) if bv != w]
435: (12)                        for k in nblist:
436: (16)                            (i, j) = k
437: (16)                            if inblossom[j] == b:
438: (20)                                i, j = j, i
439: (16)                            bj = inblossom[j]
440: (16)                            if (
441: (20)                                bj != b
442: (20)                                and label.get(bj) == 1
443: (20)                                and ((bj not in bestedgeto) or slack(i, j) <
slack(*bestedgeto[bj]))
444: (16)                            ):
445: (20)                                bestedgeto[bj] = k
446: (12)                        bestedge[bv] = None
447: (8)                    b.mybestedges = list(bestedgeto.values())
448: (8)                    mybestedge = None
449: (8)                    bestedge[b] = None
450: (8)                    for k in b.mybestedges:
451: (12)                        kslack = slack(*k)
452: (12)                        if mybestedge is None or kslack < mybestslack:
453: (16)                            mybestedge = k
454: (16)                            mybestslack = kslack
455: (8)                    bestedge[b] = mybestedge
456: (4)                def expandBlossom(b, endstage):
457: (8)                    def _recurse(b, endstage):
458: (12)                        for s in b.childs:
459: (16)                            blossomparent[s] = None
460: (16)                            if isinstance(s, Blossom):
461: (20)                                if endstage and blossomdual[s] == 0:
```

```
462: (24)                                   yield s
463: (20)                               else:
464: (24)                                   for v in s.leaves():
465: (28)                                       inblossom[v] = s
466: (16)                           else:
467: (20)                               inblossom[s] = s
468: (12)                       if (not endstage) and label.get(b) == 2:
469: (16)                           entrychild = inblossom[labeledge[b][1]]
470: (16)                           j = b.childs.index(entrychild)
471: (16)                           if j & 1:
472: (20)                               j -= len(b.childs)
473: (20)                               jstep = 1
474: (16)                           else:
475: (20)                               jstep = -1
476: (16)                           v, w = labeledge[b]
477: (16)                           while j != 0:
478: (20)                               if jstep == 1:
479: (24)                                   p, q = b.edges[j]
480: (20)                               else:
481: (24)                                   q, p = b.edges[j - 1]
482: (20)                               label[w] = None
483: (20)                               label[q] = None
484: (20)                               assignLabel(w, 2, v)
485: (20)                               allowedge[(p, q)] = allowedge[(q, p)] = True
486: (20)                               j += jstep
487: (20)                               if jstep == 1:
488: (24)                                   v, w = b.edges[j]
489: (20)                               else:
490: (24)                                   w, v = b.edges[j - 1]
491: (20)                               allowedge[(v, w)] = allowedge[(w, v)] = True
492: (20)                               j += jstep
493: (16)                           bw = b.childs[j]
494: (16)                           label[w] = label[bw] = 2
495: (16)                           labeledge[w] = labeledge[bw] = (v, w)
496: (16)                           bestedge[bw] = None
497: (16)                           j += jstep
498: (16)                           while b.childs[j] != entrychild:
499: (20)                               bv = b.childs[j]
500: (20)                               if label.get(bv) == 1:
501: (24)                                   j += jstep
502: (24)                                   continue
503: (20)                               if isinstance(bv, Blossom):
504: (24)                                   for v in bv.leaves():
505: (28)                                       if label.get(v):
506: (32)                                           break
507: (20)                               else:
508: (24)                                   v = bv
509: (20)                               if label.get(v):
510: (24)                                   assert label[v] == 2
511: (24)                                   assert inblossom[v] == bv
512: (24)                                   label[v] = None
513: (24)                                   label[mate[blossombase[bv]]] = None
514: (24)                                   assignLabel(v, 2, labeledge[v][0])
515: (20)                               j += jstep
516: (12)                       label.pop(b, None)
517: (12)                       labeledge.pop(b, None)
518: (12)                       bestedge.pop(b, None)
519: (12)                       del blossomparent[b]
520: (12)                       del blossombase[b]
521: (12)                       del blossomdual[b]
522: (8)                    stack = [_recurse(b, endstage)]
523: (8)                    while stack:
524: (12)                       top = stack[-1]
525: (12)                       for s in top:
526: (16)                           stack.append(_recurse(s, endstage))
527: (16)                           break
528: (12)                       else:
529: (16)                           stack.pop()
530: (4)                def augmentBlossom(b, v):
```

```
(8)                 def _recurse(b, v):
(12)                    t = v
(12)                    while blossomparent[t] != b:
(16)                        t = blossomparent[t]
(12)                    if isinstance(t, Blossom):
(16)                        yield (t, v)
(12)                    i = j = b.childs.index(t)
(12)                    if i & 1:
(16)                        j -= len(b.childs)
(16)                        jstep = 1
(12)                    else:
(16)                        jstep = -1
(12)                    while j != 0:
(16)                        j += jstep
(16)                        t = b.childs[j]
(16)                        if jstep == 1:
(20)                            w, x = b.edges[j]
(16)                        else:
(20)                            x, w = b.edges[j - 1]
(16)                        if isinstance(t, Blossom):
(20)                            yield (t, w)
(16)                        j += jstep
(16)                        t = b.childs[j]
(16)                        if isinstance(t, Blossom):
(20)                            yield (t, x)
(16)                        mate[w] = x
(16)                        mate[x] = w
(12)                    b.childs = b.childs[i:] + b.childs[:i]
(12)                    b.edges = b.edges[i:] + b.edges[:i]
(12)                    blossombase[b] = blossombase[b.childs[0]]
(12)                    assert blossombase[b] == v
(8)                 stack = [_recurse(b, v)]
(8)                 while stack:
(12)                    top = stack[-1]
(12)                    for args in top:
(16)                        stack.append(_recurse(*args))
(16)                        break
(12)                    else:
(16)                        stack.pop()
(4)             def augmentMatching(v, w):
(8)                 for s, j in ((v, w), (w, v)):
(12)                    while 1:
(16)                        bs = inblossom[s]
(16)                        assert label[bs] == 1
(16)                        assert (labeledge[bs] is None and blossombase[bs] not in mate)
or (
(20)                            labeledge[bs][0] == mate[blossombase[bs]]
(16)                        )
(16)                        if isinstance(bs, Blossom):
(20)                            augmentBlossom(bs, s)
(16)                        mate[s] = j
(16)                        if labeledge[bs] is None:
(20)                            break
(16)                        t = labeledge[bs][0]
(16)                        bt = inblossom[t]
(16)                        assert label[bt] == 2
(16)                        s, j = labeledge[bt]
(16)                        assert blossombase[bt] == t
(16)                        if isinstance(bt, Blossom):
(20)                            augmentBlossom(bt, j)
(16)                        mate[j] = s
(4)             def verifyOptimum():
(8)                 if maxcardinality:
(12)                    vdualoffset = max(0, -min(dualvar.values()))
(8)                 else:
(12)                    vdualoffset = 0
(8)                 assert min(dualvar.values()) + vdualoffset >= 0
(8)                 assert len(blossomdual) == 0 or min(blossomdual.values()) >= 0
(8)                 for i, j, d in G.edges(data=True):
```

```
599: (12)                        wt = d.get(weight, 1)
600: (12)                        if i == j:
601: (16)                            continue  # ignore self-loops
602: (12)                        s = dualvar[i] + dualvar[j] - 2 * wt
603: (12)                        iblossoms = [i]
604: (12)                        jblossoms = [j]
605: (12)                        while blossomparent[iblossoms[-1]] is not None:
606: (16)                            iblossoms.append(blossomparent[iblossoms[-1]])
607: (12)                        while blossomparent[jblossoms[-1]] is not None:
608: (16)                            jblossoms.append(blossomparent[jblossoms[-1]])
609: (12)                        iblossoms.reverse()
610: (12)                        jblossoms.reverse()
611: (12)                        for bi, bj in zip(iblossoms, jblossoms):
612: (16)                            if bi != bj:
613: (20)                                break
614: (16)                            s += 2 * blossomdual[bi]
615: (12)                        assert s >= 0
616: (12)                        if mate.get(i) == j or mate.get(j) == i:
617: (16)                            assert mate[i] == j and mate[j] == i
618: (16)                            assert s == 0
619: (8)                    for v in gnodes:
620: (12)                        assert (v in mate) or dualvar[v] + vdualoffset == 0
621: (8)                    for b in blossomdual:
622: (12)                        if blossomdual[b] > 0:
623: (16)                            assert len(b.edges) % 2 == 1
624: (16)                            for i, j in b.edges[1::2]:
625: (20)                                assert mate[i] == j and mate[j] == i
626: (4)                while 1:
627: (8)                    label.clear()
628: (8)                    labeledge.clear()
629: (8)                    bestedge.clear()
630: (8)                    for b in blossomdual:
631: (12)                        b.mybestedges = None
632: (8)                    allowedge.clear()
633: (8)                    queue[:] = []
634: (8)                    for v in gnodes:
635: (12)                        if (v not in mate) and label.get(inblossom[v]) is None:
636: (16)                            assignLabel(v, 1, None)
637: (8)                    augmented = 0
638: (8)                    while 1:
639: (12)                        while queue and not augmented:
640: (16)                            v = queue.pop()
641: (16)                            assert label[inblossom[v]] == 1
642: (16)                            for w in G.neighbors(v):
643: (20)                                if w == v:
644: (24)                                    continue  # ignore self-loops
645: (20)                                bv = inblossom[v]
646: (20)                                bw = inblossom[w]
647: (20)                                if bv == bw:
648: (24)                                    continue
649: (20)                                if (v, w) not in allowedge:
650: (24)                                    kslack = slack(v, w)
651: (24)                                    if kslack <= 0:
652: (28)                                        allowedge[(v, w)] = allowedge[(w, v)] = True
653: (20)                                if (v, w) in allowedge:
654: (24)                                    if label.get(bw) is None:
655: (28)                                        assignLabel(w, 2, v)
656: (24)                                    elif label.get(bw) == 1:
657: (28)                                        base = scanBlossom(v, w)
658: (28)                                        if base is not NoNode:
659: (32)                                            addBlossom(base, v, w)
660: (28)                                        else:
661: (32)                                            augmentMatching(v, w)
662: (32)                                            augmented = 1
663: (32)                                            break
664: (24)                                    elif label.get(w) is None:
665: (28)                                        assert label[bw] == 2
666: (28)                                        label[w] = 2
667: (28)                                        labeledge[w] = (v, w)
```

```
668: (20)                                elif label.get(bw) == 1:
669: (24)                                    if bestedge.get(bv) is None or kslack <
slack(*bestedge[bv]):
670: (28)                                        bestedge[bv] = (v, w)
671: (20)                                elif label.get(w) is None:
672: (24)                                    if bestedge.get(w) is None or kslack <
slack(*bestedge[w]):
673: (28)                                        bestedge[w] = (v, w)
674: (12)                        if augmented:
675: (16)                            break
676: (12)                        deltatype = -1
677: (12)                        delta = deltaedge = deltablossom = None
678: (12)                        if not maxcardinality:
679: (16)                            deltatype = 1
680: (16)                            delta = min(dualvar.values())
681: (12)                        for v in G.nodes():
682: (16)                            if label.get(inblossom[v]) is None and bestedge.get(v) is not
None:
683: (20)                                d = slack(*bestedge[v])
684: (20)                                if deltatype == -1 or d < delta:
685: (24)                                    delta = d
686: (24)                                    deltatype = 2
687: (24)                                    deltaedge = bestedge[v]
688: (12)                        for b in blossomparent:
689: (16)                            if (
690: (20)                                blossomparent[b] is None
691: (20)                                and label.get(b) == 1
692: (20)                                and bestedge.get(b) is not None
693: (16)                            ):
694: (20)                                kslack = slack(*bestedge[b])
695: (20)                                if allinteger:
696: (24)                                    assert (kslack % 2) == 0
697: (24)                                    d = kslack // 2
698: (20)                                else:
699: (24)                                    d = kslack / 2.0
700: (20)                                if deltatype == -1 or d < delta:
701: (24)                                    delta = d
702: (24)                                    deltatype = 3
703: (24)                                    deltaedge = bestedge[b]
704: (12)                        for b in blossomdual:
705: (16)                            if (
706: (20)                                blossomparent[b] is None
707: (20)                                and label.get(b) == 2
708: (20)                                and (deltatype == -1 or blossomdual[b] < delta)
709: (16)                            ):
710: (20)                                delta = blossomdual[b]
711: (20)                                deltatype = 4
712: (20)                                deltablossom = b
713: (12)                        if deltatype == -1:
714: (16)                            assert maxcardinality
715: (16)                            deltatype = 1
716: (16)                            delta = max(0, min(dualvar.values()))
717: (12)                        for v in gnodes:
718: (16)                            if label.get(inblossom[v]) == 1:
719: (20)                                dualvar[v] -= delta
720: (16)                            elif label.get(inblossom[v]) == 2:
721: (20)                                dualvar[v] += delta
722: (12)                        for b in blossomdual:
723: (16)                            if blossomparent[b] is None:
724: (20)                                if label.get(b) == 1:
725: (24)                                    blossomdual[b] += delta
726: (20)                                elif label.get(b) == 2:
727: (24)                                    blossomdual[b] -= delta
728: (12)                        if deltatype == 1:
729: (16)                            break
730: (12)                        elif deltatype == 2:
731: (16)                            (v, w) = deltaedge
732: (16)                            assert label[inblossom[v]] == 1
733: (16)                            allowedge[(v, w)] = allowedge[(w, v)] = True
```

```
734: (16)                           queue.append(v)
735: (12)                       elif deltatype == 3:
736: (16)                           (v, w) = deltaedge
737: (16)                           allowedge[(v, w)] = allowedge[(w, v)] = True
738: (16)                           assert label[inblossom[v]] == 1
739: (16)                           queue.append(v)
740: (12)                       elif deltatype == 4:
741: (16)                           expandBlossom(deltablossom, False)
742: (8)                    for v in mate:
743: (12)                       assert mate[mate[v]] == v
744: (8)                    if not augmented:
745: (12)                       break
746: (8)                    for b in list(blossomdual.keys()):
747: (12)                       if b not in blossomdual:
748: (16)                           continue  # already expanded
749: (12)                       if blossomparent[b] is None and label.get(b) == 1 and
blossomdual[b] == 0:
750: (16)                           expandBlossom(b, True)
751: (4)                if allinteger:
752: (8)                    verifyOptimum()
753: (4)                return matching_dict_to_set(mate)
```

File 38 - mis.py:

```
1: (0)              """
2: (0)              Algorithm to find a maximal (not maximum) independent set.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for, py_random_state
6: (0)              __all__ = ["maximal_independent_set"]
7: (0)              @not_implemented_for("directed")
8: (0)              @py_random_state(2)
9: (0)              @nx._dispatchable
10: (0)             def maximal_independent_set(G, nodes=None, seed=None):
11: (4)                 """Returns a random maximal independent set guaranteed to contain
12: (4)                 a given set of nodes.
13: (4)                 An independent set is a set of nodes such that the subgraph
14: (4)                 of G induced by these nodes contains no edges. A maximal
15: (4)                 independent set is an independent set such that it is not possible
16: (4)                 to add a new node and still get an independent set.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : NetworkX graph
20: (4)                 nodes : list or iterable
21: (7)                    Nodes that must be part of the independent set. This set of nodes
22: (7)                    must be independent.
23: (4)                 seed : integer, random_state, or None (default)
24: (8)                     Indicator of random number generation state.
25: (8)                     See :ref:`Randomness<randomness>`.
26: (4)                 Returns
27: (4)                 -------
28: (4)                 indep_nodes : list
29: (7)                    List of nodes that are part of a maximal independent set.
30: (4)                 Raises
31: (4)                 ------
32: (4)                 NetworkXUnfeasible
33: (7)                    If the nodes in the provided list are not part of the graph or
34: (7)                    do not form an independent set, an exception is raised.
35: (4)                 NetworkXNotImplemented
36: (8)                     If `G` is directed.
37: (4)                 Examples
38: (4)                 --------
39: (4)                 >>> G = nx.path_graph(5)
40: (4)                 >>> nx.maximal_independent_set(G)  # doctest: +SKIP
41: (4)                 [4, 0, 2]
42: (4)                 >>> nx.maximal_independent_set(G, [1])  # doctest: +SKIP
43: (4)                 [1, 3]
```

```
44: (4)                 Notes
45: (4)                 -----
46: (4)                 This algorithm does not solve the maximum independent set problem.
47: (4)                 """
48: (4)                 if not nodes:
49: (8)                     nodes = {seed.choice(list(G))}
50: (4)                 else:
51: (8)                     nodes = set(nodes)
52: (4)                 if not nodes.issubset(G):
53: (8)                     raise nx.NetworkXUnfeasible(f"{nodes} is not a subset of the nodes of
G")
54: (4)                 neighbors = set.union(*[set(G.adj[v]) for v in nodes])
55: (4)                 if set.intersection(neighbors, nodes):
56: (8)                     raise nx.NetworkXUnfeasible(f"{nodes} is not an independent set of G")
57: (4)                 indep_nodes = list(nodes)
58: (4)                 available_nodes = set(G.nodes()).difference(neighbors.union(nodes))
59: (4)                 while available_nodes:
60: (8)                     node = seed.choice(list(available_nodes))
61: (8)                     indep_nodes.append(node)
62: (8)                     available_nodes.difference_update(list(G.adj[node]) + [node])
63: (4)                 return indep_nodes
```

File 39 - moral.py:

```
1: (0)              r"""Function for computing the moral graph of a directed graph."""
2: (0)              import itertools
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["moral_graph"]
6: (0)              @not_implemented_for("undirected")
7: (0)              @nx._dispatchable(returns_graph=True)
8: (0)              def moral_graph(G):
9: (4)                  r"""Return the Moral Graph
10: (4)                 Returns the moralized graph of a given directed graph.
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 G : NetworkX graph
14: (8)                     Directed graph
15: (4)                 Returns
16: (4)                 -------
17: (4)                 H : NetworkX graph
18: (8)                     The undirected moralized graph of G
19: (4)                 Raises
20: (4)                 ------
21: (4)                 NetworkXNotImplemented
22: (8)                     If `G` is undirected.
23: (4)                 Examples
24: (4)                 --------
25: (4)                 >>> G = nx.DiGraph([(1, 2), (2, 3), (2, 5), (3, 4), (4, 3)])
26: (4)                 >>> G_moral = nx.moral_graph(G)
27: (4)                 >>> G_moral.edges()
28: (4)                 EdgeView([(1, 2), (2, 3), (2, 5), (2, 4), (3, 4)])
29: (4)                 Notes
30: (4)                 -----
31: (4)                 A moral graph is an undirected graph H = (V, E) generated from a
32: (4)                 directed Graph, where if a node has more than one parent node, edges
33: (4)                 between these parent nodes are inserted and all directed edges become
34: (4)                 undirected.
35: (4)                 https://en.wikipedia.org/wiki/Moral_graph
36: (4)                 References
37: (4)                 ----------
38: (4)                 .. [1] Wray L. Buntine. 1995. Chain graphs for learning.
39: (11)                       In Proceedings of the Eleventh conference on Uncertainty
40: (11)                       in artificial intelligence (UAI'95)
41: (4)                 """
42: (4)                 H = G.to_undirected()
43: (4)                 for preds in G.pred.values():
```

```
44: (8)                     predecessors_combinations = itertools.combinations(preds, r=2)
45: (8)                     H.add_edges_from(predecessors_combinations)
46: (4)                 return H

----------------------------------------

File 40 - node_classification.py:

1: (0)              """ This module provides the functions for node classification problem.
2: (0)              The functions in this module are not imported
3: (0)              into the top level `networkx` namespace.
4: (0)              You can access these functions by importing
5: (0)              the `networkx.algorithms.node_classification` modules,
6: (0)              then accessing the functions as attributes of `node_classification`.
7: (0)              For example:
8: (2)                >>> from networkx.algorithms import node_classification
9: (2)                >>> G = nx.path_graph(4)
10: (2)               >>> G.edges()
11: (2)               EdgeView([(0, 1), (1, 2), (2, 3)])
12: (2)               >>> G.nodes[0]["label"] = "A"
13: (2)               >>> G.nodes[3]["label"] = "B"
14: (2)               >>> node_classification.harmonic_function(G)
15: (2)               ['A', 'A', 'B', 'B']
16: (0)             References
17: (0)             ----------
18: (0)             Zhu, X., Ghahramani, Z., & Lafferty, J. (2003, August).
19: (0)             Semi-supervised learning using gaussian fields and harmonic functions.
20: (0)             In ICML (Vol. 3, pp. 912-919).
21: (0)             """
22: (0)             import networkx as nx
23: (0)             __all__ = ["harmonic_function", "local_and_global_consistency"]
24: (0)             @nx.utils.not_implemented_for("directed")
25: (0)             @nx._dispatchable(node_attrs="label_name")
26: (0)             def harmonic_function(G, max_iter=30, label_name="label"):
27: (4)                 """Node classification by Harmonic function
28: (4)                 Function for computing Harmonic function algorithm by Zhu et al.
29: (4)                 Parameters
30: (4)                 ----------
31: (4)                 G : NetworkX Graph
32: (4)                 max_iter : int
33: (8)                     maximum number of iterations allowed
34: (4)                 label_name : string
35: (8)                     name of target labels to predict
36: (4)                 Returns
37: (4)                 -------
38: (4)                 predicted : list
39: (8)                     List of length ``len(G)`` with the predicted labels for each node.
40: (4)                 Raises
41: (4)                 ------
42: (4)                 NetworkXError
43: (8)                     If no nodes in `G` have attribute `label_name`.
44: (4)                 Examples
45: (4)                 --------
46: (4)                 >>> from networkx.algorithms import node_classification
47: (4)                 >>> G = nx.path_graph(4)
48: (4)                 >>> G.nodes[0]["label"] = "A"
49: (4)                 >>> G.nodes[3]["label"] = "B"
50: (4)                 >>> G.nodes(data=True)
51: (4)                 NodeDataView({0: {'label': 'A'}, 1: {}, 2: {}, 3: {'label': 'B'}})
52: (4)                 >>> G.edges()
53: (4)                 EdgeView([(0, 1), (1, 2), (2, 3)])
54: (4)                 >>> predicted = node_classification.harmonic_function(G)
55: (4)                 >>> predicted
56: (4)                 ['A', 'A', 'B', 'B']
57: (4)                 References
58: (4)                 ----------
59: (4)                 Zhu, X., Ghahramani, Z., & Lafferty, J. (2003, August).
60: (4)                 Semi-supervised learning using gaussian fields and harmonic functions.
61: (4)                 In ICML (Vol. 3, pp. 912-919).
```

```
62: (4)                 """
63: (4)                 import numpy as np
64: (4)                 import scipy as sp
65: (4)                 X = nx.to_scipy_sparse_array(G)  # adjacency matrix
66: (4)                 labels, label_dict = _get_label_info(G, label_name)
67: (4)                 if labels.shape[0] == 0:
68: (8)                     raise nx.NetworkXError(
69: (12)                        f"No node on the input graph is labeled by '{label_name}'."
70: (8)                     )
71: (4)                 n_samples = X.shape[0]
72: (4)                 n_classes = label_dict.shape[0]
73: (4)                 F = np.zeros((n_samples, n_classes))
74: (4)                 degrees = X.sum(axis=0)
75: (4)                 degrees[degrees == 0] = 1  # Avoid division by 0
76: (4)                 D = sp.sparse.csr_array(sp.sparse.diags((1.0 / degrees), offsets=0))
77: (4)                 P = (D @ X).tolil()
78: (4)                 P[labels[:, 0]] = 0  # labels[:, 0] indicates IDs of labeled nodes
79: (4)                 B = np.zeros((n_samples, n_classes))
80: (4)                 B[labels[:, 0], labels[:, 1]] = 1
81: (4)                 for _ in range(max_iter):
82: (8)                     F = (P @ F) + B
83: (4)                 return label_dict[np.argmax(F, axis=1)].tolist()
84: (0)             @nx.utils.not_implemented_for("directed")
85: (0)             @nx._dispatchable(node_attrs="label_name")
86: (0)             def local_and_global_consistency(G, alpha=0.99, max_iter=30,
label_name="label"):
87: (4)                 """Node classification by Local and Global Consistency
88: (4)                 Function for computing Local and global consistency algorithm by Zhou et
al.
89: (4)                 Parameters
90: (4)                 ----------
91: (4)                 G : NetworkX Graph
92: (4)                 alpha : float
93: (8)                     Clamping factor
94: (4)                 max_iter : int
95: (8)                     Maximum number of iterations allowed
96: (4)                 label_name : string
97: (8)                     Name of target labels to predict
98: (4)                 Returns
99: (4)                 -------
100: (4)                predicted : list
101: (8)                    List of length ``len(G)`` with the predicted labels for each node.
102: (4)                Raises
103: (4)                ------
104: (4)                NetworkXError
105: (8)                    If no nodes in `G` have attribute `label_name`.
106: (4)                Examples
107: (4)                --------
108: (4)                >>> from networkx.algorithms import node_classification
109: (4)                >>> G = nx.path_graph(4)
110: (4)                >>> G.nodes[0]["label"] = "A"
111: (4)                >>> G.nodes[3]["label"] = "B"
112: (4)                >>> G.nodes(data=True)
113: (4)                NodeDataView({0: {'label': 'A'}, 1: {}, 2: {}, 3: {'label': 'B'}})
114: (4)                >>> G.edges()
115: (4)                EdgeView([(0, 1), (1, 2), (2, 3)])
116: (4)                >>> predicted = node_classification.local_and_global_consistency(G)
117: (4)                >>> predicted
118: (4)                ['A', 'A', 'B', 'B']
119: (4)                References
120: (4)                ----------
121: (4)                Zhou, D., Bousquet, O., Lal, T. N., Weston, J., & Schölkopf, B. (2004).
122: (4)                Learning with local and global consistency.
123: (4)                Advances in neural information processing systems, 16(16), 321-328.
124: (4)                """
125: (4)                import numpy as np
126: (4)                import scipy as sp
127: (4)                X = nx.to_scipy_sparse_array(G)  # adjacency matrix
128: (4)                labels, label_dict = _get_label_info(G, label_name)
```

```
129: (4)                if labels.shape[0] == 0:
130: (8)                    raise nx.NetworkXError(
131: (12)                       f"No node on the input graph is labeled by '{label_name}'."
132: (8)                    )
133: (4)                n_samples = X.shape[0]
134: (4)                n_classes = label_dict.shape[0]
135: (4)                F = np.zeros((n_samples, n_classes))
136: (4)                degrees = X.sum(axis=0)
137: (4)                degrees[degrees == 0] = 1  # Avoid division by 0
138: (4)                D2 = np.sqrt(sp.sparse.csr_array(sp.sparse.diags((1.0 / degrees),
offsets=0)))
139: (4)                P = alpha * ((D2 @ X) @ D2)
140: (4)                B = np.zeros((n_samples, n_classes))
141: (4)                B[labels[:, 0], labels[:, 1]] = 1 - alpha
142: (4)                for _ in range(max_iter):
143: (8)                    F = (P @ F) + B
144: (4)                return label_dict[np.argmax(F, axis=1)].tolist()
145: (0)            def _get_label_info(G, label_name):
146: (4)                """Get and return information of labels from the input graph
147: (4)                Parameters
148: (4)                ----------
149: (4)                G : Network X graph
150: (4)                label_name : string
151: (8)                    Name of the target label
152: (4)                Returns
153: (4)                -------
154: (4)                labels : numpy array, shape = [n_labeled_samples, 2]
155: (8)                    Array of pairs of labeled node ID and label ID
156: (4)                label_dict : numpy array, shape = [n_classes]
157: (8)                    Array of labels
158: (8)                    i-th element contains the label corresponding label ID `i`
159: (4)                """
160: (4)                import numpy as np
161: (4)                labels = []
162: (4)                label_to_id = {}
163: (4)                lid = 0
164: (4)                for i, n in enumerate(G.nodes(data=True)):
165: (8)                    if label_name in n[1]:
166: (12)                       label = n[1][label_name]
167: (12)                       if label not in label_to_id:
168: (16)                           label_to_id[label] = lid
169: (16)                           lid += 1
170: (12)                       labels.append([i, label_to_id[label]])
171: (4)                labels = np.array(labels)
172: (4)                label_dict = np.array(
173: (8)                    [label for label, _ in sorted(label_to_id.items(), key=lambda x:
x[1])]
174: (4)                )
175: (4)                return (labels, label_dict)
```

File 41 - non_randomness.py:

```
1: (0)              r""" Computation of graph non-randomness
2: (0)              """
3: (0)              import math
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = ["non_randomness"]
7: (0)              @not_implemented_for("directed")
8: (0)              @not_implemented_for("multigraph")
9: (0)              @nx._dispatchable(edge_attrs="weight")
10: (0)             def non_randomness(G, k=None, weight="weight"):
11: (4)                 """Compute the non-randomness of graph G.
12: (4)                 The first returned value nr is the sum of non-randomness values of all
13: (4)                 edges within the graph (where the non-randomness of an edge tends to be
14: (4)                 small when the two nodes linked by that edge are from two different
15: (4)                 communities).
```

```
16: (4)                 The second computed value nr_rd is a relative measure that indicates
17: (4)                 to what extent graph G is different from random graphs in terms
18: (4)                 of probability. When it is close to 0, the graph tends to be more
19: (4)                 likely generated by an Erdos Renyi model.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : NetworkX graph
23: (8)                     Graph must be symmetric, connected, and without self-loops.
24: (4)                 k : int
25: (8)                     The number of communities in G.
26: (8)                     If k is not set, the function will use a default community
27: (8)                     detection algorithm to set it.
28: (4)                 weight : string or None, optional (default=None)
29: (8)                     The name of an edge attribute that holds the numerical value used
30: (8)                     as a weight. If None, then each edge has weight 1, i.e., the graph is
31: (8)                     binary.
32: (4)                 Returns
33: (4)                 -------
34: (4)                 non-randomness : (float, float) tuple
35: (8)                     Non-randomness, Relative non-randomness w.r.t.
36: (8)                     Erdos Renyi random graphs.
37: (4)                 Raises
38: (4)                 ------
39: (4)                 NetworkXException
40: (8)                     if the input graph is not connected.
41: (4)                 NetworkXError
42: (8)                     if the input graph contains self-loops.
43: (4)                 Examples
44: (4)                 --------
45: (4)                 >>> G = nx.karate_club_graph()
46: (4)                 >>> nr, nr_rd = nx.non_randomness(G, 2)
47: (4)                 >>> nr, nr_rd = nx.non_randomness(G, 2, "weight")
48: (4)                 Notes
49: (4)                 -----
50: (4)                 This computes Eq. (4.4) and (4.5) in Ref. [1]_.
51: (4)                 If a weight field is passed, this algorithm will use the eigenvalues
52: (4)                 of the weighted adjacency matrix to compute Eq. (4.4) and (4.5).
53: (4)                 References
54: (4)                 ----------
55: (4)                 .. [1] Xiaowei Ying and Xintao Wu,
56: (11)                       On Randomness Measures for Social Networks,
57: (11)                       SIAM International Conference on Data Mining. 2009
58: (4)                 """
59: (4)                 import numpy as np
60: (4)                 if not nx.is_connected(G):
61: (8)                     raise nx.NetworkXException("Non connected graph.")
62: (4)                 if len(list(nx.selfloop_edges(G))) > 0:
63: (8)                     raise nx.NetworkXError("Graph must not contain self-loops")
64: (4)                 if k is None:
65: (8)                     k = len(tuple(nx.community.label_propagation_communities(G)))
66: (4)                 eigenvalues = np.linalg.eigvals(nx.to_numpy_array(G, weight=weight))
67: (4)                 nr = float(np.real(np.sum(eigenvalues[:k])))
68: (4)                 n = G.number_of_nodes()
69: (4)                 m = G.number_of_edges()
70: (4)                 p = (2 * k * m) / (n * (n - k))
71: (4)                 nr_rd = (nr - ((n - 2 * k) * p + k)) / math.sqrt(2 * k * p * (1 - p))
72: (4)                 return nr, nr_rd

----------------------------------------

File 42 - planar_drawing.py:

1: (0)              from collections import defaultdict
2: (0)              import networkx as nx
3: (0)              __all__ = ["combinatorial_embedding_to_pos"]
4: (0)              def combinatorial_embedding_to_pos(embedding, fully_triangulate=False):
5: (4)                  """Assigns every node a (x, y) position based on the given embedding
6: (4)                  The algorithm iteratively inserts nodes of the input graph in a certain
7: (4)                  order and rearranges previously inserted nodes so that the planar drawing
```

```
 8: (4)                    stays valid. This is done efficiently by only maintaining relative
 9: (4)                    positions during the node placements and calculating the absolute
positions
10: (4)                    at the end. For more information see [1]_.
11: (4)                    Parameters
12: (4)                    ----------
13: (4)                    embedding : nx.PlanarEmbedding
14: (8)                        This defines the order of the edges
15: (4)                    fully_triangulate : bool
16: (8)                        If set to True the algorithm adds edges to a copy of the input
17: (8)                        embedding and makes it chordal.
18: (4)                    Returns
19: (4)                    -------
20: (4)                    pos : dict
21: (8)                        Maps each node to a tuple that defines the (x, y) position
22: (4)                    References
23: (4)                    ----------
24: (4)                    .. [1] M. Chrobak and T.H. Payne:
25: (8)                        A Linear-time Algorithm for Drawing a Planar Graph on a Grid 1989
26: (8)                        http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.6677
27: (4)                    """
28: (4)                    if len(embedding.nodes()) < 4:
29: (8)                        default_positions = [(0, 0), (2, 0), (1, 1)]
30: (8)                        pos = {}
31: (8)                        for i, v in enumerate(embedding.nodes()):
32: (12)                           pos[v] = default_positions[i]
33: (8)                        return pos
34: (4)                    embedding, outer_face = triangulate_embedding(embedding,
fully_triangulate)
35: (4)                    left_t_child = {}
36: (4)                    right_t_child = {}
37: (4)                    delta_x = {}
38: (4)                    y_coordinate = {}
39: (4)                    node_list = get_canonical_ordering(embedding, outer_face)
40: (4)                    v1, v2, v3 = node_list[0][0], node_list[1][0], node_list[2][0]
41: (4)                    delta_x[v1] = 0
42: (4)                    y_coordinate[v1] = 0
43: (4)                    right_t_child[v1] = v3
44: (4)                    left_t_child[v1] = None
45: (4)                    delta_x[v2] = 1
46: (4)                    y_coordinate[v2] = 0
47: (4)                    right_t_child[v2] = None
48: (4)                    left_t_child[v2] = None
49: (4)                    delta_x[v3] = 1
50: (4)                    y_coordinate[v3] = 1
51: (4)                    right_t_child[v3] = v2
52: (4)                    left_t_child[v3] = None
53: (4)                    for k in range(3, len(node_list)):
54: (8)                        vk, contour_nbrs = node_list[k]
55: (8)                        wp = contour_nbrs[0]
56: (8)                        wp1 = contour_nbrs[1]
57: (8)                        wq = contour_nbrs[-1]
58: (8)                        wq1 = contour_nbrs[-2]
59: (8)                        adds_mult_tri = len(contour_nbrs) > 2
60: (8)                        delta_x[wp1] += 1
61: (8)                        delta_x[wq] += 1
62: (8)                        delta_x_wp_wq = sum(delta_x[x] for x in contour_nbrs[1:])
63: (8)                        delta_x[vk] = (-y_coordinate[wp] + delta_x_wp_wq + y_coordinate[wq])
// 2
64: (8)                        y_coordinate[vk] = (y_coordinate[wp] + delta_x_wp_wq +
y_coordinate[wq]) // 2
65: (8)                        delta_x[wq] = delta_x_wp_wq - delta_x[vk]
66: (8)                        if adds_mult_tri:
67: (12)                           delta_x[wp1] -= delta_x[vk]
68: (8)                        right_t_child[wp] = vk
69: (8)                        right_t_child[vk] = wq
70: (8)                        if adds_mult_tri:
71: (12)                           left_t_child[vk] = wp1
72: (12)                           right_t_child[wq1] = None
```

```
73: (8)                     else:
74: (12)                        left_t_child[vk] = None
75: (4)                 pos = {}
76: (4)                 pos[v1] = (0, y_coordinate[v1])
77: (4)                 remaining_nodes = [v1]
78: (4)                 while remaining_nodes:
79: (8)                     parent_node = remaining_nodes.pop()
80: (8)                     set_position(
81: (12)                        parent_node, left_t_child, remaining_nodes, delta_x, y_coordinate,
pos
82: (8)                     )
83: (8)                     set_position(
84: (12)                        parent_node, right_t_child, remaining_nodes, delta_x,
y_coordinate, pos
85: (8)                     )
86: (4)                 return pos
87: (0)             def set_position(parent, tree, remaining_nodes, delta_x, y_coordinate, pos):
88: (4)                 """Helper method to calculate the absolute position of nodes."""
89: (4)                 child = tree[parent]
90: (4)                 parent_node_x = pos[parent][0]
91: (4)                 if child is not None:
92: (8)                     child_x = parent_node_x + delta_x[child]
93: (8)                     pos[child] = (child_x, y_coordinate[child])
94: (8)                     remaining_nodes.append(child)
95: (0)             def get_canonical_ordering(embedding, outer_face):
96: (4)                 """Returns a canonical ordering of the nodes
97: (4)                 The canonical ordering of nodes (v1, ..., vn) must fulfill the following
98: (4)                 conditions:
99: (4)                 (See Lemma 1 in [2]_)
100: (4)                - For the subgraph G_k of the input graph induced by v1, ..., vk it holds:
101: (8)                    - 2-connected
102: (8)                    - internally triangulated
103: (8)                    - the edge (v1, v2) is part of the outer face
104: (4)                - For a node v(k+1) the following holds:
105: (8)                    - The node v(k+1) is part of the outer face of G_k
106: (8)                    - It has at least two neighbors in G_k
107: (8)                    - All neighbors of v(k+1) in G_k lie consecutively on the outer face
of
108: (10)                     G_k (excluding the edge (v1, v2)).
109: (4)                The algorithm used here starts with G_n (containing all nodes). It first
110: (4)                selects the nodes v1 and v2. And then tries to find the order of the other
111: (4)                nodes by checking which node can be removed in order to fulfill the
112: (4)                conditions mentioned above. This is done by calculating the number of
113: (4)                chords of nodes on the outer face. For more information see [1]_.
114: (4)                Parameters
115: (4)                ----------
116: (4)                embedding : nx.PlanarEmbedding
117: (8)                    The embedding must be triangulated
118: (4)                outer_face : list
119: (8)                    The nodes on the outer face of the graph
120: (4)                Returns
121: (4)                -------
122: (4)                ordering : list
123: (8)                    A list of tuples `(vk, wp_wq)`. Here `vk` is the node at this position
124: (8)                    in the canonical ordering. The element `wp_wq` is a list of nodes that
125: (8)                    make up the outer face of G_k.
126: (4)                References
127: (4)                ----------
128: (4)                .. [1] Steven Chaplick.
129: (8)                    Canonical Orders of Planar Graphs and (some of) Their Applications
2015
130: (8)                    https://wuecampus2.uni-
wuerzburg.de/moodle/pluginfile.php/545727/mod_resource/content/0/vg-ss15-vl03-canonical-orders-
druckversion.pdf
131: (4)                .. [2] M. Chrobak and T.H. Payne:
132: (8)                    A Linear-time Algorithm for Drawing a Planar Graph on a Grid 1989
133: (8)                    http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.6677
134: (4)                """
135: (4)                v1 = outer_face[0]
```

```
(4)     v2 = outer_face[1]
(4)     chords = defaultdict(int)  # Maps nodes to the number of their chords
(4)     marked_nodes = set()
(4)     ready_to_pick = set(outer_face)
(4)     outer_face_ccw_nbr = {}
(4)     prev_nbr = v2
(4)     for idx in range(2, len(outer_face)):
(8)         outer_face_ccw_nbr[prev_nbr] = outer_face[idx]
(8)         prev_nbr = outer_face[idx]
(4)     outer_face_ccw_nbr[prev_nbr] = v1
(4)     outer_face_cw_nbr = {}
(4)     prev_nbr = v1
(4)     for idx in range(len(outer_face) - 1, 0, -1):
(8)         outer_face_cw_nbr[prev_nbr] = outer_face[idx]
(8)         prev_nbr = outer_face[idx]
(4)     def is_outer_face_nbr(x, y):
(8)         if x not in outer_face_ccw_nbr:
(12)            return outer_face_cw_nbr[x] == y
(8)         if x not in outer_face_cw_nbr:
(12)            return outer_face_ccw_nbr[x] == y
(8)         return outer_face_ccw_nbr[x] == y or outer_face_cw_nbr[x] == y
(4)     def is_on_outer_face(x):
(8)         return x not in marked_nodes and (x in outer_face_ccw_nbr or x == v1)
(4)     for v in outer_face:
(8)         for nbr in embedding.neighbors_cw_order(v):
(12)            if is_on_outer_face(nbr) and not is_outer_face_nbr(v, nbr):
(16)                chords[v] += 1
(16)                ready_to_pick.discard(v)
(4)     canonical_ordering = [None] * len(embedding.nodes())
(4)     canonical_ordering[0] = (v1, [])
(4)     canonical_ordering[1] = (v2, [])
(4)     ready_to_pick.discard(v1)
(4)     ready_to_pick.discard(v2)
(4)     for k in range(len(embedding.nodes()) - 1, 1, -1):
(8)         v = ready_to_pick.pop()
(8)         marked_nodes.add(v)
(8)         wp = None
(8)         wq = None
(8)         nbr_iterator = iter(embedding.neighbors_cw_order(v))
(8)         while True:
(12)            nbr = next(nbr_iterator)
(12)            if nbr in marked_nodes:
(16)                continue
(12)            if is_on_outer_face(nbr):
(16)                if nbr == v1:
(20)                    wp = v1
(16)                elif nbr == v2:
(20)                    wq = v2
(16)                else:
(20)                    if outer_face_cw_nbr[nbr] == v:
(24)                        wp = nbr
(20)                    else:
(24)                        wq = nbr
(12)            if wp is not None and wq is not None:
(16)                break
(8)         wp_wq = [wp]
(8)         nbr = wp
(8)         while nbr != wq:
(12)            next_nbr = embedding[v][nbr]["ccw"]
(12)            wp_wq.append(next_nbr)
(12)            outer_face_cw_nbr[nbr] = next_nbr
(12)            outer_face_ccw_nbr[next_nbr] = nbr
(12)            nbr = next_nbr
(8)         if len(wp_wq) == 2:
(12)            chords[wp] -= 1
(12)            if chords[wp] == 0:
(16)                ready_to_pick.add(wp)
(12)            chords[wq] -= 1
(12)            if chords[wq] == 0:
```

```
(16)                            ready_to_pick.add(wq)
(8)                     else:
(12)                        new_face_nodes = set(wp_wq[1:-1])
(12)                        for w in new_face_nodes:
(16)                            ready_to_pick.add(w)
(16)                            for nbr in embedding.neighbors_cw_order(w):
(20)                                if is_on_outer_face(nbr) and not is_outer_face_nbr(w, nbr):
(24)                                    chords[w] += 1
(24)                                    ready_to_pick.discard(w)
(24)                                    if nbr not in new_face_nodes:
(28)                                        chords[nbr] += 1
(28)                                        ready_to_pick.discard(nbr)
(8)                     canonical_ordering[k] = (v, wp_wq)
(4)                 return canonical_ordering
(0)             def triangulate_face(embedding, v1, v2):
(4)                 """Triangulates the face given by half edge (v, w)
(4)                 Parameters
(4)                 ----------
(4)                 embedding : nx.PlanarEmbedding
(4)                 v1 : node
(8)                     The half-edge (v1, v2) belongs to the face that gets triangulated
(4)                 v2 : node
(4)                 """
(4)                 _, v3 = embedding.next_face_half_edge(v1, v2)
(4)                 _, v4 = embedding.next_face_half_edge(v2, v3)
(4)                 if v1 in (v2, v3):
(8)                     return
(4)                 while v1 != v4:
(8)                     if embedding.has_edge(v1, v3):
(12)                        v1, v2, v3 = v2, v3, v4
(8)                     else:
(12)                        embedding.add_half_edge(v1, v3, ccw=v2)
(12)                        embedding.add_half_edge(v3, v1, cw=v2)
(12)                        v1, v2, v3 = v1, v3, v4
(8)                     _, v4 = embedding.next_face_half_edge(v2, v3)
(0)             def triangulate_embedding(embedding, fully_triangulate=True):
(4)                 """Triangulates the embedding.
(4)                 Traverses faces of the embedding and adds edges to a copy of the
(4)                 embedding to triangulate it.
(4)                 The method also ensures that the resulting graph is 2-connected by adding
(4)                 edges if the same vertex is contained twice on a path around a face.
(4)                 Parameters
(4)                 ----------
(4)                 embedding : nx.PlanarEmbedding
(8)                     The input graph must contain at least 3 nodes.
(4)                 fully_triangulate : bool
(8)                     If set to False the face with the most nodes is chooses as outer face.
(8)                     This outer face does not get triangulated.
(4)                 Returns
(4)                 -------
(4)                 (embedding, outer_face) : (nx.PlanarEmbedding, list) tuple
(8)                     The element `embedding` is a new embedding containing all edges from
(8)                     the input embedding and the additional edges to triangulate the graph.
(8)                     The element `outer_face` is a list of nodes that lie on the outer face.
(8)                     If the graph is fully triangulated these are three arbitrary connected
(8)                     nodes.
(4)                 """
(4)                 if len(embedding.nodes) <= 1:
(8)                     return embedding, list(embedding.nodes)
(4)                 embedding = nx.PlanarEmbedding(embedding)
(4)                 component_nodes = [next(iter(x)) for x in nx.connected_components(embedding)]
(4)                 for i in range(len(component_nodes) - 1):
(8)                     v1 = component_nodes[i]
(8)                     v2 = component_nodes[i + 1]
(8)                     embedding.connect_components(v1, v2)
(4)                 outer_face = []  # A face with the most number of nodes
```

```
271: (4)                face_list = []
272: (4)                edges_visited = set()  # Used to keep track of already visited faces
273: (4)                for v in embedding.nodes():
274: (8)                    for w in embedding.neighbors_cw_order(v):
275: (12)                       new_face = make_bi_connected(embedding, v, w, edges_visited)
276: (12)                       if new_face:
277: (16)                           face_list.append(new_face)
278: (16)                           if len(new_face) > len(outer_face):
279: (20)                               outer_face = new_face
280: (4)                for face in face_list:
281: (8)                    if face is not outer_face or fully_triangulate:
282: (12)                       triangulate_face(embedding, face[0], face[1])
283: (4)                if fully_triangulate:
284: (8)                    v1 = outer_face[0]
285: (8)                    v2 = outer_face[1]
286: (8)                    v3 = embedding[v2][v1]["ccw"]
287: (8)                    outer_face = [v1, v2, v3]
288: (4)                return embedding, outer_face
289: (0)            def make_bi_connected(embedding, starting_node, outgoing_node, edges_counted):
290: (4)                """Triangulate a face and make it 2-connected
291: (4)                This method also adds all edges on the face to `edges_counted`.
292: (4)                Parameters
293: (4)                ----------
294: (4)                embedding: nx.PlanarEmbedding
295: (8)                    The embedding that defines the faces
296: (4)                starting_node : node
297: (8)                    A node on the face
298: (4)                outgoing_node : node
299: (8)                    A node such that the half edge (starting_node, outgoing_node) belongs
300: (8)                    to the face
301: (4)                edges_counted: set
302: (8)                    Set of all half-edges that belong to a face that have been visited
303: (4)                Returns
304: (4)                -------
305: (4)                face_nodes: list
306: (8)                    A list of all nodes at the border of this face
307: (4)                """
308: (4)                if (starting_node, outgoing_node) in edges_counted:
309: (8)                    return []
310: (4)                edges_counted.add((starting_node, outgoing_node))
311: (4)                v1 = starting_node
312: (4)                v2 = outgoing_node
313: (4)                face_list = [starting_node]  # List of nodes around the face
314: (4)                face_set = set(face_list)  # Set for faster queries
315: (4)                _, v3 = embedding.next_face_half_edge(v1, v2)
316: (4)                while v2 != starting_node or v3 != outgoing_node:
317: (8)                    if v1 == v2:
318: (12)                       raise nx.NetworkXException("Invalid half-edge")
319: (8)                    if v2 in face_set:
320: (12)                       embedding.add_half_edge(v1, v3, ccw=v2)
321: (12)                       embedding.add_half_edge(v3, v1, cw=v2)
322: (12)                       edges_counted.add((v2, v3))
323: (12)                       edges_counted.add((v3, v1))
324: (12)                       v2 = v1
325: (8)                    else:
326: (12)                       face_set.add(v2)
327: (12)                       face_list.append(v2)
328: (8)                    v1 = v2
329: (8)                    v2, v3 = embedding.next_face_half_edge(v2, v3)
330: (8)                    edges_counted.add((v1, v2))
331: (4)                return face_list
```

File 43 - planarity.py:

```
1: (0)              from collections import defaultdict
2: (0)              import networkx as nx
3: (0)              __all__ = ["check_planarity", "is_planar", "PlanarEmbedding"]
```

```
4: (0)              @nx._dispatchable
5: (0)              def is_planar(G):
6: (4)                  """Returns True if and only if `G` is planar.
7: (4)                  A graph is *planar* iff it can be drawn in a plane without
8: (4)                  any edge intersections.
9: (4)                  Parameters
10: (4)                 ----------
11: (4)                 G : NetworkX graph
12: (4)                 Returns
13: (4)                 -------
14: (4)                 bool
15: (7)                    Whether the graph is planar.
16: (4)                 Examples
17: (4)                 --------
18: (4)                 >>> G = nx.Graph([(0, 1), (0, 2)])
19: (4)                 >>> nx.is_planar(G)
20: (4)                 True
21: (4)                 >>> nx.is_planar(nx.complete_graph(5))
22: (4)                 False
23: (4)                 See Also
24: (4)                 --------
25: (4)                 check_planarity :
26: (8)                     Check if graph is planar *and* return a `PlanarEmbedding` instance if
True.
27: (4)                 """
28: (4)                 return check_planarity(G, counterexample=False)[0]
29: (0)             @nx._dispatchable(returns_graph=True)
30: (0)             def check_planarity(G, counterexample=False):
31: (4)                 """Check if a graph is planar and return a counterexample or an embedding.
32: (4)                 A graph is planar iff it can be drawn in a plane without
33: (4)                 any edge intersections.
34: (4)                 Parameters
35: (4)                 ----------
36: (4)                 G : NetworkX graph
37: (4)                 counterexample : bool
38: (8)                     A Kuratowski subgraph (to proof non planarity) is only returned if set
39: (8)                     to true.
40: (4)                 Returns
41: (4)                 -------
42: (4)                 (is_planar, certificate) : (bool, NetworkX graph) tuple
43: (8)                     is_planar is true if the graph is planar.
44: (8)                     If the graph is planar `certificate` is a PlanarEmbedding
45: (8)                     otherwise it is a Kuratowski subgraph.
46: (4)                 Examples
47: (4)                 --------
48: (4)                 >>> G = nx.Graph([(0, 1), (0, 2)])
49: (4)                 >>> is_planar, P = nx.check_planarity(G)
50: (4)                 >>> print(is_planar)
51: (4)                 True
52: (4)                 When `G` is planar, a `PlanarEmbedding` instance is returned:
53: (4)                 >>> P.get_data()
54: (4)                 {0: [1, 2], 1: [0], 2: [0]}
55: (4)                 Notes
56: (4)                 -----
57: (4)                 A (combinatorial) embedding consists of cyclic orderings of the incident
58: (4)                 edges at each vertex. Given such an embedding there are multiple
approaches
59: (4)                 discussed in literature to drawing the graph (subject to various
60: (4)                 constraints, e.g. integer coordinates), see e.g. [2].
61: (4)                 The planarity check algorithm and extraction of the combinatorial
embedding
62: (4)                 is based on the Left-Right Planarity Test [1].
63: (4)                 A counterexample is only generated if the corresponding parameter is set,
64: (4)                 because the complexity of the counterexample generation is higher.
65: (4)                 See also
66: (4)                 --------
67: (4)                 is_planar :
68: (8)                     Check for planarity without creating a `PlanarEmbedding` or
counterexample.
```

```
(4)     References
(4)     ----------
(4)     .. [1] Ulrik Brandes:
(8)        The Left-Right Planarity Test
(8)        2009
(8)        http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.217.9208
(4)     .. [2] Takao Nishizeki, Md Saidur Rahman:
(8)        Planar graph drawing
(8)        Lecture Notes Series on Computing: Volume 12
(8)        2004
(4)     """
(4)     planarity_state = LRPlanarity(G)
(4)     embedding = planarity_state.lr_planarity()
(4)     if embedding is None:
(8)         if counterexample:
(12)            return False, get_counterexample(G)
(8)         else:
(12)            return False, None
(4)     else:
(8)         return True, embedding
(0)  @nx._dispatchable(returns_graph=True)
(0)  def check_planarity_recursive(G, counterexample=False):
(4)     """Recursive version of :meth:`check_planarity`."""
(4)     planarity_state = LRPlanarity(G)
(4)     embedding = planarity_state.lr_planarity_recursive()
(4)     if embedding is None:
(8)         if counterexample:
(12)            return False, get_counterexample_recursive(G)
(8)         else:
(12)            return False, None
(4)     else:
(8)         return True, embedding
(0)  @nx._dispatchable(returns_graph=True)
(0)  def get_counterexample(G):
(4)     """Obtains a Kuratowski subgraph.
(4)     Raises nx.NetworkXException if G is planar.
(4)     The function removes edges such that the graph is still not planar.
(4)     At some point the removal of any edge would make the graph planar.
(4)     This subgraph must be a Kuratowski subgraph.
(4)     Parameters
(4)     ----------
(4)     G : NetworkX graph
(4)     Returns
(4)     -------
(4)     subgraph : NetworkX graph
(8)         A Kuratowski subgraph that proves that G is not planar.
(4)     """
(4)     G = nx.Graph(G)
(4)     if check_planarity(G)[0]:
(8)         raise nx.NetworkXException("G is planar - no counter example.")
(4)     subgraph = nx.Graph()
(4)     for u in G:
(8)         nbrs = list(G[u])
(8)         for v in nbrs:
(12)            G.remove_edge(u, v)
(12)            if check_planarity(G)[0]:
(16)                G.add_edge(u, v)
(16)                subgraph.add_edge(u, v)
(4)     return subgraph
(0)  @nx._dispatchable(returns_graph=True)
(0)  def get_counterexample_recursive(G):
(4)     """Recursive version of :meth:`get_counterexample`."""
(4)     G = nx.Graph(G)
(4)     if check_planarity_recursive(G)[0]:
(8)         raise nx.NetworkXException("G is planar - no counter example.")
(4)     subgraph = nx.Graph()
(4)     for u in G:
(8)         nbrs = list(G[u])
(8)         for v in nbrs:
```

```
138: (12)                        G.remove_edge(u, v)
139: (12)                        if check_planarity_recursive(G)[0]:
140: (16)                            G.add_edge(u, v)
141: (16)                            subgraph.add_edge(u, v)
142: (4)                return subgraph
143: (0)            class Interval:
144: (4)                """Represents a set of return edges.
145: (4)                All return edges in an interval induce a same constraint on the contained
146: (4)                edges, which means that all edges must either have a left orientation or
147: (4)                all edges must have a right orientation.
148: (4)                """
149: (4)                def __init__(self, low=None, high=None):
150: (8)                    self.low = low
151: (8)                    self.high = high
152: (4)                def empty(self):
153: (8)                    """Check if the interval is empty"""
154: (8)                    return self.low is None and self.high is None
155: (4)                def copy(self):
156: (8)                    """Returns a copy of this interval"""
157: (8)                    return Interval(self.low, self.high)
158: (4)                def conflicting(self, b, planarity_state):
159: (8)                    """Returns True if interval I conflicts with edge b"""
160: (8)                    return (
161: (12)                       not self.empty()
162: (12)                       and planarity_state.lowpt[self.high] > planarity_state.lowpt[b]
163: (8)                    )
164: (0)            class ConflictPair:
165: (4)                """Represents a different constraint between two intervals.
166: (4)                The edges in the left interval must have a different orientation than
167: (4)                the one in the right interval.
168: (4)                """
169: (4)                def __init__(self, left=Interval(), right=Interval()):
170: (8)                    self.left = left
171: (8)                    self.right = right
172: (4)                def swap(self):
173: (8)                    """Swap left and right intervals"""
174: (8)                    temp = self.left
175: (8)                    self.left = self.right
176: (8)                    self.right = temp
177: (4)                def lowest(self, planarity_state):
178: (8)                    """Returns the lowest lowpoint of a conflict pair"""
179: (8)                    if self.left.empty():
180: (12)                       return planarity_state.lowpt[self.right.low]
181: (8)                    if self.right.empty():
182: (12)                       return planarity_state.lowpt[self.left.low]
183: (8)                    return min(
184: (12)                       planarity_state.lowpt[self.left.low],
planarity_state.lowpt[self.right.low]
185: (8)                    )
186: (0)            def top_of_stack(l):
187: (4)                """Returns the element on top of the stack."""
188: (4)                if not l:
189: (8)                    return None
190: (4)                return l[-1]
191: (0)            class LRPlanarity:
192: (4)                """A class to maintain the state during planarity check."""
193: (4)                __slots__ = [
194: (8)                    "G",
195: (8)                    "roots",
196: (8)                    "height",
197: (8)                    "lowpt",
198: (8)                    "lowpt2",
199: (8)                    "nesting_depth",
200: (8)                    "parent_edge",
201: (8)                    "DG",
202: (8)                    "adjs",
203: (8)                    "ordered_adjs",
204: (8)                    "ref",
205: (8)                    "side",
```

```
(8)                 "S",
(8)                 "stack_bottom",
(8)                 "lowpt_edge",
(8)                 "left_ref",
(8)                 "right_ref",
(8)                 "embedding",
(4)             ]
(4)             def __init__(self, G):
(8)                 self.G = nx.Graph()
(8)                 self.G.add_nodes_from(G.nodes)
(8)                 for e in G.edges:
(12)                    if e[0] != e[1]:
(16)                        self.G.add_edge(e[0], e[1])
(8)                 self.roots = []
(8)                 self.height = defaultdict(lambda: None)
(8)                 self.lowpt = {}  # height of lowest return point of an edge
(8)                 self.lowpt2 = {}  # height of second lowest return point
(8)                 self.nesting_depth = {}  # for nesting order
(8)                 self.parent_edge = defaultdict(lambda: None)
(8)                 self.DG = nx.DiGraph()
(8)                 self.DG.add_nodes_from(G.nodes)
(8)                 self.adjs = {}
(8)                 self.ordered_adjs = {}
(8)                 self.ref = defaultdict(lambda: None)
(8)                 self.side = defaultdict(lambda: 1)
(8)                 self.S = []
(8)                 self.stack_bottom = {}
(8)                 self.lowpt_edge = {}
(8)                 self.left_ref = {}
(8)                 self.right_ref = {}
(8)                 self.embedding = PlanarEmbedding()
(4)             def lr_planarity(self):
(8)                 """Execute the LR planarity test.
(8)                 Returns
(8)                 -------
(8)                 embedding : dict
(12)                    If the graph is planar an embedding is returned. Otherwise None.
(8)                 """
(8)                 if self.G.order() > 2 and self.G.size() > 3 * self.G.order() - 6:
(12)                    return None
(8)                 for v in self.G:
(12)                    self.adjs[v] = list(self.G[v])
(8)                 for v in self.G:
(12)                    if self.height[v] is None:
(16)                        self.height[v] = 0
(16)                        self.roots.append(v)
(16)                        self.dfs_orientation(v)
(8)                 self.G = None
(8)                 self.lowpt2 = None
(8)                 self.adjs = None
(8)                 for v in self.DG:  # sort the adjacency lists by nesting depth
(12)                    self.ordered_adjs[v] = sorted(
(16)                        self.DG[v], key=lambda x: self.nesting_depth[(v, x)]
(12)                    )
(8)                 for v in self.roots:
(12)                    if not self.dfs_testing(v):
(16)                        return None
(8)                 self.height = None
(8)                 self.lowpt = None
(8)                 self.S = None
(8)                 self.stack_bottom = None
(8)                 self.lowpt_edge = None
(8)                 for e in self.DG.edges:
(12)                    self.nesting_depth[e] = self.sign(e) * self.nesting_depth[e]
(8)                 self.embedding.add_nodes_from(self.DG.nodes)
(8)                 for v in self.DG:
(12)                    self.ordered_adjs[v] = sorted(
(16)                        self.DG[v], key=lambda x: self.nesting_depth[(v, x)]
(12)                    )
```

```
275: (12)                        previous_node = None
276: (12)                        for w in self.ordered_adjs[v]:
277: (16)                            self.embedding.add_half_edge(v, w, ccw=previous_node)
278: (16)                            previous_node = w
279: (8)                    self.DG = None
280: (8)                    self.nesting_depth = None
281: (8)                    self.ref = None
282: (8)                    for v in self.roots:
283: (12)                        self.dfs_embedding(v)
284: (8)                    self.roots = None
285: (8)                    self.parent_edge = None
286: (8)                    self.ordered_adjs = None
287: (8)                    self.left_ref = None
288: (8)                    self.right_ref = None
289: (8)                    self.side = None
290: (8)                    return self.embedding
291: (4)                def lr_planarity_recursive(self):
292: (8)                    """Recursive version of :meth:`lr_planarity`."""
293: (8)                    if self.G.order() > 2 and self.G.size() > 3 * self.G.order() - 6:
294: (12)                        return None
295: (8)                    for v in self.G:
296: (12)                        if self.height[v] is None:
297: (16)                            self.height[v] = 0
298: (16)                            self.roots.append(v)
299: (16)                            self.dfs_orientation_recursive(v)
300: (8)                    self.G = None
301: (8)                    for v in self.DG:  # sort the adjacency lists by nesting depth
302: (12)                        self.ordered_adjs[v] = sorted(
303: (16)                            self.DG[v], key=lambda x: self.nesting_depth[(v, x)]
304: (12)                        )
305: (8)                    for v in self.roots:
306: (12)                        if not self.dfs_testing_recursive(v):
307: (16)                            return None
308: (8)                    for e in self.DG.edges:
309: (12)                        self.nesting_depth[e] = self.sign_recursive(e) *
self.nesting_depth[e]
310: (8)                    self.embedding.add_nodes_from(self.DG.nodes)
311: (8)                    for v in self.DG:
312: (12)                        self.ordered_adjs[v] = sorted(
313: (16)                            self.DG[v], key=lambda x: self.nesting_depth[(v, x)]
314: (12)                        )
315: (12)                        previous_node = None
316: (12)                        for w in self.ordered_adjs[v]:
317: (16)                            self.embedding.add_half_edge(v, w, ccw=previous_node)
318: (16)                            previous_node = w
319: (8)                    for v in self.roots:
320: (12)                        self.dfs_embedding_recursive(v)
321: (8)                    return self.embedding
322: (4)                def dfs_orientation(self, v):
323: (8)                    """Orient the graph by DFS, compute lowpoints and nesting order."""
324: (8)                    dfs_stack = [v]
325: (8)                    ind = defaultdict(lambda: 0)
326: (8)                    skip_init = defaultdict(lambda: False)
327: (8)                    while dfs_stack:
328: (12)                        v = dfs_stack.pop()
329: (12)                        e = self.parent_edge[v]
330: (12)                        for w in self.adjs[v][ind[v] :]:
331: (16)                            vw = (v, w)
332: (16)                            if not skip_init[vw]:
333: (20)                                if (v, w) in self.DG.edges or (w, v) in self.DG.edges:
334: (24)                                    ind[v] += 1
335: (24)                                    continue  # the edge was already oriented
336: (20)                                self.DG.add_edge(v, w)  # orient the edge
337: (20)                                self.lowpt[vw] = self.height[v]
338: (20)                                self.lowpt2[vw] = self.height[v]
339: (20)                                if self.height[w] is None:  # (v, w) is a tree edge
340: (24)                                    self.parent_edge[w] = vw
341: (24)                                    self.height[w] = self.height[v] + 1
342: (24)                                    dfs_stack.append(v)  # revisit v after finishing w
```

```
(24)                                dfs_stack.append(w)  # visit w next
(24)                                skip_init[vw] = True  # don't redo this block
(24)                                break  # handle next node in dfs_stack (i.e. w)
(20)                            else:  # (v, w) is a back edge
(24)                                self.lowpt[vw] = self.height[w]
(16)                        self.nesting_depth[vw] = 2 * self.lowpt[vw]
(16)                        if self.lowpt2[vw] < self.height[v]:  # chordal
(20)                            self.nesting_depth[vw] += 1
(16)                        if e is not None:
(20)                            if self.lowpt[vw] < self.lowpt[e]:
(24)                                self.lowpt2[e] = min(self.lowpt[e], self.lowpt2[vw])
(24)                                self.lowpt[e] = self.lowpt[vw]
(20)                            elif self.lowpt[vw] > self.lowpt[e]:
(24)                                self.lowpt2[e] = min(self.lowpt2[e], self.lowpt[vw])
(20)                            else:
(24)                                self.lowpt2[e] = min(self.lowpt2[e], self.lowpt2[vw])
(16)                        ind[v] += 1
(4)                 def dfs_orientation_recursive(self, v):
(8)                     """Recursive version of :meth:`dfs_orientation`."""
(8)                     e = self.parent_edge[v]
(8)                     for w in self.G[v]:
(12)                        if (v, w) in self.DG.edges or (w, v) in self.DG.edges:
(16)                            continue  # the edge was already oriented
(12)                        vw = (v, w)
(12)                        self.DG.add_edge(v, w)  # orient the edge
(12)                        self.lowpt[vw] = self.height[v]
(12)                        self.lowpt2[vw] = self.height[v]
(12)                        if self.height[w] is None:  # (v, w) is a tree edge
(16)                            self.parent_edge[w] = vw
(16)                            self.height[w] = self.height[v] + 1
(16)                            self.dfs_orientation_recursive(w)
(12)                        else:  # (v, w) is a back edge
(16)                            self.lowpt[vw] = self.height[w]
(12)                        self.nesting_depth[vw] = 2 * self.lowpt[vw]
(12)                        if self.lowpt2[vw] < self.height[v]:  # chordal
(16)                            self.nesting_depth[vw] += 1
(12)                        if e is not None:
(16)                            if self.lowpt[vw] < self.lowpt[e]:
(20)                                self.lowpt2[e] = min(self.lowpt[e], self.lowpt2[vw])
(20)                                self.lowpt[e] = self.lowpt[vw]
(16)                            elif self.lowpt[vw] > self.lowpt[e]:
(20)                                self.lowpt2[e] = min(self.lowpt2[e], self.lowpt[vw])
(16)                            else:
(20)                                self.lowpt2[e] = min(self.lowpt2[e], self.lowpt2[vw])
(4)                 def dfs_testing(self, v):
(8)                     """Test for LR partition."""
(8)                     dfs_stack = [v]
(8)                     ind = defaultdict(lambda: 0)
(8)                     skip_init = defaultdict(lambda: False)
(8)                     while dfs_stack:
(12)                        v = dfs_stack.pop()
(12)                        e = self.parent_edge[v]
(12)                        skip_final = False
(12)                        for w in self.ordered_adjs[v][ind[v] :]:
(16)                            ei = (v, w)
(16)                            if not skip_init[ei]:
(20)                                self.stack_bottom[ei] = top_of_stack(self.S)
(20)                                if ei == self.parent_edge[w]:  # tree edge
(24)                                    dfs_stack.append(v)  # revisit v after finishing w
(24)                                    dfs_stack.append(w)  # visit w next
(24)                                    skip_init[ei] = True  # don't redo this block
(24)                                    skip_final = True  # skip final work after breaking
(24)                                    break  # handle next node in dfs_stack (i.e. w)
(20)                                else:  # back edge
(24)                                    self.lowpt_edge[ei] = ei
(24)                                    self.S.append(ConflictPair(right=Interval(ei, ei)))
(16)                            if self.lowpt[ei] < self.height[v]:
(20)                                if w == self.ordered_adjs[v][0]:  # e_i has return edge
(24)                                    self.lowpt_edge[e] = self.lowpt_edge[ei]
```

```
(20)                                else:  # add constraints of e_i
(24)                                    if not self.add_constraints(ei, e):
(28)                                        return False
(16)                            ind[v] += 1
(12)                        if not skip_final:
(16)                            if e is not None:  # v isn't root
(20)                                self.remove_back_edges(e)
(8)                     return True
(4)                 def dfs_testing_recursive(self, v):
(8)                     """Recursive version of :meth:`dfs_testing`."""
(8)                     e = self.parent_edge[v]
(8)                     for w in self.ordered_adjs[v]:
(12)                        ei = (v, w)
(12)                        self.stack_bottom[ei] = top_of_stack(self.S)
(12)                        if ei == self.parent_edge[w]:  # tree edge
(16)                            if not self.dfs_testing_recursive(w):
(20)                                return False
(12)                        else:  # back edge
(16)                            self.lowpt_edge[ei] = ei
(16)                            self.S.append(ConflictPair(right=Interval(ei, ei)))
(12)                        if self.lowpt[ei] < self.height[v]:
(16)                            if w == self.ordered_adjs[v][0]:  # e_i has return edge
(20)                                self.lowpt_edge[e] = self.lowpt_edge[ei]
(16)                            else:  # add constraints of e_i
(20)                                if not self.add_constraints(ei, e):
(24)                                    return False
(8)                     if e is not None:  # v isn't root
(12)                        self.remove_back_edges(e)
(8)                     return True
(4)                 def add_constraints(self, ei, e):
(8)                     P = ConflictPair()
(8)                     while True:
(12)                        Q = self.S.pop()
(12)                        if not Q.left.empty():
(16)                            Q.swap()
(12)                        if not Q.left.empty():  # not planar
(16)                            return False
(12)                        if self.lowpt[Q.right.low] > self.lowpt[e]:
(16)                            if P.right.empty():  # topmost interval
(20)                                P.right = Q.right.copy()
(16)                            else:
(20)                                self.ref[P.right.low] = Q.right.high
(16)                            P.right.low = Q.right.low
(12)                        else:  # align
(16)                            self.ref[Q.right.low] = self.lowpt_edge[e]
(12)                        if top_of_stack(self.S) == self.stack_bottom[ei]:
(16)                            break
(8)                     while top_of_stack(self.S).left.conflicting(ei, self) or top_of_stack(
(12)                        self.S
(8)                     ).right.conflicting(ei, self):
(12)                        Q = self.S.pop()
(12)                        if Q.right.conflicting(ei, self):
(16)                            Q.swap()
(12)                        if Q.right.conflicting(ei, self):  # not planar
(16)                            return False
(12)                        self.ref[P.right.low] = Q.right.high
(12)                        if Q.right.low is not None:
(16)                            P.right.low = Q.right.low
(12)                        if P.left.empty():  # topmost interval
(16)                            P.left = Q.left.copy()
(12)                        else:
(16)                            self.ref[P.left.low] = Q.left.high
(12)                        P.left.low = Q.left.low
(8)                     if not (P.left.empty() and P.right.empty()):
(12)                        self.S.append(P)
(8)                     return True
(4)                 def remove_back_edges(self, e):
(8)                     u = e[0]
(8)                     while self.S and top_of_stack(self.S).lowest(self) == self.height[u]:
```

```
481: (12)                        P = self.S.pop()
482: (12)                        if P.left.low is not None:
483: (16)                            self.side[P.left.low] = -1
484: (8)                    if self.S:  # one more conflict pair to consider
485: (12)                        P = self.S.pop()
486: (12)                        while P.left.high is not None and P.left.high[1] == u:
487: (16)                            P.left.high = self.ref[P.left.high]
488: (12)                        if P.left.high is None and P.left.low is not None:
489: (16)                            self.ref[P.left.low] = P.right.low
490: (16)                            self.side[P.left.low] = -1
491: (16)                            P.left.low = None
492: (12)                        while P.right.high is not None and P.right.high[1] == u:
493: (16)                            P.right.high = self.ref[P.right.high]
494: (12)                        if P.right.high is None and P.right.low is not None:
495: (16)                            self.ref[P.right.low] = P.left.low
496: (16)                            self.side[P.right.low] = -1
497: (16)                            P.right.low = None
498: (12)                        self.S.append(P)
499: (8)                    if self.lowpt[e] < self.height[u]:  # e has return edge
500: (12)                        hl = top_of_stack(self.S).left.high
501: (12)                        hr = top_of_stack(self.S).right.high
502: (12)                        if hl is not None and (hr is None or self.lowpt[hl] >
self.lowpt[hr]):
503: (16)                            self.ref[e] = hl
504: (12)                        else:
505: (16)                            self.ref[e] = hr
506: (4)                def dfs_embedding(self, v):
507: (8)                    """Completes the embedding."""
508: (8)                    dfs_stack = [v]
509: (8)                    ind = defaultdict(lambda: 0)
510: (8)                    while dfs_stack:
511: (12)                        v = dfs_stack.pop()
512: (12)                        for w in self.ordered_adjs[v][ind[v] :]:
513: (16)                            ind[v] += 1
514: (16)                            ei = (v, w)
515: (16)                            if ei == self.parent_edge[w]:  # tree edge
516: (20)                                self.embedding.add_half_edge_first(w, v)
517: (20)                                self.left_ref[v] = w
518: (20)                                self.right_ref[v] = w
519: (20)                                dfs_stack.append(v)  # revisit v after finishing w
520: (20)                                dfs_stack.append(w)  # visit w next
521: (20)                                break  # handle next node in dfs_stack (i.e. w)
522: (16)                            else:  # back edge
523: (20)                                if self.side[ei] == 1:
524: (24)                                    self.embedding.add_half_edge(w, v,
ccw=self.right_ref[w])
525: (20)                                else:
526: (24)                                    self.embedding.add_half_edge(w, v,
cw=self.left_ref[w])
527: (24)                                    self.left_ref[w] = v
528: (4)                def dfs_embedding_recursive(self, v):
529: (8)                    """Recursive version of :meth:`dfs_embedding`."""
530: (8)                    for w in self.ordered_adjs[v]:
531: (12)                        ei = (v, w)
532: (12)                        if ei == self.parent_edge[w]:  # tree edge
533: (16)                            self.embedding.add_half_edge_first(w, v)
534: (16)                            self.left_ref[v] = w
535: (16)                            self.right_ref[v] = w
536: (16)                            self.dfs_embedding_recursive(w)
537: (12)                        else:  # back edge
538: (16)                            if self.side[ei] == 1:
539: (20)                                self.embedding.add_half_edge(w, v, ccw=self.right_ref[w])
540: (16)                            else:
541: (20)                                self.embedding.add_half_edge(w, v, cw=self.left_ref[w])
542: (20)                                self.left_ref[w] = v
543: (4)                def sign(self, e):
544: (8)                    """Resolve the relative side of an edge to the absolute side."""
545: (8)                    dfs_stack = [e]
546: (8)                    old_ref = defaultdict(lambda: None)
```

```
547: (8)                    while dfs_stack:
548: (12)                       e = dfs_stack.pop()
549: (12)                       if self.ref[e] is not None:
550: (16)                           dfs_stack.append(e)  # revisit e after finishing self.ref[e]
551: (16)                           dfs_stack.append(self.ref[e])  # visit self.ref[e] next
552: (16)                           old_ref[e] = self.ref[e]  # remember value of self.ref[e]
553: (16)                           self.ref[e] = None
554: (12)                       else:
555: (16)                           self.side[e] *= self.side[old_ref[e]]
556: (8)                    return self.side[e]
557: (4)                def sign_recursive(self, e):
558: (8)                    """Recursive version of :meth:`sign`."""
559: (8)                    if self.ref[e] is not None:
560: (12)                       self.side[e] = self.side[e] * self.sign_recursive(self.ref[e])
561: (12)                       self.ref[e] = None
562: (8)                    return self.side[e]
563: (0)            class PlanarEmbedding(nx.DiGraph):
564: (4)                """Represents a planar graph with its planar embedding.
565: (4)                The planar embedding is given by a `combinatorial embedding
566: (4)                <https://en.wikipedia.org/wiki/Graph_embedding#Combinatorial_embedding>`_.
567: (4)                .. note:: `check_planarity` is the preferred way to check if a graph is
planar.
568: (4)                **Neighbor ordering:**
569: (4)                In comparison to a usual graph structure, the embedding also stores the
570: (4)                order of all neighbors for every vertex.
571: (4)                The order of the neighbors can be given in clockwise (cw) direction or
572: (4)                counterclockwise (ccw) direction. This order is stored as edge attributes
573: (4)                in the underlying directed graph. For the edge (u, v) the edge attribute
574: (4)                'cw' is set to the neighbor of u that follows immediately after v in
575: (4)                clockwise direction.
576: (4)                In order for a PlanarEmbedding to be valid it must fulfill multiple
577: (4)                conditions. It is possible to check if these conditions are fulfilled with
578: (4)                the method :meth:`check_structure`.
579: (4)                The conditions are:
580: (4)                * Edges must go in both directions (because the edge attributes differ)
581: (4)                * Every edge must have a 'cw' and 'ccw' attribute which corresponds to a
582: (6)                  correct planar embedding.
583: (4)                As long as a PlanarEmbedding is invalid only the following methods should
584: (4)                be called:
585: (4)                * :meth:`add_half_edge`
586: (4)                * :meth:`connect_components`
587: (4)                Even though the graph is a subclass of nx.DiGraph, it can still be used
588: (4)                for algorithms that require undirected graphs, because the method
589: (4)                :meth:`is_directed` is overridden. This is possible, because a valid
590: (4)                PlanarGraph must have edges in both directions.
591: (4)                **Half edges:**
592: (4)                In methods like `add_half_edge` the term "half-edge" is used, which is
593: (4)                a term that is used in `doubly connected edge lists
594: (4)                <https://en.wikipedia.org/wiki/Doubly_connected_edge_list>`_. It is used
595: (4)                to emphasize that the edge is only in one direction and there exists
596: (4)                another half-edge in the opposite direction.
597: (4)                While conventional edges always have two faces (including outer face) next
598: (4)                to them, it is possible to assign each half-edge *exactly one* face.
599: (4)                For a half-edge (u, v) that is oriented such that u is below v then the
600: (4)                face that belongs to (u, v) is to the right of this half-edge.
601: (4)                See Also
602: (4)                --------
603: (4)                is_planar :
604: (8)                    Preferred way to check if an existing graph is planar.
605: (4)                check_planarity :
606: (8)                    A convenient way to create a `PlanarEmbedding`. If not planar,
607: (8)                    it returns a subgraph that shows this.
608: (4)                Examples
609: (4)                --------
610: (4)                Create an embedding of a star graph (compare `nx.star_graph(3)`):
611: (4)                >>> G = nx.PlanarEmbedding()
612: (4)                >>> G.add_half_edge(0, 1)
613: (4)                >>> G.add_half_edge(0, 2, ccw=1)
614: (4)                >>> G.add_half_edge(0, 3, ccw=2)
```

```
(4)                 >>> G.add_half_edge(1, 0)
(4)                 >>> G.add_half_edge(2, 0)
(4)                 >>> G.add_half_edge(3, 0)
(4)                 Alternatively the same embedding can also be defined in counterclockwise
(4)                 orientation. The following results in exactly the same PlanarEmbedding:
(4)                 >>> G = nx.PlanarEmbedding()
(4)                 >>> G.add_half_edge(0, 1)
(4)                 >>> G.add_half_edge(0, 3, cw=1)
(4)                 >>> G.add_half_edge(0, 2, cw=3)
(4)                 >>> G.add_half_edge(1, 0)
(4)                 >>> G.add_half_edge(2, 0)
(4)                 >>> G.add_half_edge(3, 0)
(4)                 After creating a graph, it is possible to validate that the
PlanarEmbedding
(4)                 object is correct:
(4)                 >>> G.check_structure()
(4)                 """
(4)                 def __init__(self, incoming_graph_data=None, **attr):
(8)                     super().__init__(incoming_graph_data=incoming_graph_data, **attr)
(8)                     self.add_edge = self.__forbidden
(8)                     self.add_edges_from = self.__forbidden
(8)                     self.add_weighted_edges_from = self.__forbidden
(4)                 def __forbidden(self, *args, **kwargs):
(8)                     """Forbidden operation
(8)                     Any edge additions to a PlanarEmbedding should be done using
(8)                     method `add_half_edge`.
(8)                     """
(8)                     raise NotImplementedError(
(12)                        "Use `add_half_edge` method to add edges to a PlanarEmbedding."
(8)                     )
(4)                 def get_data(self):
(8)                     """Converts the adjacency structure into a better readable structure.
(8)                     Returns
(8)                     -------
(8)                     embedding : dict
(12)                        A dict mapping all nodes to a list of neighbors sorted in
(12)                        clockwise order.
(8)                     See Also
(8)                     --------
(8)                     set_data
(8)                     """
(8)                     embedding = {}
(8)                     for v in self:
(12)                        embedding[v] = list(self.neighbors_cw_order(v))
(8)                     return embedding
(4)                 def set_data(self, data):
(8)                     """Inserts edges according to given sorted neighbor list.
(8)                     The input format is the same as the output format of get_data().
(8)                     Parameters
(8)                     ----------
(8)                     data : dict
(12)                        A dict mapping all nodes to a list of neighbors sorted in
(12)                        clockwise order.
(8)                     See Also
(8)                     --------
(8)                     get_data
(8)                     """
(8)                     for v in data:
(12)                        ref = None
(12)                        for w in reversed(data[v]):
(16)                            self.add_half_edge(v, w, cw=ref)
(16)                            ref = w
(4)                 def remove_node(self, n):
(8)                     """Remove node n.
(8)                     Removes the node n and all adjacent edges, updating the
(8)                     PlanarEmbedding to account for any resulting edge removal.
(8)                     Attempting to remove a non-existent node will raise an exception.
(8)                     Parameters
(8)                     ----------
```

```
        n : node
           A node in the graph
        Raises
        ------
        NetworkXError
           If n is not in the graph.
        See Also
        --------
        remove_nodes_from
        """
        try:
            for u in self._pred[n]:
                succs_u = self._succ[u]
                un_cw = succs_u[n]["cw"]
                un_ccw = succs_u[n]["ccw"]
                del succs_u[n]
                del self._pred[u][n]
                if n != un_cw:
                    succs_u[un_cw]["ccw"] = un_ccw
                    succs_u[un_ccw]["cw"] = un_cw
            del self._node[n]
            del self._succ[n]
            del self._pred[n]
        except KeyError as err:  # NetworkXError if n not in self
            raise nx.NetworkXError(
                f"The node {n} is not in the planar embedding."
            ) from err
        nx._clear_cache(self)
    def remove_nodes_from(self, nodes):
        """Remove multiple nodes.
        Parameters
        ----------
        nodes : iterable container
            A container of nodes (list, dict, set, etc.).  If a node
            in the container is not in the graph it is silently ignored.
        See Also
        --------
        remove_node
        Notes
        -----
        When removing nodes from an iterator over the graph you are changing,
        a `RuntimeError` will be raised with message:
        `RuntimeError: dictionary changed size during iteration`. This
        happens when the graph's underlying dictionary is modified during
        iteration. To avoid this error, evaluate the iterator into a separate
        object, e.g. by using `list(iterator_of_nodes)`, and pass this
        object to `G.remove_nodes_from`.
        """
        for n in nodes:
            if n in self._node:
                self.remove_node(n)
    def neighbors_cw_order(self, v):
        """Generator for the neighbors of v in clockwise order.
        Parameters
        ----------
        v : node
        Yields
        ------
        node
        """
        succs = self._succ[v]
        if not succs:
            return
        start_node = next(reversed(succs))
        yield start_node
        current_node = succs[start_node]["cw"]
        while start_node != current_node:
            yield current_node
            current_node = succs[current_node]["cw"]
```

```
    def add_half_edge(self, start_node, end_node, *, cw=None, ccw=None):
        """Adds a half-edge from `start_node` to `end_node`.
        If the half-edge is not the first one out of `start_node`, a reference
        node must be provided either in the clockwise (parameter `cw`) or in
        the counterclockwise (parameter `ccw`) direction. Only one of `cw`/`ccw`
        can be specified (or neither in the case of the first edge).
        Note that specifying a reference in the clockwise (`cw`) direction means
        inserting the new edge in the first counterclockwise position with
        respect to the reference (and vice-versa).
        Parameters
        ----------
        start_node : node
            Start node of inserted edge.
        end_node : node
            End node of inserted edge.
        cw, ccw: node
            End node of reference edge.
            Omit or pass `None` if adding the first out-half-edge of `start_node`.
        Raises
        ------
        NetworkXException
            If the `cw` or `ccw` node is not a successor of `start_node`.
            If `start_node` has successors, but neither `cw` or `ccw` is provided.
            If both `cw` and `ccw` are specified.
        See Also
        --------
        connect_components
        """
        succs = self._succ.get(start_node)
        if succs:
            leftmost_nbr = next(reversed(self._succ[start_node]))
            if cw is not None:
                if cw not in succs:
                    raise nx.NetworkXError("Invalid clockwise reference node.")
                if ccw is not None:
                    raise nx.NetworkXError("Only one of cw/ccw can be specified.")
                ref_ccw = succs[cw]["ccw"]
                super().add_edge(start_node, end_node, cw=cw, ccw=ref_ccw)
                succs[ref_ccw]["cw"] = end_node
                succs[cw]["ccw"] = end_node
                move_leftmost_nbr_to_end = cw != leftmost_nbr
            elif ccw is not None:
                if ccw not in succs:
                    raise nx.NetworkXError("Invalid counterclockwise reference node.")
                ref_cw = succs[ccw]["cw"]
                super().add_edge(start_node, end_node, cw=ref_cw, ccw=ccw)
                succs[ref_cw]["ccw"] = end_node
                succs[ccw]["cw"] = end_node
                move_leftmost_nbr_to_end = True
            else:
                raise nx.NetworkXError(
                    "Node already has out-half-edge(s), either cw or ccw reference node required."
                )
            if move_leftmost_nbr_to_end:
                succs[leftmost_nbr] = succs.pop(leftmost_nbr)
        else:
            if cw is not None or ccw is not None:
                raise nx.NetworkXError("Invalid reference node.")
            super().add_edge(start_node, end_node, ccw=end_node, cw=end_node)
    def check_structure(self):
        """Runs without exceptions if this object is valid.
```

```
(8)                     Checks that the following properties are fulfilled:
(8)                     * Edges go in both directions (because the edge attributes differ).
(8)                     * Every edge has a 'cw' and 'ccw' attribute which corresponds to a
(10)                      correct planar embedding.
(8)                     Running this method verifies that the underlying Graph must be planar.
(8)                     Raises
(8)                     ------
(8)                     NetworkXException
(12)                        This exception is raised with a short explanation if the
(12)                        PlanarEmbedding is invalid.
(8)                     """
(8)                     for v in self:
(12)                        try:
(16)                            sorted_nbrs = set(self.neighbors_cw_order(v))
(12)                        except KeyError as err:
(16)                            msg = f"Bad embedding. Missing orientation for a neighbor of
{v}"
(16)                            raise nx.NetworkXException(msg) from err
(12)                        unsorted_nbrs = set(self[v])
(12)                        if sorted_nbrs != unsorted_nbrs:
(16)                            msg = "Bad embedding. Edge orientations not set correctly."
(16)                            raise nx.NetworkXException(msg)
(12)                        for w in self[v]:
(16)                            if not self.has_edge(w, v):
(20)                                msg = "Bad embedding. Opposite half-edge is missing."
(20)                                raise nx.NetworkXException(msg)
(8)                     counted_half_edges = set()
(8)                     for component in nx.connected_components(self):
(12)                        if len(component) == 1:
(16)                            continue
(12)                        num_nodes = len(component)
(12)                        num_half_edges = 0
(12)                        num_faces = 0
(12)                        for v in component:
(16)                            for w in self.neighbors_cw_order(v):
(20)                                num_half_edges += 1
(20)                                if (v, w) not in counted_half_edges:
(24)                                    num_faces += 1
(24)                                    self.traverse_face(v, w, counted_half_edges)
(12)                        num_edges = num_half_edges // 2  # num_half_edges is even
(12)                        if num_nodes - num_edges + num_faces != 2:
(16)                            msg = "Bad embedding. The graph does not match Euler's
formula"
(16)                            raise nx.NetworkXException(msg)
(4)                 def add_half_edge_ccw(self, start_node, end_node, reference_neighbor):
(8)                     """Adds a half-edge from start_node to end_node.
(8)                     The half-edge is added counter clockwise next to the existing half-
edge
(8)                     (start_node, reference_neighbor).
(8)                     Parameters
(8)                     ----------
(8)                     start_node : node
(12)                        Start node of inserted edge.
(8)                     end_node : node
(12)                        End node of inserted edge.
(8)                     reference_neighbor: node
(12)                        End node of reference edge.
(8)                     Raises
(8)                     ------
(8)                     NetworkXException
(12)                        If the reference_neighbor does not exist.
(8)                     See Also
(8)                     --------
(8)                     add_half_edge
(8)                     add_half_edge_cw
(8)                     connect_components
(8)                     """
(8)                     self.add_half_edge(start_node, end_node, cw=reference_neighbor)
(4)                 def add_half_edge_cw(self, start_node, end_node, reference_neighbor):
```

```
        """Adds a half-edge from start_node to end_node.
        The half-edge is added clockwise next to the existing half-edge
        (start_node, reference_neighbor).
        Parameters
        ----------
        start_node : node
            Start node of inserted edge.
        end_node : node
            End node of inserted edge.
        reference_neighbor: node
            End node of reference edge.
        Raises
        ------
        NetworkXException
            If the reference_neighbor does not exist.
        See Also
        --------
        add_half_edge
        add_half_edge_ccw
        connect_components
        """
        self.add_half_edge(start_node, end_node, ccw=reference_neighbor)
    def remove_edge(self, u, v):
        """Remove the edge between u and v.
        Parameters
        ----------
        u, v : nodes
        Remove the half-edges (u, v) and (v, u) and update the
        edge ordering around the removed edge.
        Raises
        ------
        NetworkXError
        If there is not an edge between u and v.
        See Also
        --------
        remove_edges_from : remove a collection of edges
        """
        try:
            succs_u = self._succ[u]
            succs_v = self._succ[v]
            uv_cw = succs_u[v]["cw"]
            uv_ccw = succs_u[v]["ccw"]
            vu_cw = succs_v[u]["cw"]
            vu_ccw = succs_v[u]["ccw"]
            del succs_u[v]
            del self._pred[v][u]
            del succs_v[u]
            del self._pred[u][v]
            if v != uv_cw:
                succs_u[uv_cw]["ccw"] = uv_ccw
                succs_u[uv_ccw]["cw"] = uv_cw
            if u != vu_cw:
                succs_v[vu_cw]["ccw"] = vu_ccw
                succs_v[vu_ccw]["cw"] = vu_cw
        except KeyError as err:
            raise nx.NetworkXError(
                f"The edge {u}-{v} is not in the planar embedding."
            ) from err
        nx._clear_cache(self)
    def remove_edges_from(self, ebunch):
        """Remove all edges specified in ebunch.
        Parameters
        ----------
        ebunch: list or container of edge tuples
            Each pair of half-edges between the nodes given in the tuples
            will be removed from the graph. The nodes can be passed as:
                - 2-tuples (u, v) half-edges (u, v) and (v, u).
                - 3-tuples (u, v, k) where k is ignored.
        See Also
```

```
        --------
        remove_edge : remove a single edge
        Notes
        -----
        Will fail silently if an edge in ebunch is not in the graph.
        Examples
        --------
        >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> ebunch = [(1, 2), (2, 3)]
        >>> G.remove_edges_from(ebunch)
        """
        for e in ebunch:
            u, v = e[:2]  # ignore edge data
            if u in self._succ and v in self._succ[u]:
                self.remove_edge(u, v)
    def connect_components(self, v, w):
        """Adds half-edges for (v, w) and (w, v) at some position.
        This method should only be called if v and w are in different
        components, or it might break the embedding.
        This especially means that if `connect_components(v, w)`
        is called it is not allowed to call `connect_components(w, v)`
        afterwards. The neighbor orientations in both directions are
        all set correctly after the first call.
        Parameters
        ----------
        v : node
        w : node
        See Also
        --------
        add_half_edge
        """
        if v in self._succ and self._succ[v]:
            ref = next(reversed(self._succ[v]))
        else:
            ref = None
        self.add_half_edge(v, w, cw=ref)
        if w in self._succ and self._succ[w]:
            ref = next(reversed(self._succ[w]))
        else:
            ref = None
        self.add_half_edge(w, v, cw=ref)
    def add_half_edge_first(self, start_node, end_node):
        """Add a half-edge and set end_node as start_node's leftmost neighbor.
        The new edge is inserted counterclockwise with respect to the current
        leftmost neighbor, if there is one.
        Parameters
        ----------
        start_node : node
        end_node : node
        See Also
        --------
        add_half_edge
        connect_components
        """
        succs = self._succ.get(start_node)
        leftmost_nbr = next(reversed(succs)) if succs else None
        self.add_half_edge(start_node, end_node, cw=leftmost_nbr)
    def next_face_half_edge(self, v, w):
        """Returns the following half-edge left of a face.
        Parameters
        ----------
        v : node
        w : node
        Returns
        -------
        half-edge : tuple
        """
        new_node = self[w][v]["ccw"]
        return w, new_node
```

```
1017: (4)               def traverse_face(self, v, w, mark_half_edges=None):
1018: (8)                   """Returns nodes on the face that belong to the half-edge (v, w).
1019: (8)                   The face that is traversed lies to the right of the half-edge (in an
1020: (8)                   orientation where v is below w).
1021: (8)                   Optionally it is possible to pass a set to which all encountered half
1022: (8)                   edges are added. Before calling this method, this set must not include
1023: (8)                   any half-edges that belong to the face.
1024: (8)                   Parameters
1025: (8)                   ----------
1026: (8)                   v : node
1027: (12)                      Start node of half-edge.
1028: (8)                   w : node
1029: (12)                      End node of half-edge.
1030: (8)                   mark_half_edges: set, optional
1031: (12)                      Set to which all encountered half-edges are added.
1032: (8)                   Returns
1033: (8)                   -------
1034: (8)                   face : list
1035: (12)                      A list of nodes that lie on this face.
1036: (8)                   """
1037: (8)                   if mark_half_edges is None:
1038: (12)                      mark_half_edges = set()
1039: (8)                   face_nodes = [v]
1040: (8)                   mark_half_edges.add((v, w))
1041: (8)                   prev_node = v
1042: (8)                   cur_node = w
1043: (8)                   incoming_node = self[v][w]["cw"]
1044: (8)                   while cur_node != v or prev_node != incoming_node:
1045: (12)                      face_nodes.append(cur_node)
1046: (12)                      prev_node, cur_node = self.next_face_half_edge(prev_node,
cur_node)
1047: (12)                      if (prev_node, cur_node) in mark_half_edges:
1048: (16)                          raise nx.NetworkXException("Bad planar embedding. Impossible
face.")
1049: (12)                      mark_half_edges.add((prev_node, cur_node))
1050: (8)                   return face_nodes
1051: (4)               def is_directed(self):
1052: (8)                   """A valid PlanarEmbedding is undirected.
1053: (8)                   All reverse edges are contained, i.e. for every existing
1054: (8)                   half-edge (v, w) the half-edge in the opposite direction (w, v) is
also
1055: (8)                   contained.
1056: (8)                   """
1057: (8)                   return False
1058: (4)               def copy(self, as_view=False):
1059: (8)                   if as_view is True:
1060: (12)                      return nx.graphviews.generic_graph_view(self)
1061: (8)                   G = self.__class__()
1062: (8)                   G.graph.update(self.graph)
1063: (8)                   G.add_nodes_from((n, d.copy()) for n, d in self._node.items())
1064: (8)                   super(self.__class__, G).add_edges_from(
1065: (12)                      (u, v, datadict.copy())
1066: (12)                      for u, nbrs in self._adj.items()
1067: (12)                      for v, datadict in nbrs.items()
1068: (8)                   )
1069: (8)                   return G
```

File 44 - polynomials.py:

```
1: (0)              """Provides algorithms supporting the computation of graph polynomials.
2: (0)              Graph polynomials are polynomial-valued graph invariants that encode a wide
3: (0)              variety of structural information. Examples include the Tutte polynomial,
4: (0)              chromatic polynomial, characteristic polynomial, and matching polynomial. An
5: (0)              extensive treatment is provided in [1]_.
6: (0)              For a simple example, the
`~sympy.matrices.matrices.MatrixDeterminant.charpoly`
7: (0)              method can be used to compute the characteristic polynomial from the adjacency
```

```
matrix of a graph. Consider the complete graph ``K_4``:
>>> import sympy
>>> x = sympy.Symbol("x")
>>> G = nx.complete_graph(4)
>>> A = nx.adjacency_matrix(G)
>>> M = sympy.SparseMatrix(A.todense())
>>> M.charpoly(x).as_expr()
x**4 - 6*x**2 - 8*x - 3
.. [1] Y. Shi, M. Dehmer, X. Li, I. Gutman,
   "Graph Polynomials"
"""
from collections import deque
import networkx as nx
from networkx.utils import not_implemented_for
__all__ = ["tutte_polynomial", "chromatic_polynomial"]
@not_implemented_for("directed")
@nx._dispatchable
def tutte_polynomial(G):
    r"""Returns the Tutte polynomial of `G`
    This function computes the Tutte polynomial via an iterative version of
    the deletion-contraction algorithm.
    The Tutte polynomial `T_G(x, y)` is a fundamental graph polynomial invariant in
    two variables. It encodes a wide array of information related to the
    edge-connectivity of a graph; "Many problems about graphs can be reduced to
    problems of finding and evaluating the Tutte polynomial at certain values" [1]_.
    In fact, every deletion-contraction-expressible feature of a graph is a
    specialization of the Tutte polynomial [2]_ (see Notes for examples).
    There are several equivalent definitions; here are three:
    Def 1 (rank-nullity expansion): For `G` an undirected graph, `n(G)` the
    number of vertices of `G`, `E` the edge set of `G`, `V` the vertex set of
    `G`, and `c(A)` the number of connected components of the graph with vertex
    set `V` and edge set `A` [3]_:
    .. math::
        T_G(x, y) = \sum_{A \in E} (x-1)^{c(A) - c(E)} (y-1)^{c(A) + |A| - n(G)}
    Def 2 (spanning tree expansion): Let `G` be an undirected graph, `T` a spanning
    tree of `G`, and `E` the edge set of `G`. Let `E` have an arbitrary strict
    linear order `L`. Let `B_e` be the unique minimal nonempty edge cut of
    $E \setminus T \cup {e}$. An edge `e` is internally active with respect to
    `T` and `L` if `e` is the least edge in `B_e` according to the linear order
    `L`. The internal activity of `T` (denoted `i(T)`) is the number of edges
    in $E \setminus T$ that are internally active with respect to `T` and `L`.
    Let `P_e` be the unique path in $T \cup {e}$ whose source and target vertex
    are the same. An edge `e` is externally active with respect to `T` and `L`
    if `e` is the least edge in `P_e` according to the linear order `L`. The
    external activity of `T` (denoted `e(T)`) is the number of edges in
    $E \setminus T$ that are externally active with respect to `T` and `L`.
    Then [4]_ [5]_:
    .. math::
        T_G(x, y) = \sum_{T \text{ a spanning tree of } G} x^{i(T)} y^{e(T)}
    Def 3 (deletion-contraction recurrence): For `G` an undirected graph, `G-e`
    the graph obtained from `G` by deleting edge `e`, `G/e` the graph obtained
    from `G` by contracting edge `e`, `k(G)` the number of cut-edges of `G`,
    and `l(G)` the number of self-loops of `G`:
    .. math::
        T_G(x, y) = \begin{cases}
           x^{k(G)} y^{l(G)}, & \text{if all edges are cut-edges or self-loops} \\
           T_{G-e}(x, y) + T_{G/e}(x, y), & \text{otherwise, for an arbitrary edge $e$ not a cut-edge or loop}
        \end{cases}
```

```
66: (4)                 Parameters
67: (4)                 ----------
68: (4)                 G : NetworkX graph
69: (4)                 Returns
70: (4)                 -------
71: (4)                 instance of `sympy.core.add.Add`
72: (8)                     A Sympy expression representing the Tutte polynomial for `G`.
73: (4)                 Examples
74: (4)                 --------
75: (4)                 >>> C = nx.cycle_graph(5)
76: (4)                 >>> nx.tutte_polynomial(C)
77: (4)                 x**4 + x**3 + x**2 + x + y
78: (4)                 >>> D = nx.diamond_graph()
79: (4)                 >>> nx.tutte_polynomial(D)
80: (4)                 x**3 + 2*x**2 + 2*x*y + x + y**2 + y
81: (4)                 Notes
82: (4)                 -----
83: (4)                 Some specializations of the Tutte polynomial:
84: (4)                 - `T_G(1, 1)` counts the number of spanning trees of `G`
85: (4)                 - `T_G(1, 2)` counts the number of connected spanning subgraphs of `G`
86: (4)                 - `T_G(2, 1)` counts the number of spanning forests in `G`
87: (4)                 - `T_G(0, 2)` counts the number of strong orientations of `G`
88: (4)                 - `T_G(2, 0)` counts the number of acyclic orientations of `G`
89: (4)                 Edge contraction is defined and deletion-contraction is introduced in
[6]_.
90: (4)                 Combinatorial meaning of the coefficients is introduced in [7]_.
91: (4)                 Universality, properties, and applications are discussed in [8]_.
92: (4)                 Practically, up-front computation of the Tutte polynomial may be useful
when
93: (4)                 users wish to repeatedly calculate edge-connectivity-related information
94: (4)                 about one or more graphs.
95: (4)                 References
96: (4)                 ----------
97: (4)                 .. [1] M. Brandt,
98: (7)                    "The Tutte Polynomial."
99: (7)                    Talking About Combinatorial Objects Seminar, 2015
100: (7)                   https://math.berkeley.edu/~brandtm/talks/tutte.pdf
101: (4)                .. [2] A. Björklund, T. Husfeldt, P. Kaski, M. Koivisto,
102: (7)                   "Computing the Tutte polynomial in vertex-exponential time"
103: (7)                   49th Annual IEEE Symposium on Foundations of Computer Science, 2008
104: (7)                   https://ieeexplore.ieee.org/abstract/document/4691000
105: (4)                .. [3] Y. Shi, M. Dehmer, X. Li, I. Gutman,
106: (7)                   "Graph Polynomials," p. 14
107: (4)                .. [4] Y. Shi, M. Dehmer, X. Li, I. Gutman,
108: (7)                   "Graph Polynomials," p. 46
109: (4)                .. [5] A. Nešetril, J. Goodall,
110: (7)                   "Graph invariants, homomorphisms, and the Tutte polynomial"
111: (7)                   https://iuuk.mff.cuni.cz/~andrew/Tutte.pdf
112: (4)                .. [6] D. B. West,
113: (7)                   "Introduction to Graph Theory," p. 84
114: (4)                .. [7] G. Coutinho,
115: (7)                   "A brief introduction to the Tutte polynomial"
116: (7)                   Structural Analysis of Complex Networks, 2011
117: (7)
https://homepages.dcc.ufmg.br/~gabriel/seminars/coutinho_tuttepolynomial_seminar.pdf
118: (4)                .. [8] J. A. Ellis-Monaghan, C. Merino,
119: (7)                   "Graph polynomials and their applications I: The Tutte polynomial"
120: (7)                   Structural Analysis of Complex Networks, 2011
121: (7)                   https://arxiv.org/pdf/0803.3079.pdf
122: (4)                """
123: (4)                import sympy
124: (4)                x = sympy.Symbol("x")
125: (4)                y = sympy.Symbol("y")
126: (4)                stack = deque()
127: (4)                stack.append(nx.MultiGraph(G))
128: (4)                polynomial = 0
129: (4)                while stack:
130: (8)                    G = stack.pop()
131: (8)                    bridges = set(nx.bridges(G))
```

```
(8)                     e = None
(8)                     for i in G.edges:
(12)                        if (i[0], i[1]) not in bridges and i[0] != i[1]:
(16)                            e = i
(16)                            break
(8)                     if not e:
(12)                        loops = list(nx.selfloop_edges(G, keys=True))
(12)                        polynomial += x ** len(bridges) * y ** len(loops)
(8)                     else:
(12)                        C = nx.contracted_edge(G, e, self_loops=True)
(12)                        C.remove_edge(e[0], e[0])
(12)                        G.remove_edge(*e)
(12)                        stack.append(G)
(12)                        stack.append(C)
(4)                 return sympy.simplify(polynomial)
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable
(0)             def chromatic_polynomial(G):
(4)                 r"""Returns the chromatic polynomial of `G`
(4)                 This function computes the chromatic polynomial via an iterative version
of
(4)                 the deletion-contraction algorithm.
(4)                 The chromatic polynomial `X_G(x)` is a fundamental graph polynomial
(4)                 invariant in one variable. Evaluating `X_G(k)` for an natural number `k`
(4)                 enumerates the proper k-colorings of `G`.
(4)                 There are several equivalent definitions; here are three:
(4)                 Def 1 (explicit formula):
(4)                 For `G` an undirected graph, `c(G)` the number of connected components of
(4)                 `G`, `E` the edge set of `G`, and `G(S)` the spanning subgraph of `G` with
(4)                 edge set `S` [1]_:
(4)                 .. math::
(8)                     X_G(x) = \sum_{S \subseteq E} (-1)^{|S|} x^{c(G(S))}
(4)                 Def 2 (interpolating polynomial):
(4)                 For `G` an undirected graph, `n(G)` the number of vertices of `G`, `k_0 =
0`,
(4)                 and `k_i` the number of distinct ways to color the vertices of `G` with
`i`
(4)                 unique colors (for `i` a natural number at most `n(G)`), `X_G(x)` is the
(4)                 unique Lagrange interpolating polynomial of degree `n(G)` through the
points
(4)                 `(0, k_0), (1, k_1), \dots, (n(G), k_{n(G)})` [2]_.
(4)                 Def 3 (chromatic recurrence):
(4)                 For `G` an undirected graph, `G-e` the graph obtained from `G` by deleting
(4)                 edge `e`, `G/e` the graph obtained from `G` by contracting edge `e`,
`n(G)`
(4)                 the number of vertices of `G`, and `e(G)` the number of edges of `G` [3]_:
(4)                 .. math::
(8)                     X_G(x) = \begin{cases}
(11)                       x^{n(G)}, & \text{if $e(G)=0$} \\
(11)                       X_{G-e}(x) - X_{G/e}(x), & \text{otherwise, for an arbitrary edge
$e$}
(8)                     \end{cases}
(4)                 This formulation is also known as the Fundamental Reduction Theorem [4]_.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 Returns
(4)                 -------
(4)                 instance of `sympy.core.add.Add`
(8)                     A Sympy expression representing the chromatic polynomial for `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> C = nx.cycle_graph(5)
(4)                 >>> nx.chromatic_polynomial(C)
(4)                 x**5 - 5*x**4 + 10*x**3 - 10*x**2 + 4*x
(4)                 >>> G = nx.complete_graph(4)
(4)                 >>> nx.chromatic_polynomial(G)
(4)                 x**4 - 6*x**3 + 11*x**2 - 6*x
(4)                 Notes
```

```
195: (4)                -----
196: (4)                Interpretation of the coefficients is discussed in [5]_. Several special
197: (4)                cases are listed in [2]_.
198: (4)                The chromatic polynomial is a specialization of the Tutte polynomial; in
199: (4)                particular, ``X_G(x) = T_G(x, 0)`` [6]_.
200: (4)                The chromatic polynomial may take negative arguments, though evaluations
201: (4)                may not have chromatic interpretations. For instance, ``X_G(-1)``
enumerates
202: (4)                the acyclic orientations of `G` [7]_.
203: (4)                References
204: (4)                ----------
205: (4)                .. [1] D. B. West,
206: (7)                   "Introduction to Graph Theory," p. 222
207: (4)                .. [2] E. W. Weisstein
208: (7)                   "Chromatic Polynomial"
209: (7)                   MathWorld--A Wolfram Web Resource
210: (7)                   https://mathworld.wolfram.com/ChromaticPolynomial.html
211: (4)                .. [3] D. B. West,
212: (7)                   "Introduction to Graph Theory," p. 221
213: (4)                .. [4] J. Zhang, J. Goodall,
214: (7)                   "An Introduction to Chromatic Polynomials"
215: (7)                   https://math.mit.edu/~apost/courses/18.204_2018/Julie_Zhang_paper.pdf
216: (4)                .. [5] R. C. Read,
217: (7)                   "An Introduction to Chromatic Polynomials"
218: (7)                   Journal of Combinatorial Theory, 1968
219: (7)                   https://math.berkeley.edu/~mrklug/ReadChromatic.pdf
220: (4)                .. [6] W. T. Tutte,
221: (7)                   "Graph-polynomials"
222: (7)                   Advances in Applied Mathematics, 2004
223: (7)                   https://www.sciencedirect.com/science/article/pii/S0196885803000411
224: (4)                .. [7] R. P. Stanley,
225: (7)                   "Acyclic orientations of graphs"
226: (7)                   Discrete Mathematics, 2006
227: (7)                   https://math.mit.edu/~rstan/pubs/pubfiles/18.pdf
228: (4)                """
229: (4)                import sympy
230: (4)                x = sympy.Symbol("x")
231: (4)                stack = deque()
232: (4)                stack.append(nx.MultiGraph(G, contraction_idx=0))
233: (4)                polynomial = 0
234: (4)                while stack:
235: (8)                    G = stack.pop()
236: (8)                    edges = list(G.edges)
237: (8)                    if not edges:
238: (12)                       polynomial += (-1) ** G.graph["contraction_idx"] * x ** len(G)
239: (8)                    else:
240: (12)                       e = edges[0]
241: (12)                       C = nx.contracted_edge(G, e, self_loops=True)
242: (12)                       C.graph["contraction_idx"] = G.graph["contraction_idx"] + 1
243: (12)                       C.remove_edge(e[0], e[0])
244: (12)                       G.remove_edge(*e)
245: (12)                       stack.append(G)
246: (12)                       stack.append(C)
247: (4)                return polynomial

----------------------------------------

File 45 - reciprocity.py:

1: (0)              """Algorithms to calculate reciprocity in a directed graph."""
2: (0)              import networkx as nx
3: (0)              from networkx import NetworkXError
4: (0)              from ..utils import not_implemented_for
5: (0)              __all__ = ["reciprocity", "overall_reciprocity"]
6: (0)              @not_implemented_for("undirected", "multigraph")
7: (0)              @nx._dispatchable
8: (0)              def reciprocity(G, nodes=None):
9: (4)                  r"""Compute the reciprocity in a directed graph.
10: (4)                 The reciprocity of a directed graph is defined as the ratio
```

```
11: (4)                 of the number of edges pointing in both directions to the total
12: (4)                 number of edges in the graph.
13: (4)                 Formally, $r = |{(u,v) \in G|(v,u) \in G}| / |{(u,v) \in G}|$.
14: (4)                 The reciprocity of a single node u is defined similarly,
15: (4)                 it is the ratio of the number of edges in both directions to
16: (4)                 the total number of edges attached to node u.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : graph
20: (7)                    A networkx directed graph
21: (4)                 nodes : container of nodes, optional (default=whole graph)
22: (7)                    Compute reciprocity for nodes in this container.
23: (4)                 Returns
24: (4)                 -------
25: (4)                 out : dictionary
26: (7)                    Reciprocity keyed by node label.
27: (4)                 Notes
28: (4)                 -----
29: (4)                 The reciprocity is not defined for isolated nodes.
30: (4)                 In such cases this function will return None.
31: (4)                 """
32: (4)                 if nodes is None:
33: (8)                     return overall_reciprocity(G)
34: (4)                 if nodes in G:
35: (8)                     reciprocity = next(_reciprocity_iter(G, nodes))[1]
36: (8)                     if reciprocity is None:
37: (12)                        raise NetworkXError("Not defined for isolated nodes.")
38: (8)                     else:
39: (12)                        return reciprocity
40: (4)                 return dict(_reciprocity_iter(G, nodes))
41: (0)             def _reciprocity_iter(G, nodes):
42: (4)                 """Return an iterator of (node, reciprocity)."""
43: (4)                 n = G.nbunch_iter(nodes)
44: (4)                 for node in n:
45: (8)                     pred = set(G.predecessors(node))
46: (8)                     succ = set(G.successors(node))
47: (8)                     overlap = pred & succ
48: (8)                     n_total = len(pred) + len(succ)
49: (8)                     if n_total == 0:
50: (12)                        yield (node, None)
51: (8)                     else:
52: (12)                        reciprocity = 2 * len(overlap) / n_total
53: (12)                        yield (node, reciprocity)
54: (0)             @not_implemented_for("undirected", "multigraph")
55: (0)             @nx._dispatchable
56: (0)             def overall_reciprocity(G):
57: (4)                 """Compute the reciprocity for the whole graph.
58: (4)                 See the doc of reciprocity for the definition.
59: (4)                 Parameters
60: (4)                 ----------
61: (4)                 G : graph
62: (7)                    A networkx graph
63: (4)                 """
64: (4)                 n_all_edge = G.number_of_edges()
65: (4)                 n_overlap_edge = (n_all_edge - G.to_undirected().number_of_edges()) * 2
66: (4)                 if n_all_edge == 0:
67: (8)                     raise NetworkXError("Not defined for empty graphs")
68: (4)                 return n_overlap_edge / n_all_edge
```

File 46 - regular.py:

```
1: (0)              """Functions for computing and verifying regular graphs."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for
4: (0)              __all__ = ["is_regular", "is_k_regular", "k_factor"]
5: (0)              @nx._dispatchable
6: (0)              def is_regular(G):
```

```
(4)                 """Determines whether the graph ``G`` is a regular graph.
(4)                 A regular graph is a graph where each vertex has the same degree. A
(4)                 regular digraph is a graph where the indegree and outdegree of each
(4)                 vertex are equal.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     Whether the given graph or digraph is regular.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 4), (4, 1)])
(4)                 >>> nx.is_regular(G)
(4)                 True
(4)                 """
(4)                 if len(G) == 0:
(8)                     raise nx.NetworkXPointlessConcept("Graph has no nodes.")
(4)                 n1 = nx.utils.arbitrary_element(G)
(4)                 if not G.is_directed():
(8)                     d1 = G.degree(n1)
(8)                     return all(d1 == d for _, d in G.degree)
(4)                 else:
(8)                     d_in = G.in_degree(n1)
(8)                     in_regular = all(d_in == d for _, d in G.in_degree)
(8)                     d_out = G.out_degree(n1)
(8)                     out_regular = all(d_out == d for _, d in G.out_degree)
(8)                     return in_regular and out_regular
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable
(0)             def is_k_regular(G, k):
(4)                 """Determines whether the graph ``G`` is a k-regular graph.
(4)                 A k-regular graph is a graph where each vertex has degree k.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     Whether the given graph is k-regular.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
(4)                 >>> nx.is_k_regular(G, k=3)
(4)                 False
(4)                 """
(4)                 return all(d == k for n, d in G.degree)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(preserve_edge_attrs=True, returns_graph=True)
(0)             def k_factor(G, k, matching_weight="weight"):
(4)                 """Compute a k-factor of G
(4)                 A k-factor of a graph is a spanning k-regular subgraph.
(4)                 A spanning k-regular subgraph of G is a subgraph that contains
(4)                 each vertex of G and a subset of the edges of G such that each
(4)                 vertex has degree k.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(6)                   Undirected graph
(4)                 matching_weight: string, optional (default='weight')
(7)                    Edge data key corresponding to the edge weight.
(7)                    Used for finding the max-weighted perfect matching.
(7)                    If key not found, uses 1 as weight.
(4)                 Returns
(4)                 -------
(4)                 G2 : NetworkX graph
(8)                     A k-factor of G
```

```
 76: (4)                 Examples
 77: (4)                 --------
 78: (4)                 >>> G = nx.Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
 79: (4)                 >>> G2 = nx.k_factor(G, k=1)
 80: (4)                 >>> G2.edges()
 81: (4)                 EdgeView([(1, 2), (3, 4)])
 82: (4)                 References
 83: (4)                 ----------
 84: (4)                 .. [1] "An algorithm for computing simple k-factors.",
 85: (7)                    Meijer, Henk, Yurai Núñez-Rodríguez, and David Rappaport,
 86: (7)                    Information processing letters, 2009.
 87: (4)                 """
 88: (4)                 from networkx.algorithms.matching import is_perfect_matching,
max_weight_matching
 89: (4)                 class LargeKGadget:
 90: (8)                     def __init__(self, k, degree, node, g):
 91: (12)                        self.original = node
 92: (12)                        self.g = g
 93: (12)                        self.k = k
 94: (12)                        self.degree = degree
 95: (12)                        self.outer_vertices = [(node, x) for x in range(degree)]
 96: (12)                        self.core_vertices = [(node, x + degree) for x in range(degree -
k)]
 97: (8)                     def replace_node(self):
 98: (12)                        adj_view = self.g[self.original]
 99: (12)                        neighbors = list(adj_view.keys())
100: (12)                        edge_attrs = list(adj_view.values())
101: (12)                        for outer, neighbor, edge_attrs in zip(
102: (16)                            self.outer_vertices, neighbors, edge_attrs
103: (12)                        ):
104: (16)                            self.g.add_edge(outer, neighbor, **edge_attrs)
105: (12)                        for core in self.core_vertices:
106: (16)                            for outer in self.outer_vertices:
107: (20)                                self.g.add_edge(core, outer)
108: (12)                        self.g.remove_node(self.original)
109: (8)                     def restore_node(self):
110: (12)                        self.g.add_node(self.original)
111: (12)                        for outer in self.outer_vertices:
112: (16)                            adj_view = self.g[outer]
113: (16)                            for neighbor, edge_attrs in list(adj_view.items()):
114: (20)                                if neighbor not in self.core_vertices:
115: (24)                                    self.g.add_edge(self.original, neighbor, **edge_attrs)
116: (24)                                    break
117: (12)                        g.remove_nodes_from(self.outer_vertices)
118: (12)                        g.remove_nodes_from(self.core_vertices)
119: (4)                 class SmallKGadget:
120: (8)                     def __init__(self, k, degree, node, g):
121: (12)                        self.original = node
122: (12)                        self.k = k
123: (12)                        self.degree = degree
124: (12)                        self.g = g
125: (12)                        self.outer_vertices = [(node, x) for x in range(degree)]
126: (12)                        self.inner_vertices = [(node, x + degree) for x in range(degree)]
127: (12)                        self.core_vertices = [(node, x + 2 * degree) for x in range(k)]
128: (8)                     def replace_node(self):
129: (12)                        adj_view = self.g[self.original]
130: (12)                        for outer, inner, (neighbor, edge_attrs) in zip(
131: (16)                            self.outer_vertices, self.inner_vertices,
list(adj_view.items())
132: (12)                        ):
133: (16)                            self.g.add_edge(outer, inner)
134: (16)                            self.g.add_edge(outer, neighbor, **edge_attrs)
135: (12)                        for core in self.core_vertices:
136: (16)                            for inner in self.inner_vertices:
137: (20)                                self.g.add_edge(core, inner)
138: (12)                        self.g.remove_node(self.original)
139: (8)                     def restore_node(self):
140: (12)                        self.g.add_node(self.original)
141: (12)                        for outer in self.outer_vertices:
```

```
142: (16)                            adj_view = self.g[outer]
143: (16)                            for neighbor, edge_attrs in adj_view.items():
144: (20)                                if neighbor not in self.core_vertices:
145: (24)                                    self.g.add_edge(self.original, neighbor, **edge_attrs)
146: (24)                                    break
147: (12)                        self.g.remove_nodes_from(self.outer_vertices)
148: (12)                        self.g.remove_nodes_from(self.inner_vertices)
149: (12)                        self.g.remove_nodes_from(self.core_vertices)
150: (4)                if any(d < k for _, d in G.degree):
151: (8)                    raise nx.NetworkXUnfeasible("Graph contains a vertex with degree less
than k")
152: (4)                g = G.copy()
153: (4)                gadgets = []
154: (4)                for node, degree in list(g.degree):
155: (8)                    if k < degree / 2.0:
156: (12)                       gadget = SmallKGadget(k, degree, node, g)
157: (8)                    else:
158: (12)                       gadget = LargeKGadget(k, degree, node, g)
159: (8)                    gadget.replace_node()
160: (8)                    gadgets.append(gadget)
161: (4)                matching = max_weight_matching(g, maxcardinality=True,
weight=matching_weight)
162: (4)                if not is_perfect_matching(g, matching):
163: (8)                    raise nx.NetworkXUnfeasible(
164: (12)                       "Cannot find k-factor because no perfect matching exists"
165: (8)                    )
166: (4)                for edge in g.edges():
167: (8)                    if edge not in matching and (edge[1], edge[0]) not in matching:
168: (12)                       g.remove_edge(edge[0], edge[1])
169: (4)                for gadget in gadgets:
170: (8)                    gadget.restore_node()
171: (4)                return g
```

File 47 - richclub.py:

```
1: (0)              """Functions for computing rich-club coefficients."""
2: (0)              from itertools import accumulate
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["rich_club_coefficient"]
6: (0)              @not_implemented_for("directed")
7: (0)              @not_implemented_for("multigraph")
8: (0)              @nx._dispatchable
9: (0)              def rich_club_coefficient(G, normalized=True, Q=100, seed=None):
10: (4)                 r"""Returns the rich-club coefficient of the graph `G`.
11: (4)                 For each degree *k*, the *rich-club coefficient* is the ratio of the
12: (4)                 number of actual to the number of potential edges for nodes with
13: (4)                 degree greater than *k*:
14: (4)                 .. math::
15: (8)                     \phi(k) = \frac{2 E_k}{N_k (N_k - 1)}
16: (4)                 where `N_k` is the number of nodes with degree larger than *k*, and
17: (4)                 `E_k` is the number of edges among those nodes.
18: (4)                 Parameters
19: (4)                 ----------
20: (4)                 G : NetworkX graph
21: (8)                     Undirected graph with neither parallel edges nor self-loops.
22: (4)                 normalized : bool (optional)
23: (8)                     Normalize using randomized network as in [1]_
24: (4)                 Q : float (optional, default=100)
25: (8)                     If `normalized` is True, perform `Q * m` double-edge
26: (8)                     swaps, where `m` is the number of edges in `G`, to use as a
27: (8)                     null-model for normalization.
28: (4)                 seed : integer, random_state, or None (default)
29: (8)                     Indicator of random number generation state.
30: (8)                     See :ref:`Randomness<randomness>`.
31: (4)                 Returns
32: (4)                 -------
```

```
    rc : dictionary
       A dictionary, keyed by degree, with rich-club coefficient values.
    Raises
    ------
    NetworkXError
        If `G` has fewer than four nodes and ``normalized=True``.
        A randomly sampled graph for normalization cannot be generated in this case.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (4, 5)])
    >>> rc = nx.rich_club_coefficient(G, normalized=False, seed=42)
    >>> rc[0]
    0.4
    Notes
    -----
    The rich club definition and algorithm are found in [1]_.  This
    algorithm ignores any edge weights and is not defined for directed
    graphs or graphs with parallel edges or self loops.
    Normalization is done by computing the rich club coefficient for a randomly
    sampled graph with the same degree distribution as `G` by
    repeatedly swapping the endpoints of existing edges. For graphs with fewer than 4
    nodes, it is not possible to generate a random graph with a prescribed
    degree distribution, as the degree distribution fully determines the graph
    (hence making the coefficients trivially normalized to 1).
    This function raises an exception in this case.
    Estimates for appropriate values of `Q` are found in [2]_.
    References
    ----------
    .. [1] Julian J. McAuley, Luciano da Fontoura Costa,
       and Tibério S. Caetano,
       "The rich-club phenomenon across complex network hierarchies",
       Applied Physics Letters Vol 91 Issue 8, August 2007.
       https://arxiv.org/abs/physics/0701290
    .. [2] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, U. Alon,
       "Uniform generation of random graphs with arbitrary degree
       sequences", 2006. https://arxiv.org/abs/cond-mat/0312028
    """
    if nx.number_of_selfloops(G) > 0:
        raise Exception(
            "rich_club_coefficient is not implemented for graphs with self loops."
        )
    rc = _compute_rc(G)
    if normalized:
        R = G.copy()
        E = R.number_of_edges()
        nx.double_edge_swap(R, Q * E, max_tries=Q * E * 10, seed=seed)
        rcran = _compute_rc(R)
        rc = {k: v / rcran[k] for k, v in rc.items()}
    return rc
def _compute_rc(G):
    """Returns the rich-club coefficient for each degree in the graph
    `G`.
    `G` is an undirected graph without multiedges.
    Returns a dictionary mapping degree to rich-club coefficient for
    that degree.
    """
    deghist = nx.degree_histogram(G)
    total = sum(deghist)
    nks = (total - cs for cs in accumulate(deghist) if total - cs > 1)
    edge_degrees = sorted((sorted(map(G.degree, e)) for e in G.edges()), reverse=True)
    ek = G.number_of_edges()
    if ek == 0:
        return {}
    k1, k2 = edge_degrees.pop()
```

```
97: (4)                 rc = {}
98: (4)                 for d, nk in enumerate(nks):
99: (8)                     while k1 <= d:
100: (12)                       if len(edge_degrees) == 0:
101: (16)                           ek = 0
102: (16)                           break
103: (12)                       k1, k2 = edge_degrees.pop()
104: (12)                       ek -= 1
105: (8)                    rc[d] = 2 * ek / (nk * (nk - 1))
106: (4)                return rc
```

File 48 - similarity.py:

```
1: (0)              """ Functions measuring similarity using graph edit distance.
2: (0)              The graph edit distance is the number of edge/node changes needed
3: (0)              to make two graphs isomorphic.
4: (0)              The default algorithm/implementation is sub-optimal for some graphs.
5: (0)              The problem of finding the exact Graph Edit Distance (GED) is NP-hard
6: (0)              so it is often slow. If the simple interface `graph_edit_distance`
7: (0)              takes too long for your graph, try `optimize_graph_edit_distance`
8: (0)              and/or `optimize_edit_paths`.
9: (0)              At the same time, I encourage capable people to investigate
10: (0)             alternative GED algorithms, in order to improve the choices available.
11: (0)             """
12: (0)             import math
13: (0)             import time
14: (0)             import warnings
15: (0)             from dataclasses import dataclass
16: (0)             from itertools import product
17: (0)             import networkx as nx
18: (0)             from networkx.utils import np_random_state
19: (0)             __all__ = [
20: (4)                 "graph_edit_distance",
21: (4)                 "optimal_edit_paths",
22: (4)                 "optimize_graph_edit_distance",
23: (4)                 "optimize_edit_paths",
24: (4)                 "simrank_similarity",
25: (4)                 "panther_similarity",
26: (4)                 "generate_random_paths",
27: (0)             ]
28: (0)             def debug_print(*args, **kwargs):
29: (4)                 print(*args, **kwargs)
30: (0)             @nx._dispatchable(
31: (4)                 graphs={"G1": 0, "G2": 1}, preserve_edge_attrs=True,
preserve_node_attrs=True
32: (0)             )
33: (0)             def graph_edit_distance(
34: (4)                 G1,
35: (4)                 G2,
36: (4)                 node_match=None,
37: (4)                 edge_match=None,
38: (4)                 node_subst_cost=None,
39: (4)                 node_del_cost=None,
40: (4)                 node_ins_cost=None,
41: (4)                 edge_subst_cost=None,
42: (4)                 edge_del_cost=None,
43: (4)                 edge_ins_cost=None,
44: (4)                 roots=None,
45: (4)                 upper_bound=None,
46: (4)                 timeout=None,
47: (0)             ):
48: (4)                 """Returns GED (graph edit distance) between graphs G1 and G2.
49: (4)                 Graph edit distance is a graph similarity measure analogous to
50: (4)                 Levenshtein distance for strings.  It is defined as minimum cost
51: (4)                 of edit path (sequence of node and edge edit operations)
52: (4)                 transforming graph G1 to graph isomorphic to G2.
53: (4)                 Parameters
```

```
    ----------
    G1, G2: graphs
        The two graphs G1 and G2 must be of the same type.
    node_match : callable
        A function that returns True if node n1 in G1 and n2 in G2
        should be considered equal during matching.
        The function will be called like
           node_match(G1.nodes[n1], G2.nodes[n2]).
        That is, the function will receive the node attribute
        dictionaries for n1 and n2 as inputs.
        Ignored if node_subst_cost is specified.  If neither
        node_match nor node_subst_cost are specified then node
        attributes are not considered.
    edge_match : callable
        A function that returns True if the edge attribute dictionaries
        for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should
        be considered equal during matching.
        The function will be called like
           edge_match(G1[u1][v1], G2[u2][v2]).
        That is, the function will receive the edge attribute
        dictionaries of the edges under consideration.
        Ignored if edge_subst_cost is specified.  If neither
        edge_match nor edge_subst_cost are specified then edge
        attributes are not considered.
    node_subst_cost, node_del_cost, node_ins_cost : callable
        Functions that return the costs of node substitution, node
        deletion, and node insertion, respectively.
        The functions will be called like
           node_subst_cost(G1.nodes[n1], G2.nodes[n2]),
           node_del_cost(G1.nodes[n1]),
           node_ins_cost(G2.nodes[n2]).
        That is, the functions will receive the node attribute
        dictionaries as inputs.  The functions are expected to return
        positive numeric values.
        Function node_subst_cost overrides node_match if specified.
        If neither node_match nor node_subst_cost are specified then
        default node substitution cost of 0 is used (node attributes
        are not considered during matching).
        If node_del_cost is not specified then default node deletion
        cost of 1 is used.  If node_ins_cost is not specified then
        default node insertion cost of 1 is used.
    edge_subst_cost, edge_del_cost, edge_ins_cost : callable
        Functions that return the costs of edge substitution, edge
        deletion, and edge insertion, respectively.
        The functions will be called like
           edge_subst_cost(G1[u1][v1], G2[u2][v2]),
           edge_del_cost(G1[u1][v1]),
           edge_ins_cost(G2[u2][v2]).
        That is, the functions will receive the edge attribute
        dictionaries as inputs.  The functions are expected to return
        positive numeric values.
        Function edge_subst_cost overrides edge_match if specified.
        If neither edge_match nor edge_subst_cost are specified then
        default edge substitution cost of 0 is used (edge attributes
        are not considered during matching).
        If edge_del_cost is not specified then default edge deletion
        cost of 1 is used.  If edge_ins_cost is not specified then
        default edge insertion cost of 1 is used.
    roots : 2-tuple
        Tuple where first element is a node in G1 and the second
        is a node in G2.
        These nodes are forced to be matched in the comparison to
        allow comparison between rooted graphs.
    upper_bound : numeric
        Maximum edit distance to consider.  Return None if no edit
        distance under or equal to upper_bound exists.
    timeout : numeric
        Maximum number of seconds to execute.
        After timeout is met, the current best GED is returned.
```

```
(4)                 Examples
(4)                 --------
(4)                 >>> G1 = nx.cycle_graph(6)
(4)                 >>> G2 = nx.wheel_graph(7)
(4)                 >>> nx.graph_edit_distance(G1, G2)
(4)                 7.0
(4)                 >>> G1 = nx.star_graph(5)
(4)                 >>> G2 = nx.star_graph(5)
(4)                 >>> nx.graph_edit_distance(G1, G2, roots=(0, 0))
(4)                 0.0
(4)                 >>> nx.graph_edit_distance(G1, G2, roots=(1, 0))
(4)                 8.0
(4)                 See Also
(4)                 --------
(4)                 optimal_edit_paths, optimize_graph_edit_distance,
(4)                 is_isomorphic: test for graph edit distance of 0
(4)                 References
(4)                 ----------
(4)                 .. [1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick
(7)                    Martineau. An Exact Graph Edit Distance Algorithm for Solving
(7)                    Pattern Recognition Problems. 4th International Conference on
(7)                    Pattern Recognition Applications and Methods 2015, Jan 2015,
(7)                    Lisbon, Portugal. 2015,
(7)                    <10.5220/0005209202710278>. <hal-01168816>
(7)                    https://hal.archives-ouvertes.fr/hal-01168816
(4)                 """
(4)                 bestcost = None
(4)                 for _, _, cost in optimize_edit_paths(
(8)                     G1,
(8)                     G2,
(8)                     node_match,
(8)                     edge_match,
(8)                     node_subst_cost,
(8)                     node_del_cost,
(8)                     node_ins_cost,
(8)                     edge_subst_cost,
(8)                     edge_del_cost,
(8)                     edge_ins_cost,
(8)                     upper_bound,
(8)                     True,
(8)                     roots,
(8)                     timeout,
(4)                 ):
(8)                     bestcost = cost
(4)                 return bestcost
(0)             @nx._dispatchable(graphs={"G1": 0, "G2": 1})
(0)             def optimal_edit_paths(
(4)                 G1,
(4)                 G2,
(4)                 node_match=None,
(4)                 edge_match=None,
(4)                 node_subst_cost=None,
(4)                 node_del_cost=None,
(4)                 node_ins_cost=None,
(4)                 edge_subst_cost=None,
(4)                 edge_del_cost=None,
(4)                 edge_ins_cost=None,
(4)                 upper_bound=None,
(0)             ):
(4)                 """Returns all minimum-cost edit paths transforming G1 to G2.
(4)                 Graph edit path is a sequence of node and edge edit operations
(4)                 transforming graph G1 to graph isomorphic to G2.  Edit operations
(4)                 include substitutions, deletions, and insertions.
(4)                 Parameters
(4)                 ----------
(4)                 G1, G2: graphs
(8)                     The two graphs G1 and G2 must be of the same type.
(4)                 node_match : callable
(8)                     A function that returns True if node n1 in G1 and n2 in G2
```

```
(8)                 should be considered equal during matching.
(8)                 The function will be called like
(11)                   node_match(G1.nodes[n1], G2.nodes[n2]).
(8)                 That is, the function will receive the node attribute
(8)                 dictionaries for n1 and n2 as inputs.
(8)                 Ignored if node_subst_cost is specified.  If neither
(8)                 node_match nor node_subst_cost are specified then node
(8)                 attributes are not considered.
(4)             edge_match : callable
(8)                 A function that returns True if the edge attribute dictionaries
(8)                 for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should
(8)                 be considered equal during matching.
(8)                 The function will be called like
(11)                   edge_match(G1[u1][v1], G2[u2][v2]).
(8)                 That is, the function will receive the edge attribute
(8)                 dictionaries of the edges under consideration.
(8)                 Ignored if edge_subst_cost is specified.  If neither
(8)                 edge_match nor edge_subst_cost are specified then edge
(8)                 attributes are not considered.
(4)             node_subst_cost, node_del_cost, node_ins_cost : callable
(8)                 Functions that return the costs of node substitution, node
(8)                 deletion, and node insertion, respectively.
(8)                 The functions will be called like
(11)                   node_subst_cost(G1.nodes[n1], G2.nodes[n2]),
(11)                   node_del_cost(G1.nodes[n1]),
(11)                   node_ins_cost(G2.nodes[n2]).
(8)                 That is, the functions will receive the node attribute
(8)                 dictionaries as inputs.  The functions are expected to return
(8)                 positive numeric values.
(8)                 Function node_subst_cost overrides node_match if specified.
(8)                 If neither node_match nor node_subst_cost are specified then
(8)                 default node substitution cost of 0 is used (node attributes
(8)                 are not considered during matching).
(8)                 If node_del_cost is not specified then default node deletion
(8)                 cost of 1 is used.  If node_ins_cost is not specified then
(8)                 default node insertion cost of 1 is used.
(4)             edge_subst_cost, edge_del_cost, edge_ins_cost : callable
(8)                 Functions that return the costs of edge substitution, edge
(8)                 deletion, and edge insertion, respectively.
(8)                 The functions will be called like
(11)                   edge_subst_cost(G1[u1][v1], G2[u2][v2]),
(11)                   edge_del_cost(G1[u1][v1]),
(11)                   edge_ins_cost(G2[u2][v2]).
(8)                 That is, the functions will receive the edge attribute
(8)                 dictionaries as inputs.  The functions are expected to return
(8)                 positive numeric values.
(8)                 Function edge_subst_cost overrides edge_match if specified.
(8)                 If neither edge_match nor edge_subst_cost are specified then
(8)                 default edge substitution cost of 0 is used (edge attributes
(8)                 are not considered during matching).
(8)                 If edge_del_cost is not specified then default edge deletion
(8)                 cost of 1 is used.  If edge_ins_cost is not specified then
(8)                 default edge insertion cost of 1 is used.
(4)             upper_bound : numeric
(8)                 Maximum edit distance to consider.
(4)             Returns
(4)             -------
(4)             edit_paths : list of tuples (node_edit_path, edge_edit_path)
(8)                 node_edit_path : list of tuples (u, v)
(8)                 edge_edit_path : list of tuples ((u1, v1), (u2, v2))
(4)             cost : numeric
(8)                 Optimal edit path cost (graph edit distance). When the cost
(8)                 is zero, it indicates that `G1` and `G2` are isomorphic.
(4)             Examples
(4)             --------
(4)             >>> G1 = nx.cycle_graph(4)
(4)             >>> G2 = nx.wheel_graph(5)
(4)             >>> paths, cost = nx.optimal_edit_paths(G1, G2)
(4)             >>> len(paths)
```

```
(4)                 40
(4)                 >>> cost
(4)                 5.0
(4)                 Notes
(4)                 -----
(4)                 To transform `G1` into a graph isomorphic to `G2`, apply the node
(4)                 and edge edits in the returned ``edit_paths``.
(4)                 In the case of isomorphic graphs, the cost is zero, and the paths
(4)                 represent different isomorphic mappings (isomorphisms). That is, the
(4)                 edits involve renaming nodes and edges to match the structure of `G2`.
(4)                 See Also
(4)                 --------
(4)                 graph_edit_distance, optimize_edit_paths
(4)                 References
(4)                 ----------
(4)                 .. [1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick
(7)                    Martineau. An Exact Graph Edit Distance Algorithm for Solving
(7)                    Pattern Recognition Problems. 4th International Conference on
(7)                    Pattern Recognition Applications and Methods 2015, Jan 2015,
(7)                    Lisbon, Portugal. 2015,
(7)                    <10.5220/0005209202710278>. <hal-01168816>
(7)                    https://hal.archives-ouvertes.fr/hal-01168816
(4)                 """
(4)                 paths = []
(4)                 bestcost = None
(4)                 for vertex_path, edge_path, cost in optimize_edit_paths(
(8)                     G1,
(8)                     G2,
(8)                     node_match,
(8)                     edge_match,
(8)                     node_subst_cost,
(8)                     node_del_cost,
(8)                     node_ins_cost,
(8)                     edge_subst_cost,
(8)                     edge_del_cost,
(8)                     edge_ins_cost,
(8)                     upper_bound,
(8)                     False,
(4)                 ):
(8)                     if bestcost is not None and cost < bestcost:
(12)                        paths = []
(8)                     paths.append((vertex_path, edge_path))
(8)                     bestcost = cost
(4)                 return paths, bestcost
(0)             @nx._dispatchable(graphs={"G1": 0, "G2": 1})
(0)             def optimize_graph_edit_distance(
(4)                 G1,
(4)                 G2,
(4)                 node_match=None,
(4)                 edge_match=None,
(4)                 node_subst_cost=None,
(4)                 node_del_cost=None,
(4)                 node_ins_cost=None,
(4)                 edge_subst_cost=None,
(4)                 edge_del_cost=None,
(4)                 edge_ins_cost=None,
(4)                 upper_bound=None,
(0)             ):
(4)                 """Returns consecutive approximations of GED (graph edit distance)
(4)                 between graphs G1 and G2.
(4)                 Graph edit distance is a graph similarity measure analogous to
(4)                 Levenshtein distance for strings.  It is defined as minimum cost
(4)                 of edit path (sequence of node and edge edit operations)
(4)                 transforming graph G1 to graph isomorphic to G2.
(4)                 Parameters
(4)                 ----------
(4)                 G1, G2: graphs
(8)                     The two graphs G1 and G2 must be of the same type.
(4)                 node_match : callable
```

```
(8)                 A function that returns True if node n1 in G1 and n2 in G2
(8)                 should be considered equal during matching.
(8)                 The function will be called like
(11)                   node_match(G1.nodes[n1], G2.nodes[n2]).
(8)                 That is, the function will receive the node attribute
(8)                 dictionaries for n1 and n2 as inputs.
(8)                 Ignored if node_subst_cost is specified.  If neither
(8)                 node_match nor node_subst_cost are specified then node
(8)                 attributes are not considered.
(4)             edge_match : callable
(8)                 A function that returns True if the edge attribute dictionaries
(8)                 for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should
(8)                 be considered equal during matching.
(8)                 The function will be called like
(11)                   edge_match(G1[u1][v1], G2[u2][v2]).
(8)                 That is, the function will receive the edge attribute
(8)                 dictionaries of the edges under consideration.
(8)                 Ignored if edge_subst_cost is specified.  If neither
(8)                 edge_match nor edge_subst_cost are specified then edge
(8)                 attributes are not considered.
(4)             node_subst_cost, node_del_cost, node_ins_cost : callable
(8)                 Functions that return the costs of node substitution, node
(8)                 deletion, and node insertion, respectively.
(8)                 The functions will be called like
(11)                   node_subst_cost(G1.nodes[n1], G2.nodes[n2]),
(11)                   node_del_cost(G1.nodes[n1]),
(11)                   node_ins_cost(G2.nodes[n2]).
(8)                 That is, the functions will receive the node attribute
(8)                 dictionaries as inputs.  The functions are expected to return
(8)                 positive numeric values.
(8)                 Function node_subst_cost overrides node_match if specified.
(8)                 If neither node_match nor node_subst_cost are specified then
(8)                 default node substitution cost of 0 is used (node attributes
(8)                 are not considered during matching).
(8)                 If node_del_cost is not specified then default node deletion
(8)                 cost of 1 is used.  If node_ins_cost is not specified then
(8)                 default node insertion cost of 1 is used.
(4)             edge_subst_cost, edge_del_cost, edge_ins_cost : callable
(8)                 Functions that return the costs of edge substitution, edge
(8)                 deletion, and edge insertion, respectively.
(8)                 The functions will be called like
(11)                   edge_subst_cost(G1[u1][v1], G2[u2][v2]),
(11)                   edge_del_cost(G1[u1][v1]),
(11)                   edge_ins_cost(G2[u2][v2]).
(8)                 That is, the functions will receive the edge attribute
(8)                 dictionaries as inputs.  The functions are expected to return
(8)                 positive numeric values.
(8)                 Function edge_subst_cost overrides edge_match if specified.
(8)                 If neither edge_match nor edge_subst_cost are specified then
(8)                 default edge substitution cost of 0 is used (edge attributes
(8)                 are not considered during matching).
(8)                 If edge_del_cost is not specified then default edge deletion
(8)                 cost of 1 is used.  If edge_ins_cost is not specified then
(8)                 default edge insertion cost of 1 is used.
(4)             upper_bound : numeric
(8)                 Maximum edit distance to consider.
(4)             Returns
(4)             -------
(4)             Generator of consecutive approximations of graph edit distance.
(4)             Examples
(4)             --------
(4)             >>> G1 = nx.cycle_graph(6)
(4)             >>> G2 = nx.wheel_graph(7)
(4)             >>> for v in nx.optimize_graph_edit_distance(G1, G2):
(4)             ...     minv = v
(4)             >>> minv
(4)             7.0
(4)             See Also
(4)             --------
```

```
399: (4)                graph_edit_distance, optimize_edit_paths
400: (4)                References
401: (4)                ----------
402: (4)                .. [1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick
403: (7)                   Martineau. An Exact Graph Edit Distance Algorithm for Solving
404: (7)                   Pattern Recognition Problems. 4th International Conference on
405: (7)                   Pattern Recognition Applications and Methods 2015, Jan 2015,
406: (7)                   Lisbon, Portugal. 2015,
407: (7)                   <10.5220/0005209202710278>. <hal-01168816>
408: (7)                   https://hal.archives-ouvertes.fr/hal-01168816
409: (4)                """
410: (4)                for _, _, cost in optimize_edit_paths(
411: (8)                    G1,
412: (8)                    G2,
413: (8)                    node_match,
414: (8)                    edge_match,
415: (8)                    node_subst_cost,
416: (8)                    node_del_cost,
417: (8)                    node_ins_cost,
418: (8)                    edge_subst_cost,
419: (8)                    edge_del_cost,
420: (8)                    edge_ins_cost,
421: (8)                    upper_bound,
422: (8)                    True,
423: (4)                ):
424: (8)                    yield cost
425: (0)            @nx._dispatchable(
426: (4)                graphs={"G1": 0, "G2": 1}, preserve_edge_attrs=True,
preserve_node_attrs=True
427: (0)            )
428: (0)            def optimize_edit_paths(
429: (4)                G1,
430: (4)                G2,
431: (4)                node_match=None,
432: (4)                edge_match=None,
433: (4)                node_subst_cost=None,
434: (4)                node_del_cost=None,
435: (4)                node_ins_cost=None,
436: (4)                edge_subst_cost=None,
437: (4)                edge_del_cost=None,
438: (4)                edge_ins_cost=None,
439: (4)                upper_bound=None,
440: (4)                strictly_decreasing=True,
441: (4)                roots=None,
442: (4)                timeout=None,
443: (0)            ):
444: (4)                """GED (graph edit distance) calculation: advanced interface.
445: (4)                Graph edit path is a sequence of node and edge edit operations
446: (4)                transforming graph G1 to graph isomorphic to G2.  Edit operations
447: (4)                include substitutions, deletions, and insertions.
448: (4)                Graph edit distance is defined as minimum cost of edit path.
449: (4)                Parameters
450: (4)                ----------
451: (4)                G1, G2: graphs
452: (8)                    The two graphs G1 and G2 must be of the same type.
453: (4)                node_match : callable
454: (8)                    A function that returns True if node n1 in G1 and n2 in G2
455: (8)                    should be considered equal during matching.
456: (8)                    The function will be called like
457: (11)                      node_match(G1.nodes[n1], G2.nodes[n2]).
458: (8)                    That is, the function will receive the node attribute
459: (8)                    dictionaries for n1 and n2 as inputs.
460: (8)                    Ignored if node_subst_cost is specified.  If neither
461: (8)                    node_match nor node_subst_cost are specified then node
462: (8)                    attributes are not considered.
463: (4)                edge_match : callable
464: (8)                    A function that returns True if the edge attribute dictionaries
465: (8)                    for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should
466: (8)                    be considered equal during matching.
```

```
(8)                     The function will be called like
(11)                       edge_match(G1[u1][v1], G2[u2][v2]).
(8)                     That is, the function will receive the edge attribute
(8)                     dictionaries of the edges under consideration.
(8)                     Ignored if edge_subst_cost is specified.  If neither
(8)                     edge_match nor edge_subst_cost are specified then edge
(8)                     attributes are not considered.
(4)                 node_subst_cost, node_del_cost, node_ins_cost : callable
(8)                     Functions that return the costs of node substitution, node
(8)                     deletion, and node insertion, respectively.
(8)                     The functions will be called like
(11)                       node_subst_cost(G1.nodes[n1], G2.nodes[n2]),
(11)                       node_del_cost(G1.nodes[n1]),
(11)                       node_ins_cost(G2.nodes[n2]).
(8)                     That is, the functions will receive the node attribute
(8)                     dictionaries as inputs.  The functions are expected to return
(8)                     positive numeric values.
(8)                     Function node_subst_cost overrides node_match if specified.
(8)                     If neither node_match nor node_subst_cost are specified then
(8)                     default node substitution cost of 0 is used (node attributes
(8)                     are not considered during matching).
(8)                     If node_del_cost is not specified then default node deletion
(8)                     cost of 1 is used.  If node_ins_cost is not specified then
(8)                     default node insertion cost of 1 is used.
(4)                 edge_subst_cost, edge_del_cost, edge_ins_cost : callable
(8)                     Functions that return the costs of edge substitution, edge
(8)                     deletion, and edge insertion, respectively.
(8)                     The functions will be called like
(11)                       edge_subst_cost(G1[u1][v1], G2[u2][v2]),
(11)                       edge_del_cost(G1[u1][v1]),
(11)                       edge_ins_cost(G2[u2][v2]).
(8)                     That is, the functions will receive the edge attribute
(8)                     dictionaries as inputs.  The functions are expected to return
(8)                     positive numeric values.
(8)                     Function edge_subst_cost overrides edge_match if specified.
(8)                     If neither edge_match nor edge_subst_cost are specified then
(8)                     default edge substitution cost of 0 is used (edge attributes
(8)                     are not considered during matching).
(8)                     If edge_del_cost is not specified then default edge deletion
(8)                     cost of 1 is used.  If edge_ins_cost is not specified then
(8)                     default edge insertion cost of 1 is used.
(4)                 upper_bound : numeric
(8)                     Maximum edit distance to consider.
(4)                 strictly_decreasing : bool
(8)                     If True, return consecutive approximations of strictly
(8)                     decreasing cost.  Otherwise, return all edit paths of cost
(8)                     less than or equal to the previous minimum cost.
(4)                 roots : 2-tuple
(8)                     Tuple where first element is a node in G1 and the second
(8)                     is a node in G2.
(8)                     These nodes are forced to be matched in the comparison to
(8)                     allow comparison between rooted graphs.
(4)                 timeout : numeric
(8)                     Maximum number of seconds to execute.
(8)                     After timeout is met, the current best GED is returned.
(4)                 Returns
(4)                 -------
(4)                 Generator of tuples (node_edit_path, edge_edit_path, cost)
(8)                     node_edit_path : list of tuples (u, v)
(8)                     edge_edit_path : list of tuples ((u1, v1), (u2, v2))
(8)                     cost : numeric
(4)                 See Also
(4)                 --------
(4)                 graph_edit_distance, optimize_graph_edit_distance, optimal_edit_paths
(4)                 References
(4)                 ----------
(4)                 .. [1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick
(7)                    Martineau. An Exact Graph Edit Distance Algorithm for Solving
(7)                    Pattern Recognition Problems. 4th International Conference on
```

```
(7)                 Pattern Recognition Applications and Methods 2015, Jan 2015,
(7)                 Lisbon, Portugal. 2015,
(7)                 <10.5220/0005209202710278>. <hal-01168816>
(7)                 https://hal.archives-ouvertes.fr/hal-01168816
(4)             """
(4)             import numpy as np
(4)             import scipy as sp
(4)             @dataclass
(4)             class CostMatrix:
(8)                 C: ...
(8)                 lsa_row_ind: ...
(8)                 lsa_col_ind: ...
(8)                 ls: ...
(4)             def make_CostMatrix(C, m, n):
(8)                 lsa_row_ind, lsa_col_ind = sp.optimize.linear_sum_assignment(C)
(8)                 indexes = zip(range(len(lsa_row_ind)), lsa_row_ind, lsa_col_ind)
(8)                 subst_ind = [k for k, i, j in indexes if i < m and j < n]
(8)                 indexes = zip(range(len(lsa_row_ind)), lsa_row_ind, lsa_col_ind)
(8)                 dummy_ind = [k for k, i, j in indexes if i >= m and j >= n]
(8)                 lsa_row_ind[dummy_ind] = lsa_col_ind[subst_ind] + m
(8)                 lsa_col_ind[dummy_ind] = lsa_row_ind[subst_ind] + n
(8)                 return CostMatrix(
(12)                    C, lsa_row_ind, lsa_col_ind, C[lsa_row_ind, lsa_col_ind].sum()
(8)                 )
(4)             def extract_C(C, i, j, m, n):
(8)                 row_ind = [k in i or k - m in j for k in range(m + n)]
(8)                 col_ind = [k in j or k - n in i for k in range(m + n)]
(8)                 return C[row_ind, :][:, col_ind]
(4)             def reduce_C(C, i, j, m, n):
(8)                 row_ind = [k not in i and k - m not in j for k in range(m + n)]
(8)                 col_ind = [k not in j and k - n not in i for k in range(m + n)]
(8)                 return C[row_ind, :][:, col_ind]
(4)             def reduce_ind(ind, i):
(8)                 rind = ind[[k not in i for k in ind]]
(8)                 for k in set(i):
(12)                    rind[rind >= k] -= 1
(8)                 return rind
(4)             def match_edges(u, v, pending_g, pending_h, Ce, matched_uv=None):
(8)                 """
(8)                 Parameters:
(12)                    u, v: matched vertices, u=None or v=None for
(15)                       deletion/insertion
(12)                    pending_g, pending_h: lists of edges not yet mapped
(12)                    Ce: CostMatrix of pending edge mappings
(12)                    matched_uv: partial vertex edit path
(16)                        list of tuples (u, v) of previously matched vertex
(20)                            mappings u<->v, u=None or v=None for
(20)                            deletion/insertion
(8)                 Returns:
(12)                    list of (i, j): indices of edge mappings g<->h
(12)                    localCe: local CostMatrix of edge mappings
(16)                        (basically submatrix of Ce at cross of rows i, cols j)
(8)                 """
(8)                 M = len(pending_g)
(8)                 N = len(pending_h)
(8)                 if matched_uv is None or len(matched_uv) == 0:
(12)                    g_ind = []
(12)                    h_ind = []
(8)                 else:
(12)                    g_ind = [
(16)                        i
(16)                        for i in range(M)
(16)                        if pending_g[i][:2] == (u, u)
(16)                        or any(
(20)                            pending_g[i][:2] in ((p, u), (u, p), (p, p)) for p, q in
matched_uv
(16)                        )
(12)                    ]
(12)                    h_ind = [
```

```
604: (16)                            j
605: (16)                            for j in range(N)
606: (16)                            if pending_h[j][:2] == (v, v)
607: (16)                            or any(
608: (20)                                pending_h[j][:2] in ((q, v), (v, q), (q, q)) for p, q in
matched_uv
609: (16)                            )
610: (12)                        ]
611: (8)                    m = len(g_ind)
612: (8)                    n = len(h_ind)
613: (8)                    if m or n:
614: (12)                        C = extract_C(Ce.C, g_ind, h_ind, M, N)
615: (12)                        for k, i in enumerate(g_ind):
616: (16)                            g = pending_g[i][:2]
617: (16)                            for l, j in enumerate(h_ind):
618: (20)                                h = pending_h[j][:2]
619: (20)                                if nx.is_directed(G1) or nx.is_directed(G2):
620: (24)                                    if any(
621: (28)                                        g == (p, u) and h == (q, v) or g == (u, p) and h
== (v, q)
622: (28)                                        for p, q in matched_uv
623: (24)                                    ):
624: (28)                                        continue
625: (20)                                else:
626: (24)                                    if any(
627: (28)                                        g in ((p, u), (u, p)) and h in ((q, v), (v, q))
628: (28)                                        for p, q in matched_uv
629: (24)                                    ):
630: (28)                                        continue
631: (20)                                if g == (u, u) or any(g == (p, p) for p, q in matched_uv):
632: (24)                                    continue
633: (20)                                if h == (v, v) or any(h == (q, q) for p, q in matched_uv):
634: (24)                                    continue
635: (20)                                C[k, l] = inf
636: (12)                        localCe = make_CostMatrix(C, m, n)
637: (12)                        ij = [
638: (16)                            (
639: (20)                                g_ind[k] if k < m else M + h_ind[l],
640: (20)                                h_ind[l] if l < n else N + g_ind[k],
641: (16)                            )
642: (16)                            for k, l in zip(localCe.lsa_row_ind, localCe.lsa_col_ind)
643: (16)                            if k < m or l < n
644: (12)                        ]
645: (8)                    else:
646: (12)                        ij = []
647: (12)                        localCe = CostMatrix(np.empty((0, 0)), [], [], 0)
648: (8)                    return ij, localCe
649: (4)                def reduce_Ce(Ce, ij, m, n):
650: (8)                    if len(ij):
651: (12)                        i, j = zip(*ij)
652: (12)                        m_i = m - sum(1 for t in i if t < m)
653: (12)                        n_j = n - sum(1 for t in j if t < n)
654: (12)                        return make_CostMatrix(reduce_C(Ce.C, i, j, m, n), m_i, n_j)
655: (8)                    return Ce
656: (4)                def get_edit_ops(
657: (8)                    matched_uv, pending_u, pending_v, Cv, pending_g, pending_h, Ce,
matched_cost
658: (4)                ):
659: (8)                    """
660: (8)                    Parameters:
661: (12)                        matched_uv: partial vertex edit path
662: (16)                            list of tuples (u, v) of vertex mappings u<->v,
663: (16)                            u=None or v=None for deletion/insertion
664: (12)                        pending_u, pending_v: lists of vertices not yet mapped
665: (12)                        Cv: CostMatrix of pending vertex mappings
666: (12)                        pending_g, pending_h: lists of edges not yet mapped
667: (12)                        Ce: CostMatrix of pending edge mappings
668: (12)                        matched_cost: cost of partial edit path
669: (8)                    Returns:
```

```
(12)                        sequence of
(16)                            (i, j): indices of vertex mapping u<->v
(16)                            Cv_ij: reduced CostMatrix of pending vertex mappings
(20)                                (basically Cv with row i, col j removed)
(16)                            list of (x, y): indices of edge mappings g<->h
(16)                            Ce_xy: reduced CostMatrix of pending edge mappings
(20)                                (basically Ce with rows x, cols y removed)
(16)                            cost: total cost of edit operation
(12)                        NOTE: most promising ops first
(8)                     """
(8)                     m = len(pending_u)
(8)                     n = len(pending_v)
(8)                     i, j = min(
(12)                        (k, l) for k, l in zip(Cv.lsa_row_ind, Cv.lsa_col_ind) if k < m or
l < n
(8)                     )
(8)                     xy, localCe = match_edges(
(12)                        pending_u[i] if i < m else None,
(12)                        pending_v[j] if j < n else None,
(12)                        pending_g,
(12)                        pending_h,
(12)                        Ce,
(12)                        matched_uv,
(8)                     )
(8)                     Ce_xy = reduce_Ce(Ce, xy, len(pending_g), len(pending_h))
(8)                     if prune(matched_cost + Cv.ls + localCe.ls + Ce_xy.ls):
(12)                        pass
(8)                     else:
(12)                        Cv_ij = CostMatrix(
(16)                            reduce_C(Cv.C, (i,), (j,), m, n),
(16)                            reduce_ind(Cv.lsa_row_ind, (i, m + j)),
(16)                            reduce_ind(Cv.lsa_col_ind, (j, n + i)),
(16)                            Cv.ls - Cv.C[i, j],
(12)                        )
(12)                        yield (i, j), Cv_ij, xy, Ce_xy, Cv.C[i, j] + localCe.ls
(8)                     other = []
(8)                     fixed_i, fixed_j = i, j
(8)                     if m <= n:
(12)                        candidates = (
(16)                            (t, fixed_j)
(16)                            for t in range(m + n)
(16)                            if t != fixed_i and (t < m or t == m + fixed_j)
(12)                        )
(8)                     else:
(12)                        candidates = (
(16)                            (fixed_i, t)
(16)                            for t in range(m + n)
(16)                            if t != fixed_j and (t < n or t == n + fixed_i)
(12)                        )
(8)                     for i, j in candidates:
(12)                        if prune(matched_cost + Cv.C[i, j] + Ce.ls):
(16)                            continue
(12)                        Cv_ij = make_CostMatrix(
(16)                            reduce_C(Cv.C, (i,), (j,), m, n),
(16)                            m - 1 if i < m else m,
(16)                            n - 1 if j < n else n,
(12)                        )
(12)                        if prune(matched_cost + Cv.C[i, j] + Cv_ij.ls + Ce.ls):
(16)                            continue
(12)                        xy, localCe = match_edges(
(16)                            pending_u[i] if i < m else None,
(16)                            pending_v[j] if j < n else None,
(16)                            pending_g,
(16)                            pending_h,
(16)                            Ce,
(16)                            matched_uv,
(12)                        )
(12)                        if prune(matched_cost + Cv.C[i, j] + Cv_ij.ls + localCe.ls):
(16)                            continue
```

```
(12)                        Ce_xy = reduce_Ce(Ce, xy, len(pending_g), len(pending_h))
(12)                        if prune(matched_cost + Cv.C[i, j] + Cv_ij.ls + localCe.ls +
Ce_xy.ls):
(16)                            continue
(12)                        other.append(((i, j), Cv_ij, xy, Ce_xy, Cv.C[i, j] + localCe.ls))
(8)                     yield from sorted(other, key=lambda t: t[4] + t[1].ls + t[3].ls)
(4)                 def get_edit_paths(
(8)                     matched_uv,
(8)                     pending_u,
(8)                     pending_v,
(8)                     Cv,
(8)                     matched_gh,
(8)                     pending_g,
(8)                     pending_h,
(8)                     Ce,
(8)                     matched_cost,
(4)                 ):
(8)                     """
(8)                     Parameters:
(12)                        matched_uv: partial vertex edit path
(16)                            list of tuples (u, v) of vertex mappings u<->v,
(16)                            u=None or v=None for deletion/insertion
(12)                        pending_u, pending_v: lists of vertices not yet mapped
(12)                        Cv: CostMatrix of pending vertex mappings
(12)                        matched_gh: partial edge edit path
(16)                            list of tuples (g, h) of edge mappings g<->h,
(16)                            g=None or h=None for deletion/insertion
(12)                        pending_g, pending_h: lists of edges not yet mapped
(12)                        Ce: CostMatrix of pending edge mappings
(12)                        matched_cost: cost of partial edit path
(8)                     Returns:
(12)                        sequence of (vertex_path, edge_path, cost)
(16)                            vertex_path: complete vertex edit path
(20)                                list of tuples (u, v) of vertex mappings u<->v,
(20)                                u=None or v=None for deletion/insertion
(16)                            edge_path: complete edge edit path
(20)                                list of tuples (g, h) of edge mappings g<->h,
(20)                                g=None or h=None for deletion/insertion
(16)                            cost: total cost of edit path
(12)                        NOTE: path costs are non-increasing
(8)                     """
(8)                     if prune(matched_cost + Cv.ls + Ce.ls):
(12)                        return
(8)                     if not max(len(pending_u), len(pending_v)):
(12)                        nonlocal maxcost_value
(12)                        maxcost_value = min(maxcost_value, matched_cost)
(12)                        yield matched_uv, matched_gh, matched_cost
(8)                     else:
(12)                        edit_ops = get_edit_ops(
(16)                            matched_uv,
(16)                            pending_u,
(16)                            pending_v,
(16)                            Cv,
(16)                            pending_g,
(16)                            pending_h,
(16)                            Ce,
(16)                            matched_cost,
(12)                        )
(12)                        for ij, Cv_ij, xy, Ce_xy, edit_cost in edit_ops:
(16)                            i, j = ij
(16)                            if prune(matched_cost + edit_cost + Cv_ij.ls + Ce_xy.ls):
(20)                                continue
(16)                            u = pending_u.pop(i) if i < len(pending_u) else None
(16)                            v = pending_v.pop(j) if j < len(pending_v) else None
(16)                            matched_uv.append((u, v))
(16)                            for x, y in xy:
(20)                                len_g = len(pending_g)
(20)                                len_h = len(pending_h)
(20)                                matched_gh.append(
```

```
(24)                                    (
(28)                                        pending_g[x] if x < len_g else None,
(28)                                        pending_h[y] if y < len_h else None,
(24)                                    )
(20)                                )
(16)                            sortedx = sorted(x for x, y in xy)
(16)                            sortedy = sorted(y for x, y in xy)
(16)                            G = [
(20)                                (pending_g.pop(x) if x < len(pending_g) else None)
(20)                                for x in reversed(sortedx)
(16)                            ]
(16)                            H = [
(20)                                (pending_h.pop(y) if y < len(pending_h) else None)
(20)                                for y in reversed(sortedy)
(16)                            ]
(16)                            yield from get_edit_paths(
(20)                                matched_uv,
(20)                                pending_u,
(20)                                pending_v,
(20)                                Cv_ij,
(20)                                matched_gh,
(20)                                pending_g,
(20)                                pending_h,
(20)                                Ce_xy,
(20)                                matched_cost + edit_cost,
(16)                            )
(16)                            if u is not None:
(20)                                pending_u.insert(i, u)
(16)                            if v is not None:
(20)                                pending_v.insert(j, v)
(16)                            matched_uv.pop()
(16)                            for x, g in zip(sortedx, reversed(G)):
(20)                                if g is not None:
(24)                                    pending_g.insert(x, g)
(16)                            for y, h in zip(sortedy, reversed(H)):
(20)                                if h is not None:
(24)                                    pending_h.insert(y, h)
(16)                            for _ in xy:
(20)                                matched_gh.pop()
(4)                 pending_u = list(G1.nodes)
(4)                 pending_v = list(G2.nodes)
(4)                 initial_cost = 0
(4)                 if roots:
(8)                     root_u, root_v = roots
(8)                     if root_u not in pending_u or root_v not in pending_v:
(12)                        raise nx.NodeNotFound("Root node not in graph.")
(8)                     pending_u.remove(root_u)
(8)                     pending_v.remove(root_v)
(4)                 m = len(pending_u)
(4)                 n = len(pending_v)
(4)                 C = np.zeros((m + n, m + n))
(4)                 if node_subst_cost:
(8)                     C[0:m, 0:n] = np.array(
(12)                        [
(16)                            node_subst_cost(G1.nodes[u], G2.nodes[v])
(16)                            for u in pending_u
(16)                            for v in pending_v
(12)                        ]
(8)                     ).reshape(m, n)
(8)                     if roots:
(12)                        initial_cost = node_subst_cost(G1.nodes[root_u], G2.nodes[root_v])
(4)                 elif node_match:
(8)                     C[0:m, 0:n] = np.array(
(12)                        [
(16)                            1 - int(node_match(G1.nodes[u], G2.nodes[v]))
(16)                            for u in pending_u
(16)                            for v in pending_v
(12)                        ]
(8)                     ).reshape(m, n)
```

```
(8)                     if roots:
(12)                        initial_cost = 1 - node_match(G1.nodes[root_u], G2.nodes[root_v])
(4)                 else:
(8)                     pass
(4)                 if node_del_cost:
(8)                     del_costs = [node_del_cost(G1.nodes[u]) for u in pending_u]
(4)                 else:
(8)                     del_costs = [1] * len(pending_u)
(4)                 if node_ins_cost:
(8)                     ins_costs = [node_ins_cost(G2.nodes[v]) for v in pending_v]
(4)                 else:
(8)                     ins_costs = [1] * len(pending_v)
(4)                 inf = C[0:m, 0:n].sum() + sum(del_costs) + sum(ins_costs) + 1
(4)                 C[0:m, n : n + m] = np.array(
(8)                     [del_costs[i] if i == j else inf for i in range(m) for j in range(m)]
(4)                 ).reshape(m, m)
(4)                 C[m : m + n, 0:n] = np.array(
(8)                     [ins_costs[i] if i == j else inf for i in range(n) for j in range(n)]
(4)                 ).reshape(n, n)
(4)                 Cv = make_CostMatrix(C, m, n)
(4)                 pending_g = list(G1.edges)
(4)                 pending_h = list(G2.edges)
(4)                 m = len(pending_g)
(4)                 n = len(pending_h)
(4)                 C = np.zeros((m + n, m + n))
(4)                 if edge_subst_cost:
(8)                     C[0:m, 0:n] = np.array(
(12)                        [
(16)                            edge_subst_cost(G1.edges[g], G2.edges[h])
(16)                            for g in pending_g
(16)                            for h in pending_h
(12)                        ]
(8)                     ).reshape(m, n)
(4)                 elif edge_match:
(8)                     C[0:m, 0:n] = np.array(
(12)                        [
(16)                            1 - int(edge_match(G1.edges[g], G2.edges[h]))
(16)                            for g in pending_g
(16)                            for h in pending_h
(12)                        ]
(8)                     ).reshape(m, n)
(4)                 else:
(8)                     pass
(4)                 if edge_del_cost:
(8)                     del_costs = [edge_del_cost(G1.edges[g]) for g in pending_g]
(4)                 else:
(8)                     del_costs = [1] * len(pending_g)
(4)                 if edge_ins_cost:
(8)                     ins_costs = [edge_ins_cost(G2.edges[h]) for h in pending_h]
(4)                 else:
(8)                     ins_costs = [1] * len(pending_h)
(4)                 inf = C[0:m, 0:n].sum() + sum(del_costs) + sum(ins_costs) + 1
(4)                 C[0:m, n : n + m] = np.array(
(8)                     [del_costs[i] if i == j else inf for i in range(m) for j in range(m)]
(4)                 ).reshape(m, m)
(4)                 C[m : m + n, 0:n] = np.array(
(8)                     [ins_costs[i] if i == j else inf for i in range(n) for j in range(n)]
(4)                 ).reshape(n, n)
(4)                 Ce = make_CostMatrix(C, m, n)
(4)                 maxcost_value = Cv.C.sum() + Ce.C.sum() + 1
(4)                 if timeout is not None:
(8)                     if timeout <= 0:
(12)                        raise nx.NetworkXError("Timeout value must be greater than 0")
(8)                     start = time.perf_counter()
(4)                 def prune(cost):
(8)                     if timeout is not None:
(12)                        if time.perf_counter() - start > timeout:
(16)                            return True
(8)                     if upper_bound is not None:
```

```
944: (12)                        if cost > upper_bound:
945: (16)                            return True
946: (8)                    if cost > maxcost_value:
947: (12)                        return True
948: (8)                    if strictly_decreasing and cost >= maxcost_value:
949: (12)                        return True
950: (8)                    return False
951: (4)                done_uv = [] if roots is None else [roots]
952: (4)                for vertex_path, edge_path, cost in get_edit_paths(
953: (8)                    done_uv, pending_u, pending_v, Cv, [], pending_g, pending_h, Ce,
initial_cost
954: (4)                ):
955: (8)                    yield list(vertex_path), list(edge_path), float(cost)
956: (0)            @nx._dispatchable
957: (0)            def simrank_similarity(
958: (4)                G,
959: (4)                source=None,
960: (4)                target=None,
961: (4)                importance_factor=0.9,
962: (4)                max_iterations=1000,
963: (4)                tolerance=1e-4,
964: (0)            ):
965: (4)                """Returns the SimRank similarity of nodes in the graph ``G``.
966: (4)                SimRank is a similarity metric that says "two objects are considered
967: (4)                to be similar if they are referenced by similar objects." [1]_.
968: (4)                The pseudo-code definition from the paper is::
969: (8)                    def simrank(G, u, v):
970: (12)                       in_neighbors_u = G.predecessors(u)
971: (12)                       in_neighbors_v = G.predecessors(v)
972: (12)                       scale = C / (len(in_neighbors_u) * len(in_neighbors_v))
973: (12)                       return scale * sum(
974: (16)                           simrank(G, w, x) for w, x in product(in_neighbors_u,
in_neighbors_v)
975: (12)                       )
976: (4)                where ``G`` is the graph, ``u`` is the source, ``v`` is the target,
977: (4)                and ``C`` is a float decay or importance factor between 0 and 1.
978: (4)                The SimRank algorithm for determining node similarity is defined in
979: (4)                [2]_.
980: (4)                Parameters
981: (4)                ----------
982: (4)                G : NetworkX graph
983: (8)                    A NetworkX graph
984: (4)                source : node
985: (8)                    If this is specified, the returned dictionary maps each node
986: (8)                    ``v`` in the graph to the similarity between ``source`` and
987: (8)                    ``v``.
988: (4)                target : node
989: (8)                    If both ``source`` and ``target`` are specified, the similarity
990: (8)                    value between ``source`` and ``target`` is returned. If
991: (8)                    ``target`` is specified but ``source`` is not, this argument is
992: (8)                    ignored.
993: (4)                importance_factor : float
994: (8)                    The relative importance of indirect neighbors with respect to
995: (8)                    direct neighbors.
996: (4)                max_iterations : integer
997: (8)                    Maximum number of iterations.
998: (4)                tolerance : float
999: (8)                    Error tolerance used to check convergence. When an iteration of
1000: (8)                   the algorithm finds that no similarity value changes more than
1001: (8)                   this amount, the algorithm halts.
1002: (4)               Returns
1003: (4)               -------
1004: (4)               similarity : dictionary or float
1005: (8)                   If ``source`` and ``target`` are both ``None``, this returns a
1006: (8)                   dictionary of dictionaries, where keys are node pairs and value
1007: (8)                   are similarity of the pair of nodes.
1008: (8)                   If ``source`` is not ``None`` but ``target`` is, this returns a
1009: (8)                   dictionary mapping node to the similarity of ``source`` and that
1010: (8)                   node.
```

```
(8)                 If neither ``source`` nor ``target`` is ``None``, this returns
(8)                 the similarity value for the given pair of nodes.
(4)             Raises
(4)             ------
(4)             ExceededMaxIterations
(8)                 If the algorithm does not converge within ``max_iterations``.
(4)             NodeNotFound
(8)                 If either ``source`` or ``target`` is not in `G`.
(4)             Examples
(4)             --------
(4)             >>> G = nx.cycle_graph(2)
(4)             >>> nx.simrank_similarity(G)
(4)             {0: {0: 1.0, 1: 0.0}, 1: {0: 0.0, 1: 1.0}}
(4)             >>> nx.simrank_similarity(G, source=0)
(4)             {0: 1.0, 1: 0.0}
(4)             >>> nx.simrank_similarity(G, source=0, target=0)
(4)             1.0
(4)             The result of this function can be converted to a numpy array
(4)             representing the SimRank matrix by using the node order of the
(4)             graph to determine which row and column represent each node.
(4)             Other ordering of nodes is also possible.
(4)             >>> import numpy as np
(4)             >>> sim = nx.simrank_similarity(G)
(4)             >>> np.array([[sim[u][v] for v in G] for u in G])
(4)             array([[1., 0.],
(11)                   [0., 1.]])
(4)             >>> sim_1d = nx.simrank_similarity(G, source=0)
(4)             >>> np.array([sim[0][v] for v in G])
(4)             array([1., 0.])
(4)             References
(4)             ----------
(4)             .. [1] https://en.wikipedia.org/wiki/SimRank
(4)             .. [2] G. Jeh and J. Widom.
(11)                   "SimRank: a measure of structural-context similarity",
(11)                   In KDD'02: Proceedings of the Eighth ACM SIGKDD
(11)                   International Conference on Knowledge Discovery and Data Mining,
(11)                   pp. 538--543. ACM Press, 2002.
(4)             """
(4)             import numpy as np
(4)             nodelist = list(G)
(4)             if source is not None:
(8)                 if source not in nodelist:
(12)                    raise nx.NodeNotFound(f"Source node {source} not in G")
(8)                 else:
(12)                    s_indx = nodelist.index(source)
(4)             else:
(8)                 s_indx = None
(4)             if target is not None:
(8)                 if target not in nodelist:
(12)                    raise nx.NodeNotFound(f"Target node {target} not in G")
(8)                 else:
(12)                    t_indx = nodelist.index(target)
(4)             else:
(8)                 t_indx = None
(4)             x = _simrank_similarity_numpy(
(8)                 G, s_indx, t_indx, importance_factor, max_iterations, tolerance
(4)             )
(4)             if isinstance(x, np.ndarray):
(8)                 if x.ndim == 1:
(12)                    return dict(zip(G, x.tolist()))
(8)                 return {u: dict(zip(G, row)) for u, row in zip(G, x.tolist())}
(4)             return float(x)
(0)         def _simrank_similarity_python(
(4)             G,
(4)             source=None,
(4)             target=None,
(4)             importance_factor=0.9,
(4)             max_iterations=1000,
(4)             tolerance=1e-4,
```

```
(0)         ):
(4)             """Returns the SimRank similarity of nodes in the graph ``G``.
(4)             This pure Python version is provided for pedagogical purposes.
(4)             Examples
(4)             --------
(4)             >>> G = nx.cycle_graph(2)
(4)             >>> nx.similarity._simrank_similarity_python(G)
(4)             {0: {0: 1, 1: 0.0}, 1: {0: 0.0, 1: 1}}
(4)             >>> nx.similarity._simrank_similarity_python(G, source=0)
(4)             {0: 1, 1: 0.0}
(4)             >>> nx.similarity._simrank_similarity_python(G, source=0, target=0)
(4)             1
(4)             """
(4)             newsim = {u: {v: 1 if u == v else 0 for v in G} for u in G}
(4)             def avg_sim(s):
(8)                 return sum(newsim[w][x] for (w, x) in s) / len(s) if s else 0.0
(4)             Gadj = G.pred if G.is_directed() else G.adj
(4)             def sim(u, v):
(8)                 return importance_factor * avg_sim(list(product(Gadj[u], Gadj[v])))
(4)             for its in range(max_iterations):
(8)                 oldsim = newsim
(8)                 newsim = {u: {v: sim(u, v) if u != v else 1 for v in G} for u in G}
(8)                 is_close = all(
(12)                    all(
(16)                        abs(newsim[u][v] - old) <= tolerance * (1 + abs(old))
(16)                        for v, old in nbrs.items()
(12)                    )
(12)                    for u, nbrs in oldsim.items()
(8)                 )
(8)                 if is_close:
(12)                    break
(4)             if its + 1 == max_iterations:
(8)                 raise nx.ExceededMaxIterations(
(12)                    f"simrank did not converge after {max_iterations} iterations."
(8)                 )
(4)             if source is not None and target is not None:
(8)                 return newsim[source][target]
(4)             if source is not None:
(8)                 return newsim[source]
(4)             return newsim
(0)         def _simrank_similarity_numpy(
(4)             G,
(4)             source=None,
(4)             target=None,
(4)             importance_factor=0.9,
(4)             max_iterations=1000,
(4)             tolerance=1e-4,
(0)         ):
(4)             """Calculate SimRank of nodes in ``G`` using matrices with ``numpy``.
(4)             The SimRank algorithm for determining node similarity is defined in
(4)             [1]_.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(8)                 A NetworkX graph
(4)             source : node
(8)                 If this is specified, the returned dictionary maps each node
(8)                 ``v`` in the graph to the similarity between ``source`` and
(8)                 ``v``.
(4)             target : node
(8)                 If both ``source`` and ``target`` are specified, the similarity
(8)                 value between ``source`` and ``target`` is returned. If
(8)                 ``target`` is specified but ``source`` is not, this argument is
(8)                 ignored.
(4)             importance_factor : float
(8)                 The relative importance of indirect neighbors with respect to
(8)                 direct neighbors.
(4)             max_iterations : integer
(8)                 Maximum number of iterations.
```

```
    tolerance : float
        Error tolerance used to check convergence. When an iteration of
        the algorithm finds that no similarity value changes more than
        this amount, the algorithm halts.
    Returns
    -------
    similarity : numpy array or float
        If ``source`` and ``target`` are both ``None``, this returns a
        2D array containing SimRank scores of the nodes.
        If ``source`` is not ``None`` but ``target`` is, this returns an
        1D array containing SimRank scores of ``source`` and that
        node.
        If neither ``source`` nor ``target`` is ``None``, this returns
        the similarity value for the given pair of nodes.
    Examples
    --------
    >>> G = nx.cycle_graph(2)
    >>> nx.similarity._simrank_similarity_numpy(G)
    array([[1., 0.],
           [0., 1.]])
    >>> nx.similarity._simrank_similarity_numpy(G, source=0)
    array([1., 0.])
    >>> nx.similarity._simrank_similarity_numpy(G, source=0, target=0)
    1.0
    References
    ----------
    .. [1] G. Jeh and J. Widom.
           "SimRank: a measure of structural-context similarity",
           In KDD'02: Proceedings of the Eighth ACM SIGKDD
           International Conference on Knowledge Discovery and Data Mining,
           pp. 538--543. ACM Press, 2002.
    """
    import numpy as np
    adjacency_matrix = nx.to_numpy_array(G)
    s = np.array(adjacency_matrix.sum(axis=0))
    s[s == 0] = 1
    adjacency_matrix /= s  # adjacency_matrix.sum(axis=0)
    newsim = np.eye(len(G), dtype=np.float64)
    for its in range(max_iterations):
        prevsim = newsim.copy()
        newsim = importance_factor * ((adjacency_matrix.T @ prevsim) @ adjacency_matrix)
        np.fill_diagonal(newsim, 1.0)
        if np.allclose(prevsim, newsim, atol=tolerance):
            break
    if its + 1 == max_iterations:
        raise nx.ExceededMaxIterations(
            f"simrank did not converge after {max_iterations} iterations."
        )
    if source is not None and target is not None:
        return float(newsim[source, target])
    if source is not None:
        return newsim[source]
    return newsim
@nx._dispatchable(edge_attrs="weight")
def panther_similarity(
    G, source, k=5, path_length=5, c=0.5, delta=0.1, eps=None, weight="weight"
):
    r"""Returns the Panther similarity of nodes in the graph `G` to node ``v``.
    Panther is a similarity metric that says "two objects are considered
    to be similar if they frequently appear on the same paths." [1]_.
    Parameters
    ----------
    G : NetworkX graph
        A NetworkX graph
    source : node
        Source node for which to find the top `k` similar other nodes
    k : int (default = 5)
```

```
        The number of most similar nodes to return.
    path_length : int (default = 5)
        How long the randomly generated paths should be (``T`` in [1]_)
    c : float (default = 0.5)
        A universal positive constant used to scale the number
        of sample random paths to generate.
    delta : float (default = 0.1)
        The probability that the similarity $S$ is not an epsilon-approximation to (R, phi),
        where $R$ is the number of random paths and $\phi$ is the probability
        that an element sampled from a set $A \subseteq D$, where $D$ is the domain.
    eps : float or None (default = None)
        The error bound. Per [1]_, a good value is ``sqrt(1/|E|)``. Therefore,
        if no value is provided, the recommended computed value will be used.
    weight : string or None, optional (default="weight")
        The name of an edge attribute that holds the numerical value
        used as a weight. If None then each edge has weight 1.
    Returns
    -------
    similarity : dictionary
        Dictionary of nodes to similarity scores (as floats). Note:
        the self-similarity (i.e., ``v``) will not be included in
        the returned dictionary. So, for ``k = 5``, a dictionary of
        top 4 nodes and their similarity scores will be returned.
    Raises
    ------
    NetworkXUnfeasible
        If `source` is an isolated node.
    NodeNotFound
        If `source` is not in `G`.
    Notes
    -----
        The isolated nodes in `G` are ignored.
    Examples
    --------
    >>> G = nx.star_graph(10)
    >>> sim = nx.panther_similarity(G, 0)
    References
    ----------
    .. [1] Zhang, J., Tang, J., Ma, C., Tong, H., Jing, Y., & Li, J.
           Panther: Fast top-k similarity search on large networks.
           In Proceedings of the ACM SIGKDD International Conference
           on Knowledge Discovery and Data Mining (Vol. 2015-August, pp. 1445–1454).
           Association for Computing Machinery. https://doi.org/10.1145/2783258.2783267.
    """
    import numpy as np
    if source not in G:
        raise nx.NodeNotFound(f"Source node {source} not in G")
    isolates = set(nx.isolates(G))
    if source in isolates:
        raise nx.NetworkXUnfeasible(
            f"Panther similarity is not defined for the isolated source node {source}."
        )
    G = G.subgraph([node for node in G.nodes if node not in isolates]).copy()
    num_nodes = G.number_of_nodes()
    if num_nodes < k:
        warnings.warn(
            f"Number of nodes is {num_nodes}, but requested k is {k}. "
            "Setting k to number of nodes."
        )
        k = num_nodes
    if eps is None:
        eps = np.sqrt(1.0 / G.number_of_edges())
    inv_node_map = {name: index for index, name in enumerate(G.nodes)}
    node_map = np.array(G)
```

```
(4)             t_choose_2 = math.comb(path_length, 2)
(4)             sample_size = int((c / eps**2) * (np.log2(t_choose_2) + 1 + np.log(1 / delta)))
(4)             index_map = {}
(4)             _ = list(
(8)                 generate_random_paths(
(12)                    G, sample_size, path_length=path_length, index_map=index_map, weight=weight
(8)                 )
(4)             )
(4)             S = np.zeros(num_nodes)
(4)             inv_sample_size = 1 / sample_size
(4)             source_paths = set(index_map[source])
(4)             for node, paths in index_map.items():
(8)                 common_paths = source_paths.intersection(paths)
(8)                 S[inv_node_map[node]] = len(common_paths) * inv_sample_size
(4)             top_k_unsorted = np.argpartition(S, -k)[-k:]
(4)             top_k_sorted = top_k_unsorted[np.argsort(S[top_k_unsorted])][::-1]
(4)             top_k_with_val = dict(
(8)                 zip(node_map[top_k_sorted].tolist(), S[top_k_sorted].tolist())
(4)             )
(4)             top_k_with_val.pop(source, None)
(4)             return top_k_with_val
(0)         @np_random_state(5)
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def generate_random_paths(
(4)             G, sample_size, path_length=5, index_map=None, weight="weight", seed=None
(0)         ):
(4)             """Randomly generate `sample_size` paths of length `path_length`.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(8)                 A NetworkX graph
(4)             sample_size : integer
(8)                 The number of paths to generate. This is ``R`` in [1]_.
(4)             path_length : integer (default = 5)
(8)                 The maximum size of the path to randomly generate.
(8)                 This is ``T`` in [1]_. According to the paper, ``T >= 5`` is
(8)                 recommended.
(4)             index_map : dictionary, optional
(8)                 If provided, this will be populated with the inverted
(8)                 index of nodes mapped to the set of generated random path
(8)                 indices within ``paths``.
(4)             weight : string or None, optional (default="weight")
(8)                 The name of an edge attribute that holds the numerical value
(8)                 used as a weight. If None then each edge has weight 1.
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             paths : generator of lists
(8)                 Generator of `sample_size` paths each with length `path_length`.
(4)             Examples
(4)             --------
(4)             Note that the return value is the list of paths:
(4)             >>> G = nx.star_graph(3)
(4)             >>> random_path = nx.generate_random_paths(G, 2)
(4)             By passing a dictionary into `index_map`, it will build an
(4)             inverted index mapping of nodes to the paths in which that node is present:
(4)             >>> G = nx.star_graph(3)
(4)             >>> index_map = {}
(4)             >>> random_path = nx.generate_random_paths(G, 3, index_map=index_map)
(4)             >>> paths_containing_node_0 = [
(4)             ...     random_path[path_idx] for path_idx in index_map.get(0, [])
(4)             ... ]
(4)             References
(4)             ----------
```

```
1346: (4)               .. [1] Zhang, J., Tang, J., Ma, C., Tong, H., Jing, Y., & Li, J.
1347: (11)                     Panther: Fast top-k similarity search on large networks.
1348: (11)                     In Proceedings of the ACM SIGKDD International Conference
1349: (11)                     on Knowledge Discovery and Data Mining (Vol. 2015-August, pp. 1445–
1454).
1350: (11)                     Association for Computing Machinery.
https://doi.org/10.1145/2783258.2783267.
1351: (4)               """
1352: (4)               import numpy as np
1353: (4)               randint_fn = (
1354: (8)                   seed.integers if isinstance(seed, np.random.Generator) else
seed.randint
1355: (4)               )
1356: (4)               adj_mat = nx.to_numpy_array(G, weight=weight)
1357: (4)               inv_row_sums = np.reciprocal(adj_mat.sum(axis=1)).reshape(-1, 1)
1358: (4)               transition_probabilities = adj_mat * inv_row_sums
1359: (4)               node_map = list(G)
1360: (4)               num_nodes = G.number_of_nodes()
1361: (4)               for path_index in range(sample_size):
1362: (8)                   node_index = randint_fn(num_nodes)
1363: (8)                   node = node_map[node_index]
1364: (8)                   path = [node]
1365: (8)                   if index_map is not None:
1366: (12)                      if node in index_map:
1367: (16)                          index_map[node].add(path_index)
1368: (12)                      else:
1369: (16)                          index_map[node] = {path_index}
1370: (8)                   starting_index = node_index
1371: (8)                   for _ in range(path_length):
1372: (12)                      nbr_index = seed.choice(
1373: (16)                          num_nodes, p=transition_probabilities[starting_index]
1374: (12)                      )
1375: (12)                      starting_index = nbr_index
1376: (12)                      nbr_node = node_map[nbr_index]
1377: (12)                      path.append(nbr_node)
1378: (12)                      if index_map is not None:
1379: (16)                          if nbr_node in index_map:
1380: (20)                              index_map[nbr_node].add(path_index)
1381: (16)                          else:
1382: (20)                              index_map[nbr_node] = {path_index}
1383: (8)                   yield path
```

File 49 - simple_paths.py:

```
1: (0)              from heapq import heappop, heappush
2: (0)              from itertools import count
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.shortest_paths.weighted import _weight_function
5: (0)              from networkx.utils import not_implemented_for, pairwise
6: (0)              __all__ = [
7: (4)                  "all_simple_paths",
8: (4)                  "is_simple_path",
9: (4)                  "shortest_simple_paths",
10: (4)                 "all_simple_edge_paths",
11: (0)             ]
12: (0)             @nx._dispatchable
13: (0)             def is_simple_path(G, nodes):
14: (4)                 """Returns True if and only if `nodes` form a simple path in `G`.
15: (4)                 A *simple path* in a graph is a nonempty sequence of nodes in which
16: (4)                 no node appears more than once in the sequence, and each adjacent
17: (4)                 pair of nodes in the sequence is adjacent in the graph.
18: (4)                 Parameters
19: (4)                 ----------
20: (4)                 G : graph
21: (8)                     A NetworkX graph.
22: (4)                 nodes : list
23: (8)                     A list of one or more nodes in the graph `G`.
```

```
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     Whether the given list of nodes represents a simple path in `G`.
(4)                 Notes
(4)                 -----
(4)                 An empty list of nodes is not a path but a list of one node is a
(4)                 path. Here's an explanation why.
(4)                 This function operates on *node paths*. One could also consider
(4)                 *edge paths*. There is a bijection between node paths and edge
(4)                 paths.
(4)                 The *length of a path* is the number of edges in the path, so a list
(4)                 of nodes of length *n* corresponds to a path of length *n* - 1.
(4)                 Thus the smallest edge path would be a list of zero edges, the empty
(4)                 path. This corresponds to a list of one node.
(4)                 To convert between a node path and an edge path, you can use code
(4)                 like the following::
(8)                     >>> from networkx.utils import pairwise
(8)                     >>> nodes = [0, 1, 2, 3]
(8)                     >>> edges = list(pairwise(nodes))
(8)                     >>> edges
(8)                     [(0, 1), (1, 2), (2, 3)]
(8)                     >>> nodes = [edges[0][0]] + [v for u, v in edges]
(8)                     >>> nodes
(8)                     [0, 1, 2, 3]
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.cycle_graph(4)
(4)                 >>> nx.is_simple_path(G, [2, 3, 0])
(4)                 True
(4)                 >>> nx.is_simple_path(G, [0, 2])
(4)                 False
(4)                 """
(4)                 if len(nodes) == 0:
(8)                     return False
(4)                 if len(nodes) == 1:
(8)                     return nodes[0] in G
(4)                 if not all(n in G for n in nodes):
(8)                     return False
(4)                 if len(set(nodes)) != len(nodes):
(8)                     return False
(4)                 return all(v in G[u] for u, v in pairwise(nodes))
(0)             @nx._dispatchable
(0)             def all_simple_paths(G, source, target, cutoff=None):
(4)                 """Generate all simple paths in the graph G from source to target.
(4)                 A simple path is a path with no repeated nodes.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node
(7)                    Starting node for path
(4)                 target : nodes
(7)                    Single node or iterable of nodes at which to end path
(4)                 cutoff : integer, optional
(8)                     Depth to stop the search. Only paths of length <= cutoff are returned.
(4)                 Returns
(4)                 -------
(4)                 path_generator: generator
(7)                    A generator that produces lists of simple paths.  If there are no paths
(7)                    between the source and target within the given cutoff the generator
(7)                    produces no output. If it is possible to traverse the same sequence of
(7)                    nodes in multiple ways, namely through parallel edges, then it will be
(7)                    returned multiple times (once for each viable edge combination).
(4)                 Examples
(4)                 --------
(4)                 This iterator generates lists of nodes::
(8)                     >>> G = nx.complete_graph(4)
(8)                     >>> for path in nx.all_simple_paths(G, source=0, target=3):
(8)                     ...     print(path)
```

```
93: (8)                     ...
94: (8)                     [0, 1, 2, 3]
95: (8)                     [0, 1, 3]
96: (8)                     [0, 2, 1, 3]
97: (8)                     [0, 2, 3]
98: (8)                     [0, 3]
99: (4)                 You can generate only those paths that are shorter than a certain
100: (4)                length by using the `cutoff` keyword argument::
101: (8)                    >>> paths = nx.all_simple_paths(G, source=0, target=3, cutoff=2)
102: (8)                    >>> print(list(paths))
103: (8)                    [[0, 1, 3], [0, 2, 3], [0, 3]]
104: (4)                To get each path as the corresponding list of edges, you can use the
105: (4)                :func:`networkx.utils.pairwise` helper function::
106: (8)                    >>> paths = nx.all_simple_paths(G, source=0, target=3)
107: (8)                    >>> for path in map(nx.utils.pairwise, paths):
108: (8)                    ...     print(list(path))
109: (8)                    [(0, 1), (1, 2), (2, 3)]
110: (8)                    [(0, 1), (1, 3)]
111: (8)                    [(0, 2), (2, 1), (1, 3)]
112: (8)                    [(0, 2), (2, 3)]
113: (8)                    [(0, 3)]
114: (4)                Pass an iterable of nodes as target to generate all paths ending in any of
several nodes::
115: (8)                    >>> G = nx.complete_graph(4)
116: (8)                    >>> for path in nx.all_simple_paths(G, source=0, target=[3, 2]):
117: (8)                    ...     print(path)
118: (8)                    ...
119: (8)                    [0, 1, 2]
120: (8)                    [0, 1, 2, 3]
121: (8)                    [0, 1, 3]
122: (8)                    [0, 1, 3, 2]
123: (8)                    [0, 2]
124: (8)                    [0, 2, 1, 3]
125: (8)                    [0, 2, 3]
126: (8)                    [0, 3]
127: (8)                    [0, 3, 1, 2]
128: (8)                    [0, 3, 2]
129: (4)                The singleton path from ``source`` to itself is considered a simple path
and is
130: (4)                included in the results:
131: (8)                    >>> G = nx.empty_graph(5)
132: (8)                    >>> list(nx.all_simple_paths(G, source=0, target=0))
133: (8)                    [[0]]
134: (8)                    >>> G = nx.path_graph(3)
135: (8)                    >>> list(nx.all_simple_paths(G, source=0, target={0, 1, 2}))
136: (8)                    [[0], [0, 1], [0, 1, 2]]
137: (4)                Iterate over each path from the root nodes to the leaf nodes in a
138: (4)                directed acyclic graph using a functional programming approach::
139: (8)                    >>> from itertools import chain
140: (8)                    >>> from itertools import product
141: (8)                    >>> from itertools import starmap
142: (8)                    >>> from functools import partial
143: (8)                    >>>
144: (8)                    >>> chaini = chain.from_iterable
145: (8)                    >>>
146: (8)                    >>> G = nx.DiGraph([(0, 1), (1, 2), (0, 3), (3, 2)])
147: (8)                    >>> roots = (v for v, d in G.in_degree() if d == 0)
148: (8)                    >>> leaves = (v for v, d in G.out_degree() if d == 0)
149: (8)                    >>> all_paths = partial(nx.all_simple_paths, G)
150: (8)                    >>> list(chaini(starmap(all_paths, product(roots, leaves))))
151: (8)                    [[0, 1, 2], [0, 3, 2]]
152: (4)                The same list computed using an iterative approach::
153: (8)                    >>> G = nx.DiGraph([(0, 1), (1, 2), (0, 3), (3, 2)])
154: (8)                    >>> roots = (v for v, d in G.in_degree() if d == 0)
155: (8)                    >>> leaves = (v for v, d in G.out_degree() if d == 0)
156: (8)                    >>> all_paths = []
157: (8)                    >>> for root in roots:
158: (8)                    ...     for leaf in leaves:
159: (8)                    ...         paths = nx.all_simple_paths(G, root, leaf)
```

```
(8)                 ...         all_paths.extend(paths)
(8)                 >>> all_paths
(8)                 [[0, 1, 2], [0, 3, 2]]
(4)             Iterate over each path from the root nodes to the leaf nodes in a
(4)             directed acyclic graph passing all leaves together to avoid unnecessary
(4)             compute::
(8)                 >>> G = nx.DiGraph([(0, 1), (2, 1), (1, 3), (1, 4)])
(8)                 >>> roots = (v for v, d in G.in_degree() if d == 0)
(8)                 >>> leaves = [v for v, d in G.out_degree() if d == 0]
(8)                 >>> all_paths = []
(8)                 >>> for root in roots:
(8)                 ...     paths = nx.all_simple_paths(G, root, leaves)
(8)                 ...     all_paths.extend(paths)
(8)                 >>> all_paths
(8)                 [[0, 1, 3], [0, 1, 4], [2, 1, 3], [2, 1, 4]]
(4)             If parallel edges offer multiple ways to traverse a given sequence of
(4)             nodes, this sequence of nodes will be returned multiple times:
(8)                 >>> G = nx.MultiDiGraph([(0, 1), (0, 1), (1, 2)])
(8)                 >>> list(nx.all_simple_paths(G, 0, 2))
(8)                 [[0, 1, 2], [0, 1, 2]]
(4)             Notes
(4)             -----
(4)             This algorithm uses a modified depth-first search to generate the
(4)             paths [1]_.  A single path can be found in $O(V+E)$ time but the
(4)             number of simple paths in a graph can be very large, e.g. $O(n!)$ in
(4)             the complete graph of order $n$.
(4)             This function does not check that a path exists between `source` and
(4)             `target`. For large graphs, this may result in very long runtimes.
(4)             Consider using `has_path` to check that a path exists between `source` and
(4)             `target` before calling this function on large graphs.
(4)             References
(4)             ----------
(4)             .. [1] R. Sedgewick, "Algorithms in C, Part 5: Graph Algorithms",
(7)                Addison Wesley Professional, 3rd ed., 2001.
(4)             See Also
(4)             --------
(4)             all_shortest_paths, shortest_path, has_path
(4)             """
(4)             for edge_path in all_simple_edge_paths(G, source, target, cutoff):
(8)                 yield [source] + [edge[1] for edge in edge_path]
(0)         @nx._dispatchable
(0)         def all_simple_edge_paths(G, source, target, cutoff=None):
(4)             """Generate lists of edges for all simple paths in G from source to
target.
(4)             A simple path is a path with no repeated nodes.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source : node
(7)                Starting node for path
(4)             target : nodes
(7)                Single node or iterable of nodes at which to end path
(4)             cutoff : integer, optional
(8)                 Depth to stop the search. Only paths of length <= cutoff are returned.
(4)             Returns
(4)             -------
(4)             path_generator: generator
(7)                A generator that produces lists of simple paths.  If there are no paths
(7)                between the source and target within the given cutoff the generator
(7)                produces no output.
(7)                For multigraphs, the list of edges have elements of the form `(u,v,k)`.
(7)                Where `k` corresponds to the edge key.
(4)             Examples
(4)             --------
(4)             Print the simple path edges of a Graph::
(8)                 >>> g = nx.Graph([(1, 2), (2, 4), (1, 3), (3, 4)])
(8)                 >>> for path in sorted(nx.all_simple_edge_paths(g, 1, 4)):
(8)                 ...     print(path)
(8)                 [(1, 2), (2, 4)]
```

```
(8)                 [(1, 3), (3, 4)]
(4)             Print the simple path edges of a MultiGraph. Returned edges come with
(4)             their associated keys::
(8)                 >>> mg = nx.MultiGraph()
(8)                 >>> mg.add_edge(1, 2, key="k0")
(8)                 'k0'
(8)                 >>> mg.add_edge(1, 2, key="k1")
(8)                 'k1'
(8)                 >>> mg.add_edge(2, 3, key="k0")
(8)                 'k0'
(8)                 >>> for path in sorted(nx.all_simple_edge_paths(mg, 1, 3)):
(8)                 ...     print(path)
(8)                 [(1, 2, 'k0'), (2, 3, 'k0')]
(8)                 [(1, 2, 'k1'), (2, 3, 'k0')]
(4)             When ``source`` is one of the targets, the empty path starting and ending at
(4)             ``source`` without traversing any edge is considered a valid simple edge path
(4)             and is included in the results:
(8)                 >>> G = nx.Graph()
(8)                 >>> G.add_node(0)
(8)                 >>> paths = list(nx.all_simple_edge_paths(G, 0, 0))
(8)                 >>> for path in paths:
(8)                 ...     print(path)
(8)                 []
(8)                 >>> len(paths)
(8)                 1
(4)             Notes
(4)             -----
(4)             This algorithm uses a modified depth-first search to generate the
(4)             paths [1]_.  A single path can be found in $O(V+E)$ time but the
(4)             number of simple paths in a graph can be very large, e.g. $O(n!)$ in
(4)             the complete graph of order $n$.
(4)             References
(4)             ----------
(4)             .. [1] R. Sedgewick, "Algorithms in C, Part 5: Graph Algorithms",
(7)                Addison Wesley Professional, 3rd ed., 2001.
(4)             See Also
(4)             --------
(4)             all_shortest_paths, shortest_path, all_simple_paths
(4)             """
(4)             if source not in G:
(8)                 raise nx.NodeNotFound(f"source node {source} not in graph")
(4)             if target in G:
(8)                 targets = {target}
(4)             else:
(8)                 try:
(12)                    targets = set(target)
(8)                 except TypeError as err:
(12)                    raise nx.NodeNotFound(f"target node {target} not in graph") from err
(4)             cutoff = cutoff if cutoff is not None else len(G) - 1
(4)             if cutoff >= 0 and targets:
(8)                 yield from _all_simple_edge_paths(G, source, targets, cutoff)
(0)         def _all_simple_edge_paths(G, source, targets, cutoff):
(4)             get_edges = (
(8)                 (lambda node: G.edges(node, keys=True))
(8)                 if G.is_multigraph()
(8)                 else (lambda node: G.edges(node))
(4)             )
(4)             current_path = {None: None}
(4)             stack = [iter([(None, source)])]
(4)             while stack:
(8)                 next_edge = next((e for e in stack[-1] if e[1] not in current_path), None)
(8)                 if next_edge is None:
(12)                    stack.pop()
(12)                    current_path.popitem()
(12)                    continue
```

```
293: (8)                    previous_node, next_node, *_ = next_edge
294: (8)                    if next_node in targets:
295: (12)                       yield (list(current_path.values()) + [next_edge])[2:]  # remove
dummy edge
296: (8)                    if len(current_path) - 1 < cutoff and (
297: (12)                       targets - current_path.keys() - {next_node}
298: (8)                    ):
299: (12)                       current_path[next_node] = next_edge
300: (12)                       stack.append(iter(get_edges(next_node)))
301: (0)            @not_implemented_for("multigraph")
302: (0)            @nx._dispatchable(edge_attrs="weight")
303: (0)            def shortest_simple_paths(G, source, target, weight=None):
304: (4)                """Generate all simple paths in the graph G from source to target,
305: (7)                   starting from shortest ones.
306: (4)                A simple path is a path with no repeated nodes.
307: (4)                If a weighted shortest path search is to be used, no negative weights
308: (4)                are allowed.
309: (4)                Parameters
310: (4)                ----------
311: (4)                G : NetworkX graph
312: (4)                source : node
313: (7)                   Starting node for path
314: (4)                target : node
315: (7)                   Ending node for path
316: (4)                weight : string or function
317: (8)                    If it is a string, it is the name of the edge attribute to be
318: (8)                    used as a weight.
319: (8)                    If it is a function, the weight of an edge is the value returned
320: (8)                    by the function. The function must accept exactly three positional
321: (8)                    arguments: the two endpoints of an edge and the dictionary of edge
322: (8)                    attributes for that edge. The function must return a number.
323: (8)                    If None all edges are considered to have unit weight. Default
324: (8)                    value None.
325: (4)                Returns
326: (4)                -------
327: (4)                path_generator: generator
328: (7)                   A generator that produces lists of simple paths, in order from
329: (7)                   shortest to longest.
330: (4)                Raises
331: (4)                ------
332: (4)                NetworkXNoPath
333: (7)                   If no path exists between source and target.
334: (4)                NetworkXError
335: (7)                   If source or target nodes are not in the input graph.
336: (4)                NetworkXNotImplemented
337: (7)                   If the input graph is a Multi[Di]Graph.
338: (4)                Examples
339: (4)                --------
340: (4)                >>> G = nx.cycle_graph(7)
341: (4)                >>> paths = list(nx.shortest_simple_paths(G, 0, 3))
342: (4)                >>> print(paths)
343: (4)                [[0, 1, 2, 3], [0, 6, 5, 4, 3]]
344: (4)                You can use this function to efficiently compute the k shortest/best
345: (4)                paths between two nodes.
346: (4)                >>> from itertools import islice
347: (4)                >>> def k_shortest_paths(G, source, target, k, weight=None):
348: (4)                ...     return list(
349: (4)                ...         islice(nx.shortest_simple_paths(G, source, target,
weight=weight), k)
350: (4)                ...     )
351: (4)                >>> for path in k_shortest_paths(G, 0, 3, 2):
352: (4)                ...     print(path)
353: (4)                [0, 1, 2, 3]
354: (4)                [0, 6, 5, 4, 3]
355: (4)                Notes
356: (4)                -----
357: (4)                This procedure is based on algorithm by Jin Y. Yen [1]_.  Finding
358: (4)                the first $K$ paths requires $O(KN^3)$ operations.
359: (4)                See Also
```

```
(4)                 --------
(4)                 all_shortest_paths
(4)                 shortest_path
(4)                 all_simple_paths
(4)                 References
(4)                 ----------
(4)                 .. [1] Jin Y. Yen, "Finding the K Shortest Loopless Paths in a
(7)                    Network", Management Science, Vol. 17, No. 11, Theory Series
(7)                    (Jul., 1971), pp. 712-716.
(4)                 """
(4)                 if source not in G:
(8)                     raise nx.NodeNotFound(f"source node {source} not in graph")
(4)                 if target not in G:
(8)                     raise nx.NodeNotFound(f"target node {target} not in graph")
(4)                 if weight is None:
(8)                     length_func = len
(8)                     shortest_path_func = _bidirectional_shortest_path
(4)                 else:
(8)                     wt = _weight_function(G, weight)
(8)                     def length_func(path):
(12)                        return sum(
(16)                            wt(u, v, G.get_edge_data(u, v)) for (u, v) in zip(path,
path[1:])
(12)                        )
(8)                     shortest_path_func = _bidirectional_dijkstra
(4)                 listA = []
(4)                 listB = PathBuffer()
(4)                 prev_path = None
(4)                 while True:
(8)                     if not prev_path:
(12)                        length, path = shortest_path_func(G, source, target,
weight=weight)
(12)                        listB.push(length, path)
(8)                     else:
(12)                        ignore_nodes = set()
(12)                        ignore_edges = set()
(12)                        for i in range(1, len(prev_path)):
(16)                            root = prev_path[:i]
(16)                            root_length = length_func(root)
(16)                            for path in listA:
(20)                                if path[:i] == root:
(24)                                    ignore_edges.add((path[i - 1], path[i]))
(16)                            try:
(20)                                length, spur = shortest_path_func(
(24)                                    G,
(24)                                    root[-1],
(24)                                    target,
(24)                                    ignore_nodes=ignore_nodes,
(24)                                    ignore_edges=ignore_edges,
(24)                                    weight=weight,
(20)                                )
(20)                                path = root[:-1] + spur
(20)                                listB.push(root_length + length, path)
(16)                            except nx.NetworkXNoPath:
(20)                                pass
(16)                            ignore_nodes.add(root[-1])
(8)                     if listB:
(12)                        path = listB.pop()
(12)                        yield path
(12)                        listA.append(path)
(12)                        prev_path = path
(8)                     else:
(12)                        break
(0)             class PathBuffer:
(4)                 def __init__(self):
(8)                     self.paths = set()
(8)                     self.sortedpaths = []
(8)                     self.counter = count()
(4)                 def __len__(self):
```

```
427: (8)                    return len(self.sortedpaths)
428: (4)                def push(self, cost, path):
429: (8)                    hashable_path = tuple(path)
430: (8)                    if hashable_path not in self.paths:
431: (12)                       heappush(self.sortedpaths, (cost, next(self.counter), path))
432: (12)                       self.paths.add(hashable_path)
433: (4)                def pop(self):
434: (8)                    (cost, num, path) = heappop(self.sortedpaths)
435: (8)                    hashable_path = tuple(path)
436: (8)                    self.paths.remove(hashable_path)
437: (8)                    return path
438: (0)            def _bidirectional_shortest_path(
439: (4)                G, source, target, ignore_nodes=None, ignore_edges=None, weight=None
440: (0)            ):
441: (4)                """Returns the shortest path between source and target ignoring
442: (7)                   nodes and edges in the containers ignore_nodes and ignore_edges.
443: (4)                This is a custom modification of the standard bidirectional shortest
444: (4)                path implementation at networkx.algorithms.unweighted
445: (4)                Parameters
446: (4)                ----------
447: (4)                G : NetworkX graph
448: (4)                source : node
449: (7)                   starting node for path
450: (4)                target : node
451: (7)                   ending node for path
452: (4)                ignore_nodes : container of nodes
453: (7)                   nodes to ignore, optional
454: (4)                ignore_edges : container of edges
455: (7)                   edges to ignore, optional
456: (4)                weight : None
457: (7)                   This function accepts a weight argument for convenience of
458: (7)                   shortest_simple_paths function. It will be ignored.
459: (4)                Returns
460: (4)                -------
461: (4)                path: list
462: (7)                   List of nodes in a path from source to target.
463: (4)                Raises
464: (4)                ------
465: (4)                NetworkXNoPath
466: (7)                   If no path exists between source and target.
467: (4)                See Also
468: (4)                --------
469: (4)                shortest_path
470: (4)                """
471: (4)                results = _bidirectional_pred_succ(G, source, target, ignore_nodes,
ignore_edges)
472: (4)                pred, succ, w = results
473: (4)                path = []
474: (4)                while w is not None:
475: (8)                    path.append(w)
476: (8)                    w = succ[w]
477: (4)                w = pred[path[0]]
478: (4)                while w is not None:
479: (8)                    path.insert(0, w)
480: (8)                    w = pred[w]
481: (4)                return len(path), path
482: (0)            def _bidirectional_pred_succ(G, source, target, ignore_nodes=None,
ignore_edges=None):
483: (4)                """Bidirectional shortest path helper.
484: (4)                Returns (pred,succ,w) where
485: (4)                pred is a dictionary of predecessors from w to the source, and
486: (4)                succ is a dictionary of successors from w to the target.
487: (4)                """
488: (4)                if ignore_nodes and (source in ignore_nodes or target in ignore_nodes):
489: (8)                    raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
490: (4)                if target == source:
491: (8)                    return ({target: None}, {source: None}, source)
492: (4)                if G.is_directed():
493: (8)                    Gpred = G.predecessors
```

```
494: (8)                    Gsucc = G.successors
495: (4)                else:
496: (8)                    Gpred = G.neighbors
497: (8)                    Gsucc = G.neighbors
498: (4)                if ignore_nodes:
499: (8)                    def filter_iter(nodes):
500: (12)                       def iterate(v):
501: (16)                           for w in nodes(v):
502: (20)                               if w not in ignore_nodes:
503: (24)                                   yield w
504: (12)                       return iterate
505: (8)                    Gpred = filter_iter(Gpred)
506: (8)                    Gsucc = filter_iter(Gsucc)
507: (4)                if ignore_edges:
508: (8)                    if G.is_directed():
509: (12)                       def filter_pred_iter(pred_iter):
510: (16)                           def iterate(v):
511: (20)                               for w in pred_iter(v):
512: (24)                                   if (w, v) not in ignore_edges:
513: (28)                                       yield w
514: (16)                           return iterate
515: (12)                       def filter_succ_iter(succ_iter):
516: (16)                           def iterate(v):
517: (20)                               for w in succ_iter(v):
518: (24)                                   if (v, w) not in ignore_edges:
519: (28)                                       yield w
520: (16)                           return iterate
521: (12)                       Gpred = filter_pred_iter(Gpred)
522: (12)                       Gsucc = filter_succ_iter(Gsucc)
523: (8)                    else:
524: (12)                       def filter_iter(nodes):
525: (16)                           def iterate(v):
526: (20)                               for w in nodes(v):
527: (24)                                   if (v, w) not in ignore_edges and (w, v) not in
ignore_edges:
528: (28)                                       yield w
529: (16)                           return iterate
530: (12)                       Gpred = filter_iter(Gpred)
531: (12)                       Gsucc = filter_iter(Gsucc)
532: (4)                pred = {source: None}
533: (4)                succ = {target: None}
534: (4)                forward_fringe = [source]
535: (4)                reverse_fringe = [target]
536: (4)                while forward_fringe and reverse_fringe:
537: (8)                    if len(forward_fringe) <= len(reverse_fringe):
538: (12)                       this_level = forward_fringe
539: (12)                       forward_fringe = []
540: (12)                       for v in this_level:
541: (16)                           for w in Gsucc(v):
542: (20)                               if w not in pred:
543: (24)                                   forward_fringe.append(w)
544: (24)                                   pred[w] = v
545: (20)                               if w in succ:
546: (24)                                   return pred, succ, w
547: (8)                    else:
548: (12)                       this_level = reverse_fringe
549: (12)                       reverse_fringe = []
550: (12)                       for v in this_level:
551: (16)                           for w in Gpred(v):
552: (20)                               if w not in succ:
553: (24)                                   succ[w] = v
554: (24)                                   reverse_fringe.append(w)
555: (20)                               if w in pred:
556: (24)                                   return pred, succ, w
557: (4)                raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
558: (0)            def _bidirectional_dijkstra(
559: (4)                G, source, target, weight="weight", ignore_nodes=None, ignore_edges=None
560: (0)            ):
561: (4)                """Dijkstra's algorithm for shortest paths using bidirectional search.
```

```
(4)    This function returns the shortest path between source and target
(4)    ignoring nodes and edges in the containers ignore_nodes and
(4)    ignore_edges.
(4)    This is a custom modification of the standard Dijkstra bidirectional
(4)    shortest path implementation at networkx.algorithms.weighted
(4)    Parameters
(4)    ----------
(4)    G : NetworkX graph
(4)    source : node
(7)       Starting node.
(4)    target : node
(7)       Ending node.
(4)    weight: string, function, optional (default='weight')
(7)       Edge data key or weight function corresponding to the edge weight
(4)    ignore_nodes : container of nodes
(7)       nodes to ignore, optional
(4)    ignore_edges : container of edges
(7)       edges to ignore, optional
(4)    Returns
(4)    -------
(4)    length : number
(8)        Shortest path length.
(4)    Returns a tuple of two dictionaries keyed by node.
(4)    The first dictionary stores distance from the source.
(4)    The second stores the path from the source to that node.
(4)    Raises
(4)    ------
(4)    NetworkXNoPath
(8)        If no path exists between source and target.
(4)    Notes
(4)    -----
(4)    Edge weight attributes must be numerical.
(4)    Distances are calculated as sums of weighted edges traversed.
(4)    In practice  bidirectional Dijkstra is much more than twice as fast as
(4)    ordinary Dijkstra.
(4)    Ordinary Dijkstra expands nodes in a sphere-like manner from the
(4)    source. The radius of this sphere will eventually be the length
(4)    of the shortest path. Bidirectional Dijkstra will expand nodes
(4)    from both the source and the target, making two spheres of half
(4)    this radius. Volume of the first sphere is pi*r*r while the
(4)    others are 2*pi*r/2*r/2, making up half the volume.
(4)    This algorithm is not guaranteed to work if edge weights
(4)    are negative or are floating point numbers
(4)    (overflows and roundoff errors can cause problems).
(4)    See Also
(4)    --------
(4)    shortest_path
(4)    shortest_path_length
(4)    """
(4)    if ignore_nodes and (source in ignore_nodes or target in ignore_nodes):
(8)        raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
(4)    if source == target:
(8)        if source not in G:
(12)           raise nx.NodeNotFound(f"Node {source} not in graph")
(8)        return (0, [source])
(4)    if G.is_directed():
(8)        Gpred = G.predecessors
(8)        Gsucc = G.successors
(4)    else:
(8)        Gpred = G.neighbors
(8)        Gsucc = G.neighbors
(4)    if ignore_nodes:
(8)        def filter_iter(nodes):
(12)           def iterate(v):
(16)               for w in nodes(v):
(20)                   if w not in ignore_nodes:
(24)                       yield w
(12)           return iterate
(8)        Gpred = filter_iter(Gpred)
```

```
(8)                 Gsucc = filter_iter(Gsucc)
(4)             if ignore_edges:
(8)                 if G.is_directed():
(12)                    def filter_pred_iter(pred_iter):
(16)                        def iterate(v):
(20)                            for w in pred_iter(v):
(24)                                if (w, v) not in ignore_edges:
(28)                                    yield w
(16)                        return iterate
(12)                    def filter_succ_iter(succ_iter):
(16)                        def iterate(v):
(20)                            for w in succ_iter(v):
(24)                                if (v, w) not in ignore_edges:
(28)                                    yield w
(16)                        return iterate
(12)                    Gpred = filter_pred_iter(Gpred)
(12)                    Gsucc = filter_succ_iter(Gsucc)
(8)                 else:
(12)                    def filter_iter(nodes):
(16)                        def iterate(v):
(20)                            for w in nodes(v):
(24)                                if (v, w) not in ignore_edges and (w, v) not in ignore_edges:
(28)                                    yield w
(16)                        return iterate
(12)                    Gpred = filter_iter(Gpred)
(12)                    Gsucc = filter_iter(Gsucc)
(4)             push = heappush
(4)             pop = heappop
(4)             dists = [{}, {}]  # dictionary of final distances
(4)             paths = [{source: [source]}, {target: [target]}]  # dictionary of paths
(4)             fringe = [[], []]  # heap of (distance, node) tuples for
(4)             seen = [{source: 0}, {target: 0}]  # dictionary of distances to
(4)             c = count()
(4)             push(fringe[0], (0, next(c), source))
(4)             push(fringe[1], (0, next(c), target))
(4)             neighs = [Gsucc, Gpred]
(4)             finalpath = []
(4)             dir = 1
(4)             while fringe[0] and fringe[1]:
(8)                 dir = 1 - dir
(8)                 (dist, _, v) = pop(fringe[dir])
(8)                 if v in dists[dir]:
(12)                    continue
(8)                 dists[dir][v] = dist  # equal to seen[dir][v]
(8)                 if v in dists[1 - dir]:
(12)                    return (finaldist, finalpath)
(8)                 wt = _weight_function(G, weight)
(8)                 for w in neighs[dir](v):
(12)                    if dir == 0:  # forward
(16)                        minweight = wt(v, w, G.get_edge_data(v, w))
(16)                        vwLength = dists[dir][v] + minweight
(12)                    else:  # back, must remember to change v,w->w,v
(16)                        minweight = wt(w, v, G.get_edge_data(w, v))
(16)                        vwLength = dists[dir][v] + minweight
(12)                    if w in dists[dir]:
(16)                        if vwLength < dists[dir][w]:
(20)                            raise ValueError("Contradictory paths found: negative weights?")
(12)                    elif w not in seen[dir] or vwLength < seen[dir][w]:
(16)                        seen[dir][w] = vwLength
(16)                        push(fringe[dir], (vwLength, next(c), w))
(16)                        paths[dir][w] = paths[dir][v] + [w]
(16)                        if w in seen[0] and w in seen[1]:
(20)                            totaldist = seen[0][w] + seen[1][w]
(20)                            if finalpath == [] or finaldist > totaldist:
(24)                                finaldist = totaldist
(24)                                revpath = paths[1][w][:]
(24)                                revpath.reverse()
```

```
698: (24)                                finalpath = paths[0][w] + revpath[1:]
699: (4)                raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
```

File 50 - smallworld.py:

```
1: (0)              """Functions for estimating the small-world-ness of graphs.
2: (0)              A small world network is characterized by a small average shortest path
length,
3: (0)              and a large clustering coefficient.
4: (0)              Small-worldness is commonly measured with the coefficient sigma or omega.
5: (0)              Both coefficients compare the average clustering coefficient and shortest path
6: (0)              length of a given graph against the same quantities for an equivalent random
7: (0)              or lattice graph.
8: (0)              For more information, see the Wikipedia article on small-world network [1]_.
9: (0)              .. [1] Small-world network:: https://en.wikipedia.org/wiki/Small-world_network
10: (0)             """
11: (0)             import networkx as nx
12: (0)             from networkx.utils import not_implemented_for, py_random_state
13: (0)             __all__ = ["random_reference", "lattice_reference", "sigma", "omega"]
14: (0)             @not_implemented_for("directed")
15: (0)             @not_implemented_for("multigraph")
16: (0)             @py_random_state(3)
17: (0)             @nx._dispatchable(returns_graph=True)
18: (0)             def random_reference(G, niter=1, connectivity=True, seed=None):
19: (4)                 """Compute a random graph by swapping edges of a given graph.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : graph
23: (8)                     An undirected graph with 4 or more nodes.
24: (4)                 niter : integer (optional, default=1)
25: (8)                     An edge is rewired approximately `niter` times.
26: (4)                 connectivity : boolean (optional, default=True)
27: (8)                     When True, ensure connectivity for the randomized graph.
28: (4)                 seed : integer, random_state, or None (default)
29: (8)                     Indicator of random number generation state.
30: (8)                     See :ref:`Randomness<randomness>`.
31: (4)                 Returns
32: (4)                 -------
33: (4)                 G : graph
34: (8)                     The randomized graph.
35: (4)                 Raises
36: (4)                 ------
37: (4)                 NetworkXError
38: (8)                     If there are fewer than 4 nodes or 2 edges in `G`
39: (4)                 Notes
40: (4)                 -----
41: (4)                 The implementation is adapted from the algorithm by Maslov and Sneppen
42: (4)                 (2002) [1]_.
43: (4)                 References
44: (4)                 ----------
45: (4)                 .. [1] Maslov, Sergei, and Kim Sneppen.
46: (11)                       "Specificity and stability in topology of protein networks."
47: (11)                       Science 296.5569 (2002): 910-913.
48: (4)                 """
49: (4)                 if len(G) < 4:
50: (8)                     raise nx.NetworkXError("Graph has fewer than four nodes.")
51: (4)                 if len(G.edges) < 2:
52: (8)                     raise nx.NetworkXError("Graph has fewer that 2 edges")
53: (4)                 from networkx.utils import cumulative_distribution, discrete_sequence
54: (4)                 local_conn = nx.connectivity.local_edge_connectivity
55: (4)                 G = G.copy()
56: (4)                 keys, degrees = zip(*G.degree())  # keys, degree
57: (4)                 cdf = cumulative_distribution(degrees)  # cdf of degree
58: (4)                 nnodes = len(G)
59: (4)                 nedges = nx.number_of_edges(G)
60: (4)                 niter = niter * nedges
61: (4)                 ntries = int(nnodes * nedges / (nnodes * (nnodes - 1) / 2))
```

```
(4)                 swapcount = 0
(4)                 for i in range(niter):
(8)                     n = 0
(8)                     while n < ntries:
(12)                        (ai, ci) = discrete_sequence(2, cdistribution=cdf, seed=seed)
(12)                        if ai == ci:
(16)                            continue  # same source, skip
(12)                        a = keys[ai]  # convert index to label
(12)                        c = keys[ci]
(12)                        b = seed.choice(list(G.neighbors(a)))
(12)                        d = seed.choice(list(G.neighbors(c)))
(12)                        if b in [a, c, d] or d in [a, b, c]:
(16)                            continue  # all vertices should be different
(12)                        if (d not in G[a]) and (b not in G[c]):
(16)                            G.add_edge(a, d)
(16)                            G.add_edge(c, b)
(16)                            G.remove_edge(a, b)
(16)                            G.remove_edge(c, d)
(16)                            if connectivity and local_conn(G, a, b) == 0:
(20)                                G.remove_edge(a, d)
(20)                                G.remove_edge(c, b)
(20)                                G.add_edge(a, b)
(20)                                G.add_edge(c, d)
(16)                            else:
(20)                                swapcount += 1
(20)                                break
(12)                        n += 1
(4)                 return G
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @py_random_state(4)
(0)             @nx._dispatchable(returns_graph=True)
(0)             def lattice_reference(G, niter=5, D=None, connectivity=True, seed=None):
(4)                 """Latticize the given graph by swapping edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     An undirected graph.
(4)                 niter : integer (optional, default=1)
(8)                     An edge is rewired approximately niter times.
(4)                 D : numpy.array (optional, default=None)
(8)                     Distance to the diagonal matrix.
(4)                 connectivity : boolean (optional, default=True)
(8)                     Ensure connectivity for the latticized graph when set to True.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 G : graph
(8)                     The latticized graph.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If there are fewer than 4 nodes or 2 edges in `G`
(4)                 Notes
(4)                 -----
(4)                 The implementation is adapted from the algorithm by Sporns et al. [1]_.
(4)                 which is inspired from the original work by Maslov and Sneppen(2002) [2]_.
(4)                 References
(4)                 ----------
(4)                 .. [1] Sporns, Olaf, and Jonathan D. Zwi.
(7)                    "The small world of the cerebral cortex."
(7)                    Neuroinformatics 2.2 (2004): 145-162.
(4)                 .. [2] Maslov, Sergei, and Kim Sneppen.
(7)                    "Specificity and stability in topology of protein networks."
(7)                    Science 296.5569 (2002): 910-913.
(4)                 """
(4)                 import numpy as np
```

```
(4)                 from networkx.utils import cumulative_distribution, discrete_sequence
(4)                 local_conn = nx.connectivity.local_edge_connectivity
(4)                 if len(G) < 4:
(8)                     raise nx.NetworkXError("Graph has fewer than four nodes.")
(4)                 if len(G.edges) < 2:
(8)                     raise nx.NetworkXError("Graph has fewer that 2 edges")
(4)                 G = G.copy()
(4)                 keys, degrees = zip(*G.degree())  # keys, degree
(4)                 cdf = cumulative_distribution(degrees)  # cdf of degree
(4)                 nnodes = len(G)
(4)                 nedges = nx.number_of_edges(G)
(4)                 if D is None:
(8)                     D = np.zeros((nnodes, nnodes))
(8)                     un = np.arange(1, nnodes)
(8)                     um = np.arange(nnodes - 1, 0, -1)
(8)                     u = np.append((0,), np.where(un < um, un, um))
(8)                     for v in range(int(np.ceil(nnodes / 2))):
(12)                        D[nnodes - v - 1, :] = np.append(u[v + 1 :], u[: v + 1])
(12)                        D[v, :] = D[nnodes - v - 1, :][::-1]
(4)                 niter = niter * nedges
(4)                 max_attempts = int(nnodes * nedges / (nnodes * (nnodes - 1) / 2))
(4)                 for _ in range(niter):
(8)                     n = 0
(8)                     while n < max_attempts:
(12)                        (ai, ci) = discrete_sequence(2, cdistribution=cdf, seed=seed)
(12)                        if ai == ci:
(16)                            continue  # same source, skip
(12)                        a = keys[ai]  # convert index to label
(12)                        c = keys[ci]
(12)                        b = seed.choice(list(G.neighbors(a)))
(12)                        d = seed.choice(list(G.neighbors(c)))
(12)                        bi = keys.index(b)
(12)                        di = keys.index(d)
(12)                        if b in [a, c, d] or d in [a, b, c]:
(16)                            continue  # all vertices should be different
(12)                        if (d not in G[a]) and (b not in G[c]):
(16)                            if D[ai, bi] + D[ci, di] >= D[ai, ci] + D[bi, di]:
(20)                                G.add_edge(a, d)
(20)                                G.add_edge(c, b)
(20)                                G.remove_edge(a, b)
(20)                                G.remove_edge(c, d)
(20)                                if connectivity and local_conn(G, a, b) == 0:
(24)                                    G.remove_edge(a, d)
(24)                                    G.remove_edge(c, b)
(24)                                    G.add_edge(a, b)
(24)                                    G.add_edge(c, d)
(20)                                else:
(24)                                    break
(12)                        n += 1
(4)                 return G
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @py_random_state(3)
(0)             @nx._dispatchable
(0)             def sigma(G, niter=100, nrand=10, seed=None):
(4)                 """Returns the small-world coefficient (sigma) of the given graph.
(4)                 The small-world coefficient is defined as:
(4)                 sigma = C/Cr / L/Lr
(4)                 where C and L are respectively the average clustering coefficient and
(4)                 average shortest path length of G. Cr and Lr are respectively the average
(4)                 clustering coefficient and average shortest path length of an equivalent
(4)                 random graph.
(4)                 A graph is commonly classified as small-world if sigma>1.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     An undirected graph.
(4)                 niter : integer (optional, default=100)
(8)                     Approximate number of rewiring per edge to compute the equivalent
```

```
(8)                 random graph.
(4)             nrand : integer (optional, default=10)
(8)                 Number of random graphs generated to compute the average clustering
(8)                 coefficient (Cr) and average shortest path length (Lr).
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             sigma : float
(8)                 The small-world coefficient of G.
(4)             Notes
(4)             -----
(4)             The implementation is adapted from Humphries et al. [1]_ [2]_.
(4)             References
(4)             ----------
(4)             .. [1] The brainstem reticular formation is a small-world, not scale-free,
(11)                   network M. D. Humphries, K. Gurney and T. J. Prescott,
(11)                   Proc. Roy. Soc. B 2006 273, 503-511, doi:10.1098/rspb.2005.3354.
(4)             .. [2] Humphries and Gurney (2008).
(11)                   "Network 'Small-World-Ness': A Quantitative Method for Determining
(11)                   Canonical Network Equivalence".
(11)                   PLoS One. 3 (4). PMID 18446219. doi:10.1371/journal.pone.0002051.
(4)             """
(4)             import numpy as np
(4)             randMetrics = {"C": [], "L": []}
(4)             for i in range(nrand):
(8)                 Gr = random_reference(G, niter=niter, seed=seed)
(8)                 randMetrics["C"].append(nx.transitivity(Gr))
(8)                 randMetrics["L"].append(nx.average_shortest_path_length(Gr))
(4)             C = nx.transitivity(G)
(4)             L = nx.average_shortest_path_length(G)
(4)             Cr = np.mean(randMetrics["C"])
(4)             Lr = np.mean(randMetrics["L"])
(4)             sigma = (C / Cr) / (L / Lr)
(4)             return float(sigma)
(0)         @not_implemented_for("directed")
(0)         @not_implemented_for("multigraph")
(0)         @py_random_state(3)
(0)         @nx._dispatchable
(0)         def omega(G, niter=5, nrand=10, seed=None):
(4)             """Returns the small-world coefficient (omega) of a graph
(4)             The small-world coefficient of a graph G is:
(4)             omega = Lr/L - C/Cl
(4)             where C and L are respectively the average clustering coefficient and
(4)             average shortest path length of G. Lr is the average shortest path length
(4)             of an equivalent random graph and Cl is the average clustering coefficient
(4)             of an equivalent lattice graph.
(4)             The small-world coefficient (omega) measures how much G is like a lattice
(4)             or a random graph. Negative values mean G is similar to a lattice whereas
(4)             positive values mean G is a random graph.
(4)             Values close to 0 mean that G has small-world characteristics.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(8)                 An undirected graph.
(4)             niter: integer (optional, default=5)
(8)                 Approximate number of rewiring per edge to compute the equivalent
(8)                 random graph.
(4)             nrand: integer (optional, default=10)
(8)                 Number of random graphs generated to compute the maximal clustering
(8)                 coefficient (Cr) and average shortest path length (Lr).
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             omega : float
(8)                 The small-world coefficient (omega)
```

```
269: (4)                Notes
270: (4)                -----
271: (4)                The implementation is adapted from the algorithm by Telesford et al. [1]_.
272: (4)                References
273: (4)                ----------
274: (4)                .. [1] Telesford, Joyce, Hayasaka, Burdette, and Laurienti (2011).
275: (11)                      "The Ubiquity of Small-World Networks".
276: (11)                      Brain Connectivity. 1 (0038): 367-75.  PMC 3604768. PMID 22432451.
277: (11)                      doi:10.1089/brain.2011.0038.
278: (4)                """
279: (4)                import numpy as np
280: (4)                randMetrics = {"C": [], "L": []}
281: (4)                Cl = nx.average_clustering(G)
282: (4)                niter_lattice_reference = niter
283: (4)                niter_random_reference = niter * 2
284: (4)                for _ in range(nrand):
285: (8)                    Gr = random_reference(G, niter=niter_random_reference, seed=seed)
286: (8)                    randMetrics["L"].append(nx.average_shortest_path_length(Gr))
287: (8)                    Gl = lattice_reference(G, niter=niter_lattice_reference, seed=seed)
288: (8)                    Cl_temp = nx.average_clustering(Gl)
289: (8)                    if Cl_temp > Cl:
290: (12)                       Cl = Cl_temp
291: (4)                C = nx.average_clustering(G)
292: (4)                L = nx.average_shortest_path_length(G)
293: (4)                Lr = np.mean(randMetrics["L"])
294: (4)                omega = (Lr / L) - (C / Cl)
295: (4)                return float(omega)
```

File 51 - smetric.py:

```
1: (0)              import networkx as nx
2: (0)              __all__ = ["s_metric"]
3: (0)              @nx._dispatchable
4: (0)              def s_metric(G, **kwargs):
5: (4)                  """Returns the s-metric [1]_ of graph.
6: (4)                  The s-metric is defined as the sum of the products ``deg(u) * deg(v)``
7: (4)                  for every edge ``(u, v)`` in `G`.
8: (4)                  Parameters
9: (4)                  ----------
10: (4)                 G : graph
11: (8)                     The graph used to compute the s-metric.
12: (4)                 normalized : bool (optional)
13: (8)                     Normalize the value.
14: (8)                     .. deprecated:: 3.2
15: (11)                       The `normalized` keyword argument is deprecated and will be removed
16: (11)                       in the future
17: (4)                 Returns
18: (4)                 -------
19: (4)                 s : float
20: (8)                     The s-metric of the graph.
21: (4)                 References
22: (4)                 ----------
23: (4)                 .. [1] Lun Li, David Alderson, John C. Doyle, and Walter Willinger,
24: (11)                       Towards a Theory of Scale-Free Graphs:
25: (11)                       Definition, Properties, and  Implications (Extended Version), 2005.
26: (11)                       https://arxiv.org/abs/cond-mat/0501169
27: (4)                 """
28: (4)                 if kwargs:
29: (8)                     if "normalized" in kwargs:
30: (12)                        import warnings
31: (12)                        warnings.warn(
32: (16)                            "\n\nThe `normalized` keyword is deprecated and will be
removed\n"
33: (16)                            "in the future. To silence this warning, remove
`normalized`\n"
34: (16)                            "when calling `s_metric`.\n\n"
35: (16)                            "The value of `normalized` is ignored.",
```

```
36: (16)                            DeprecationWarning,
37: (16)                            stacklevel=3,
38: (12)                        )
39: (8)                     else:
40: (12)                        raise TypeError(
41: (16)                            f"s_metric got an unexpected keyword argument
'{list(kwargs.keys())[0]}'"
42: (12)                        )
43: (4)                 return float(sum(G.degree(u) * G.degree(v) for (u, v) in G.edges()))
```

File 52 - sparsifiers.py:

```
1: (0)              """Functions for computing sparsifiers of graphs."""
2: (0)              import math
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for, py_random_state
5: (0)              __all__ = ["spanner"]
6: (0)              @not_implemented_for("directed")
7: (0)              @not_implemented_for("multigraph")
8: (0)              @py_random_state(3)
9: (0)              @nx._dispatchable(edge_attrs="weight", returns_graph=True)
10: (0)             def spanner(G, stretch, weight=None, seed=None):
11: (4)                 """Returns a spanner of the given graph with the given stretch.
12: (4)                 A spanner of a graph G = (V, E) with stretch t is a subgraph
13: (4)                 H = (V, E_S) such that E_S is a subset of E and the distance between
14: (4)                 any pair of nodes in H is at most t times the distance between the
15: (4)                 nodes in G.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G : NetworkX graph
19: (8)                     An undirected simple graph.
20: (4)                 stretch : float
21: (8)                     The stretch of the spanner.
22: (4)                 weight : object
23: (8)                     The edge attribute to use as distance.
24: (4)                 seed : integer, random_state, or None (default)
25: (8)                     Indicator of random number generation state.
26: (8)                     See :ref:`Randomness<randomness>`.
27: (4)                 Returns
28: (4)                 -------
29: (4)                 NetworkX graph
30: (8)                     A spanner of the given graph with the given stretch.
31: (4)                 Raises
32: (4)                 ------
33: (4)                 ValueError
34: (8)                     If a stretch less than 1 is given.
35: (4)                 Notes
36: (4)                 -----
37: (4)                 This function implements the spanner algorithm by Baswana and Sen,
38: (4)                 see [1].
39: (4)                 This algorithm is a randomized las vegas algorithm: The expected
40: (4)                 running time is O(km) where k = (stretch + 1) // 2 and m is the
41: (4)                 number of edges in G. The returned graph is always a spanner of the
42: (4)                 given graph with the specified stretch. For weighted graphs the
43: (4)                 number of edges in the spanner is O(k * n^(1 + 1 / k)) where k is
44: (4)                 defined as above and n is the number of nodes in G. For unweighted
45: (4)                 graphs the number of edges is O(n^(1 + 1 / k) + kn).
46: (4)                 References
47: (4)                 ----------
48: (4)                 [1] S. Baswana, S. Sen. A Simple and Linear Time Randomized
49: (4)                 Algorithm for Computing Sparse Spanners in Weighted Graphs.
50: (4)                 Random Struct. Algorithms 30(4): 532-563 (2007).
51: (4)                 """
52: (4)                 if stretch < 1:
53: (8)                     raise ValueError("stretch must be at least 1")
54: (4)                 k = (stretch + 1) // 2
55: (4)                 H = nx.empty_graph()
```

```
56: (4)                 H.add_nodes_from(G.nodes)
57: (4)                 residual_graph = _setup_residual_graph(G, weight)
58: (4)                 clustering = {v: v for v in G.nodes}
59: (4)                 sample_prob = math.pow(G.number_of_nodes(), -1 / k)
60: (4)                 size_limit = 2 * math.pow(G.number_of_nodes(), 1 + 1 / k)
61: (4)                 i = 0
62: (4)                 while i < k - 1:
63: (8)                     sampled_centers = set()
64: (8)                     for center in set(clustering.values()):
65: (12)                        if seed.random() < sample_prob:
66: (16)                            sampled_centers.add(center)
67: (8)                     edges_to_add = set()
68: (8)                     edges_to_remove = set()
69: (8)                     new_clustering = {}
70: (8)                     for v in residual_graph.nodes:
71: (12)                        if clustering[v] in sampled_centers:
72: (16)                            continue
73: (12)                        lightest_edge_neighbor, lightest_edge_weight =
_lightest_edge_dicts(
74: (16)                            residual_graph, clustering, v
75: (12)                        )
76: (12)                        neighboring_sampled_centers = (
77: (16)                            set(lightest_edge_weight.keys()) & sampled_centers
78: (12)                        )
79: (12)                        if not neighboring_sampled_centers:
80: (16)                            for neighbor in lightest_edge_neighbor.values():
81: (20)                                edges_to_add.add((v, neighbor))
82: (16)                            for neighbor in residual_graph.adj[v]:
83: (20)                                edges_to_remove.add((v, neighbor))
84: (12)                        else:  # there is a neighboring sampled center
85: (16)                            closest_center = min(
86: (20)                                neighboring_sampled_centers, key=lightest_edge_weight.get
87: (16)                            )
88: (16)                            closest_center_weight = lightest_edge_weight[closest_center]
89: (16)                            closest_center_neighbor =
lightest_edge_neighbor[closest_center]
90: (16)                            edges_to_add.add((v, closest_center_neighbor))
91: (16)                            new_clustering[v] = closest_center
92: (16)                            for center, edge_weight in lightest_edge_weight.items():
93: (20)                                if edge_weight < closest_center_weight:
94: (24)                                    neighbor = lightest_edge_neighbor[center]
95: (24)                                    edges_to_add.add((v, neighbor))
96: (16)                            for neighbor in residual_graph.adj[v]:
97: (20)                                nbr_cluster = clustering[neighbor]
98: (20)                                nbr_weight = lightest_edge_weight[nbr_cluster]
99: (20)                                if (
100: (24)                                   nbr_cluster == closest_center
101: (24)                                   or nbr_weight < closest_center_weight
102: (20)                               ):
103: (24)                                   edges_to_remove.add((v, neighbor))
104: (8)                    if len(edges_to_add) > size_limit:
105: (12)                       continue
106: (8)                    i = i + 1
107: (8)                    for u, v in edges_to_add:
108: (12)                       _add_edge_to_spanner(H, residual_graph, u, v, weight)
109: (8)                    residual_graph.remove_edges_from(edges_to_remove)
110: (8)                    for node, center in clustering.items():
111: (12)                       if center in sampled_centers:
112: (16)                           new_clustering[node] = center
113: (8)                    clustering = new_clustering
114: (8)                    for u in residual_graph.nodes:
115: (12)                       for v in list(residual_graph.adj[u]):
116: (16)                           if clustering[u] == clustering[v]:
117: (20)                               residual_graph.remove_edge(u, v)
118: (8)                    for v in list(residual_graph.nodes):
119: (12)                       if v not in clustering:
120: (16)                           residual_graph.remove_node(v)
121: (4)                for v in residual_graph.nodes:
122: (8)                    lightest_edge_neighbor, _ = _lightest_edge_dicts(residual_graph,
```

```
clustering, v)
(8)                     for neighbor in lightest_edge_neighbor.values():
(12)                        _add_edge_to_spanner(H, residual_graph, v, neighbor, weight)
(4)                 return H
(0)             def _setup_residual_graph(G, weight):
(4)                 """Setup residual graph as a copy of G with unique edges weights.
(4)                 The node set of the residual graph corresponds to the set V' from
(4)                 the Baswana-Sen paper and the edge set corresponds to the set E'
(4)                 from the paper.
(4)                 This function associates distinct weights to the edges of the
(4)                 residual graph (even for unweighted input graphs), as required by
(4)                 the algorithm.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     An undirected simple graph.
(4)                 weight : object
(8)                     The edge attribute to use as distance.
(4)                 Returns
(4)                 -------
(4)                 NetworkX graph
(8)                     The residual graph used for the Baswana-Sen algorithm.
(4)                 """
(4)                 residual_graph = G.copy()
(4)                 for u, v in G.edges():
(8)                     if not weight:
(12)                        residual_graph[u][v]["weight"] = (id(u), id(v))
(8)                     else:
(12)                        residual_graph[u][v]["weight"] = (G[u][v][weight], id(u), id(v))
(4)                 return residual_graph
(0)             def _lightest_edge_dicts(residual_graph, clustering, node):
(4)                 """Find the lightest edge to each cluster.
(4)                 Searches for the minimum-weight edge to each cluster adjacent to
(4)                 the given node.
(4)                 Parameters
(4)                 ----------
(4)                 residual_graph : NetworkX graph
(8)                     The residual graph used by the Baswana-Sen algorithm.
(4)                 clustering : dictionary
(8)                     The current clustering of the nodes.
(4)                 node : node
(8)                     The node from which the search originates.
(4)                 Returns
(4)                 -------
(4)                 lightest_edge_neighbor, lightest_edge_weight : dictionary, dictionary
(8)                     lightest_edge_neighbor is a dictionary that maps a center C to
(8)                     a node v in the corresponding cluster such that the edge from
(8)                     the given node to v is the lightest edge from the given node to
(8)                     any node in cluster. lightest_edge_weight maps a center C to the
(8)                     weight of the aforementioned edge.
(4)                 Notes
(4)                 -----
(4)                 If a cluster has no node that is adjacent to the given node in the
(4)                 residual graph then the center of the cluster is not a key in the
(4)                 returned dictionaries.
(4)                 """
(4)                 lightest_edge_neighbor = {}
(4)                 lightest_edge_weight = {}
(4)                 for neighbor in residual_graph.adj[node]:
(8)                     nbr_center = clustering[neighbor]
(8)                     weight = residual_graph[node][neighbor]["weight"]
(8)                     if (
(12)                        nbr_center not in lightest_edge_weight
(12)                        or weight < lightest_edge_weight[nbr_center]
(8)                     ):
(12)                        lightest_edge_neighbor[nbr_center] = neighbor
(12)                        lightest_edge_weight[nbr_center] = weight
(4)                 return lightest_edge_neighbor, lightest_edge_weight
(0)             def _add_edge_to_spanner(H, residual_graph, u, v, weight):
```

```
191: (4)                """Add the edge {u, v} to the spanner H and take weight from
192: (4)                the residual graph.
193: (4)                Parameters
194: (4)                ----------
195: (4)                H : NetworkX graph
196: (8)                    The spanner under construction.
197: (4)                residual_graph : NetworkX graph
198: (8)                    The residual graph used by the Baswana-Sen algorithm. The weight
199: (8)                    for the edge is taken from this graph.
200: (4)                u : node
201: (8)                    One endpoint of the edge.
202: (4)                v : node
203: (8)                    The other endpoint of the edge.
204: (4)                weight : object
205: (8)                    The edge attribute to use as distance.
206: (4)                """
207: (4)                H.add_edge(u, v)
208: (4)                if weight:
209: (8)                    H[u][v][weight] = residual_graph[u][v]["weight"][0]
```

File 53 - structuralholes.py:

```
1: (0)              """Functions for computing measures of structural holes."""
2: (0)              import networkx as nx
3: (0)              __all__ = ["constraint", "local_constraint", "effective_size"]
4: (0)              @nx._dispatchable(edge_attrs="weight")
5: (0)              def mutual_weight(G, u, v, weight=None):
6: (4)                  """Returns the sum of the weights of the edge from `u` to `v` and
7: (4)                  the edge from `v` to `u` in `G`.
8: (4)                  `weight` is the edge data key that represents the edge weight. If
9: (4)                  the specified key is `None` or is not in the edge data for an edge,
10: (4)                 that edge is assumed to have weight 1.
11: (4)                 Pre-conditions: `u` and `v` must both be in `G`.
12: (4)                 """
13: (4)                 try:
14: (8)                     a_uv = G[u][v].get(weight, 1)
15: (4)                 except KeyError:
16: (8)                     a_uv = 0
17: (4)                 try:
18: (8)                     a_vu = G[v][u].get(weight, 1)
19: (4)                 except KeyError:
20: (8)                     a_vu = 0
21: (4)                 return a_uv + a_vu
22: (0)             @nx._dispatchable(edge_attrs="weight")
23: (0)             def normalized_mutual_weight(G, u, v, norm=sum, weight=None):
24: (4)                 """Returns normalized mutual weight of the edges from `u` to `v`
25: (4)                 with respect to the mutual weights of the neighbors of `u` in `G`.
26: (4)                 `norm` specifies how the normalization factor is computed. It must
27: (4)                 be a function that takes a single argument and returns a number.
28: (4)                 The argument will be an iterable of mutual weights
29: (4)                 of pairs ``(u, w)``, where ``w`` ranges over each (in- and
30: (4)                 out-)neighbor of ``u``. Commons values for `normalization` are
31: (4)                 ``sum`` and ``max``.
32: (4)                 `weight` can be ``None`` or a string, if None, all edge weights
33: (4)                 are considered equal. Otherwise holds the name of the edge
34: (4)                 attribute used as weight.
35: (4)                 """
36: (4)                 scale = norm(mutual_weight(G, u, w, weight) for w in
set(nx.all_neighbors(G, u)))
37: (4)                 return 0 if scale == 0 else mutual_weight(G, u, v, weight) / scale
38: (0)             @nx._dispatchable(edge_attrs="weight")
39: (0)             def effective_size(G, nodes=None, weight=None):
40: (4)                 r"""Returns the effective size of all nodes in the graph ``G``.
41: (4)                 The *effective size* of a node's ego network is based on the concept
42: (4)                 of redundancy. A person's ego network has redundancy to the extent
43: (4)                 that her contacts are connected to each other as well. The
44: (4)                 nonredundant part of a person's relationships is the effective
```

```
size of her ego network [1]_.  Formally, the effective size of a
node $u$, denoted $e(u)$, is defined by
.. math::
   e(u) = \sum_{v \in N(u) \setminus \{u\}}
   \left(1 - \sum_{w \in N(v)} p_{uw} m_{vw}\right)
where $N(u)$ is the set of neighbors of $u$ and $p_{uw}$ is the
normalized mutual weight of the (directed or undirected) edges
joining $u$ and $v$, for each vertex $u$ and $v$ [1]_. And $m_{vw}$
is the mutual weight of $v$ and $w$ divided by $v$ highest mutual
weight with any of its neighbors. The *mutual weight* of $u$ and $v$
is the sum of the weights of edges joining them (edge weights are
assumed to be one if the graph is unweighted).
For the case of unweighted and undirected graphs, Borgatti proposed
a simplified formula to compute effective size [2]_
.. math::
   e(u) = n - \frac{2t}{n}
where `t` is the number of ties in the ego network (not including
ties to ego) and `n` is the number of nodes (excluding ego).
Parameters
----------
G : NetworkX graph
    The graph containing ``v``. Directed graphs are treated like
    undirected graphs when computing neighbors of ``v``.
nodes : container, optional
    Container of nodes in the graph ``G`` to compute the effective size.
    If None, the effective size of every node is computed.
weight : None or string, optional
  If None, all edge weights are considered equal.
  Otherwise holds the name of the edge attribute used as weight.
Returns
-------
dict
    Dictionary with nodes as keys and the effective size of the node as values.
Notes
-----
Burt also defined the related concept of *efficiency* of a node's ego
network, which is its effective size divided by the degree of that
node [1]_. So you can easily compute efficiency:
>>> G = nx.DiGraph()
>>> G.add_edges_from([(0, 1), (0, 2), (1, 0), (2, 1)])
>>> esize = nx.effective_size(G)
>>> efficiency = {n: v / G.degree(n) for n, v in esize.items()}
See also
--------
constraint
References
----------
.. [1] Burt, Ronald S.
       *Structural Holes: The Social Structure of Competition.*
       Cambridge: Harvard University Press, 1995.
.. [2] Borgatti, S.
       "Structural Holes: Unpacking Burt's Redundancy Measures"
       CONNECTIONS 20(1):35-38.
       http://www.analytictech.com/connections/v20(1)/holes.htm
"""
def redundancy(G, u, v, weight=None):
    nmw = normalized_mutual_weight
    r = sum(
        nmw(G, u, w, weight=weight) * nmw(G, v, w, norm=max, weight=weight)

        for w in set(nx.all_neighbors(G, u))
    )
    return 1 - r
effective_size = {}
if nodes is None:
    nodes = G
if not G.is_directed() and weight is None:
    for v in nodes:
```

```
(12)                        if len(G[v]) == 0:
(16)                            effective_size[v] = float("nan")
(16)                            continue
(12)                        E = nx.ego_graph(G, v, center=False, undirected=True)
(12)                        effective_size[v] = len(E) - (2 * E.size()) / len(E)
(4)                 else:
(8)                     for v in nodes:
(12)                        if len(G[v]) == 0:
(16)                            effective_size[v] = float("nan")
(16)                            continue
(12)                        effective_size[v] = sum(
(16)                            redundancy(G, v, u, weight) for u in set(nx.all_neighbors(G, v))
(12)                        )
(4)                 return effective_size
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def constraint(G, nodes=None, weight=None):
(4)                 r"""Returns the constraint on all nodes in the graph ``G``.
(4)                 The *constraint* is a measure of the extent to which a node *v* is
(4)                 invested in those nodes that are themselves invested in the
(4)                 neighbors of *v*. Formally, the *constraint on v*, denoted `c(v)`,
(4)                 is defined by
(4)                 .. math::
(7)                    c(v) = \sum_{w \in N(v) \setminus \{v\}} \ell(v, w)
(4)                 where $N(v)$ is the subset of the neighbors of `v` that are either
(4)                 predecessors or successors of `v` and $\ell(v, w)$ is the local
(4)                 constraint on `v` with respect to `w` [1]_. For the definition of local
(4)                 constraint, see :func:`local_constraint`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     The graph containing ``v``. This can be either directed or undirected.
(4)                 nodes : container, optional
(8)                     Container of nodes in the graph ``G`` to compute the constraint. If
(8)                     None, the constraint of every node is computed.
(4)                 weight : None or string, optional
(6)                   If None, all edge weights are considered equal.
(6)                   Otherwise holds the name of the edge attribute used as weight.
(4)                 Returns
(4)                 -------
(4)                 dict
(8)                     Dictionary with nodes as keys and the constraint on the node as values.
(4)                 See also
(4)                 --------
(4)                 local_constraint
(4)                 References
(4)                 ----------
(4)                 .. [1] Burt, Ronald S.
(11)                       "Structural holes and good ideas".
(11)                       American Journal of Sociology (110): 349–399.
(4)                 """
(4)                 if nodes is None:
(8)                     nodes = G
(4)                 constraint = {}
(4)                 for v in nodes:
(8)                     if len(G[v]) == 0:
(12)                        constraint[v] = float("nan")
(12)                        continue
(8)                     constraint[v] = sum(
(12)                        local_constraint(G, v, n, weight) for n in set(nx.all_neighbors(G, v))
(8)                     )
(4)                 return constraint
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def local_constraint(G, u, v, weight=None):
(4)                 r"""Returns the local constraint on the node ``u`` with respect to
(4)                 the node ``v`` in the graph ``G``.
(4)                 Formally, the *local constraint on u with respect to v*, denoted
```

```
178: (4)                $\ell(u, v)$, is defined by
179: (4)                .. math::
180: (7)                   \ell(u, v) = \left(p_{uv} + \sum_{w \in N(v)} p_{uw} p_{wv}\right)^2,
181: (4)                where $N(v)$ is the set of neighbors of $v$ and $p_{uv}$ is the
182: (4)                normalized mutual weight of the (directed or undirected) edges
183: (4)                joining $u$ and $v$, for each vertex $u$ and $v$ [1]_. The *mutual
184: (4)                weight* of $u$ and $v$ is the sum of the weights of edges joining
185: (4)                them (edge weights are assumed to be one if the graph is
186: (4)                unweighted).
187: (4)                Parameters
188: (4)                ----------
189: (4)                G : NetworkX graph
190: (8)                    The graph containing ``u`` and ``v``. This can be either
191: (8)                    directed or undirected.
192: (4)                u : node
193: (8)                    A node in the graph ``G``.
194: (4)                v : node
195: (8)                    A node in the graph ``G``.
196: (4)                weight : None or string, optional
197: (6)                  If None, all edge weights are considered equal.
198: (6)                  Otherwise holds the name of the edge attribute used as weight.
199: (4)                Returns
200: (4)                -------
201: (4)                float
202: (8)                    The constraint of the node ``v`` in the graph ``G``.
203: (4)                See also
204: (4)                --------
205: (4)                constraint
206: (4)                References
207: (4)                ----------
208: (4)                .. [1] Burt, Ronald S.
209: (11)                      "Structural holes and good ideas".
210: (11)                      American Journal of Sociology (110): 349–399.
211: (4)                """
212: (4)                nmw = normalized_mutual_weight
213: (4)                direct = nmw(G, u, v, weight=weight)
214: (4)                indirect = sum(
215: (8)                    nmw(G, u, w, weight=weight) * nmw(G, w, v, weight=weight)
216: (8)                    for w in set(nx.all_neighbors(G, u))
217: (4)                )
218: (4)                return (direct + indirect) ** 2
```

File 54 - summarization.py:

```
1: (0)              """
2: (0)              Graph summarization finds smaller representations of graphs resulting in
faster
3: (0)              runtime of algorithms, reduced storage needs, and noise reduction.
4: (0)              Summarization has applications in areas such as visualization, pattern mining,
5: (0)              clustering and community detection, and more.  Core graph summarization
6: (0)              techniques are grouping/aggregation, bit-compression,
7: (0)              simplification/sparsification, and influence based. Graph summarization
8: (0)              algorithms often produce either summary graphs in the form of supergraphs or
9: (0)              sparsified graphs, or a list of independent structures. Supergraphs are the
10: (0)             most common product, which consist of supernodes and original nodes and are
11: (0)             connected by edges and superedges, which represent aggregate edges between
12: (0)             nodes and supernodes.
13: (0)             Grouping/aggregation based techniques compress graphs by representing
14: (0)             close/connected nodes and edges in a graph by a single node/edge in a
15: (0)             supergraph. Nodes can be grouped together into supernodes based on their
16: (0)             structural similarities or proximity within a graph to reduce the total number
17: (0)             of nodes in a graph. Edge-grouping techniques group edges into lossy/lossless
18: (0)             nodes called compressor or virtual nodes to reduce the total number of edges
in
19: (0)             a graph. Edge-grouping techniques can be lossless, meaning that they can be
20: (0)             used to re-create the original graph, or techniques can be lossy, requiring
21: (0)             less space to store the summary graph, but at the expense of lower
```

```
 22: (0)             reconstruction accuracy of the original graph.
 23: (0)             Bit-compression techniques minimize the amount of information needed to
 24: (0)             describe the original graph, while revealing structural patterns in the
 25: (0)             original graph.  The two-part minimum description length (MDL) is often used
to
 26: (0)             represent the model and the original graph in terms of the model.  A key
 27: (0)             difference between graph compression and graph summarization is that graph
 28: (0)             summarization focuses on finding structural patterns within the original
graph,
 29: (0)             whereas graph compression focuses on compressions the original graph to be as
 30: (0)             small as possible.  **NOTE**: Some bit-compression methods exist solely to
 31: (0)             compress a graph without creating a summary graph or finding comprehensible
 32: (0)             structural patterns.
 33: (0)             Simplification/Sparsification techniques attempt to create a sparse
 34: (0)             representation of a graph by removing unimportant nodes and edges from the
 35: (0)             graph.  Sparsified graphs differ from supergraphs created by
 36: (0)             grouping/aggregation by only containing a subset of the original nodes and
 37: (0)             edges of the original graph.
 38: (0)             Influence based techniques aim to find a high-level description of influence
 39: (0)             propagation in a large graph.  These methods are scarce and have been mostly
 40: (0)             applied to social graphs.
 41: (0)             *dedensification* is a grouping/aggregation based technique to compress the
 42: (0)             neighborhoods around high-degree nodes in unweighted graphs by adding
 43: (0)             compressor nodes that summarize multiple edges of the same type to
 44: (0)             high-degree nodes (nodes with a degree greater than a given threshold).
 45: (0)             Dedensification was developed for the purpose of increasing performance of
 46: (0)             query processing around high-degree nodes in graph databases and enables
direct
 47: (0)             operations on the compressed graph.  The structural patterns surrounding
 48: (0)             high-degree nodes in the original is preserved while using fewer edges and
 49: (0)             adding a small number of compressor nodes.  The degree of nodes present in the
 50: (0)             original graph is also preserved. The current implementation of
dedensification
 51: (0)             supports graphs with one edge type.
 52: (0)             For more information on graph summarization, see `Graph Summarization Methods
 53: (0)             and Applications: A Survey <https://dl.acm.org/doi/abs/10.1145/3186727>`_
 54: (0)             """
 55: (0)             from collections import Counter, defaultdict
 56: (0)             import networkx as nx
 57: (0)             __all__ = ["dedensify", "snap_aggregation"]
 58: (0)             @nx._dispatchable(mutates_input={"not copy": 3}, returns_graph=True)
 59: (0)             def dedensify(G, threshold, prefix=None, copy=True):
 60: (4)                 """Compresses neighborhoods around high-degree nodes
 61: (4)                 Reduces the number of edges to high-degree nodes by adding compressor
nodes
 62: (4)                 that summarize multiple edges of the same type to high-degree nodes (nodes
 63: (4)                 with a degree greater than a given threshold).  Dedensification also has
 64: (4)                 the added benefit of reducing the number of edges around high-degree
nodes.
 65: (4)                 The implementation currently supports graphs with a single edge type.
 66: (4)                 Parameters
 67: (4)                 ----------
 68: (4)                 G: graph
 69: (7)                    A networkx graph
 70: (4)                 threshold: int
 71: (7)                    Minimum degree threshold of a node to be considered a high degree node.
 72: (7)                    The threshold must be greater than or equal to 2.
 73: (4)                 prefix: str or None, optional (default: None)
 74: (7)                    An optional prefix for denoting compressor nodes
 75: (4)                 copy: bool, optional (default: True)
 76: (7)                    Indicates if dedensification should be done inplace
 77: (4)                 Returns
 78: (4)                 -------
 79: (4)                 dedensified networkx graph : (graph, set)
 80: (8)                     2-tuple of the dedensified graph and set of compressor nodes
 81: (4)                 Notes
 82: (4)                 -----
 83: (4)                 According to the algorithm in [1]_, removes edges in a graph by
 84: (4)                 compressing/decompressing the neighborhoods around high degree nodes by
```

```
    adding compressor nodes that summarize multiple edges of the same type
    to high-degree nodes.  Dedensification will only add a compressor node when

    doing so will reduce the total number of edges in the given graph. This
    implementation currently supports graphs with a single edge type.
    Examples
    --------
    Dedensification will only add compressor nodes when doing so would result
    in fewer edges::
        >>> original_graph = nx.DiGraph()
        >>> original_graph.add_nodes_from(
        ...     ["1", "2", "3", "4", "5", "6", "A", "B", "C"]
        ... )
        >>> original_graph.add_edges_from(
        ...     [
        ...         ("1", "C"), ("1", "B"),
        ...         ("2", "C"), ("2", "B"), ("2", "A"),
        ...         ("3", "B"), ("3", "A"), ("3", "6"),
        ...         ("4", "C"), ("4", "B"), ("4", "A"),
        ...         ("5", "B"), ("5", "A"),
        ...         ("6", "5"),
        ...         ("A", "6")
        ...     ]
        ... )
        >>> c_graph, c_nodes = nx.dedensify(original_graph, threshold=2)
        >>> original_graph.number_of_edges()
        15
        >>> c_graph.number_of_edges()
        14
    A dedensified, directed graph can be "densified" to reconstruct the
    original graph::
        >>> original_graph = nx.DiGraph()
        >>> original_graph.add_nodes_from(
        ...     ["1", "2", "3", "4", "5", "6", "A", "B", "C"]
        ... )
        >>> original_graph.add_edges_from(
        ...     [
        ...         ("1", "C"), ("1", "B"),
        ...         ("2", "C"), ("2", "B"), ("2", "A"),
        ...         ("3", "B"), ("3", "A"), ("3", "6"),
        ...         ("4", "C"), ("4", "B"), ("4", "A"),
        ...         ("5", "B"), ("5", "A"),
        ...         ("6", "5"),
        ...         ("A", "6")
        ...     ]
        ... )
        >>> c_graph, c_nodes = nx.dedensify(original_graph, threshold=2)
        >>> # re-densifies the compressed graph into the original graph
        >>> for c_node in c_nodes:
        ...     all_neighbors = set(nx.all_neighbors(c_graph, c_node))
        ...     out_neighbors = set(c_graph.neighbors(c_node))
        ...     for out_neighbor in out_neighbors:
        ...         c_graph.remove_edge(c_node, out_neighbor)
        ...     in_neighbors = all_neighbors - out_neighbors
        ...     for in_neighbor in in_neighbors:
        ...         c_graph.remove_edge(in_neighbor, c_node)
        ...         for out_neighbor in out_neighbors:
        ...             c_graph.add_edge(in_neighbor, out_neighbor)
        ...     c_graph.remove_node(c_node)
        ...
        >>> nx.is_isomorphic(original_graph, c_graph)
        True
    References
    ----------
    .. [1] Maccioni, A., & Abadi, D. J. (2016, August).
       Scalable pattern matching over compressed graphs via dedensification.
       In Proceedings of the 22nd ACM SIGKDD International Conference on
       Knowledge Discovery and Data Mining (pp. 1755-1764).
       http://www.cs.umd.edu/~abadi/papers/graph-dedense.pdf
```

```
153: (4)                """
154: (4)                if threshold < 2:
155: (8)                    raise nx.NetworkXError("The degree threshold must be >= 2")
156: (4)                degrees = G.in_degree if G.is_directed() else G.degree
157: (4)                high_degree_nodes = {n for n, d in degrees if d > threshold}
158: (4)                low_degree_nodes = G.nodes() - high_degree_nodes
159: (4)                auxiliary = {}
160: (4)                for node in G:
161: (8)                    high_degree_nbrs = frozenset(high_degree_nodes & set(G[node]))
162: (8)                    if high_degree_nbrs:
163: (12)                       if high_degree_nbrs in auxiliary:
164: (16)                           auxiliary[high_degree_nbrs].add(node)
165: (12)                       else:
166: (16)                           auxiliary[high_degree_nbrs] = {node}
167: (4)                if copy:
168: (8)                    G = G.copy()
169: (4)                compressor_nodes = set()
170: (4)                for index, (high_degree_nodes, low_degree_nodes) in
enumerate(auxiliary.items()):
171: (8)                    low_degree_node_count = len(low_degree_nodes)
172: (8)                    high_degree_node_count = len(high_degree_nodes)
173: (8)                    old_edges = high_degree_node_count * low_degree_node_count
174: (8)                    new_edges = high_degree_node_count + low_degree_node_count
175: (8)                    if old_edges <= new_edges:
176: (12)                       continue
177: (8)                    compression_node = "".join(str(node) for node in high_degree_nodes)
178: (8)                    if prefix:
179: (12)                       compression_node = str(prefix) + compression_node
180: (8)                    for node in low_degree_nodes:
181: (12)                       for high_node in high_degree_nodes:
182: (16)                           if G.has_edge(node, high_node):
183: (20)                               G.remove_edge(node, high_node)
184: (12)                       G.add_edge(node, compression_node)
185: (8)                    for node in high_degree_nodes:
186: (12)                       G.add_edge(compression_node, node)
187: (8)                    compressor_nodes.add(compression_node)
188: (4)                return G, compressor_nodes
189: (0)            def _snap_build_graph(
190: (4)                G,
191: (4)                groups,
192: (4)                node_attributes,
193: (4)                edge_attributes,
194: (4)                neighbor_info,
195: (4)                edge_types,
196: (4)                prefix,
197: (4)                supernode_attribute,
198: (4)                superedge_attribute,
199: (0)            ):
200: (4)                """
201: (4)                Build the summary graph from the data structures produced in the SNAP
aggregation algorithm
202: (4)                Used in the SNAP aggregation algorithm to build the output summary graph
and supernode
203: (4)                lookup dictionary.  This process uses the original graph and the data
structures to
204: (4)                create the supernodes with the correct node attributes, and the superedges
with the correct
205: (4)                edge attributes
206: (4)                Parameters
207: (4)                ----------
208: (4)                G: networkx.Graph
209: (8)                    the original graph to be summarized
210: (4)                groups: dict
211: (8)                    A dictionary of unique group IDs and their corresponding node groups
212: (4)                node_attributes: iterable
213: (8)                    An iterable of the node attributes considered in the summarization
process
214: (4)                edge_attributes: iterable
215: (8)                    An iterable of the edge attributes considered in the summarization
```

```
process
            neighbor_info: dict
                A data structure indicating the number of edges a node has with the
                groups in the current summarization of each edge type
            edge_types: dict
                dictionary of edges in the graph and their corresponding attributes recognized
                in the summarization
            prefix: string
                The prefix to be added to all supernodes
            supernode_attribute: str
                The node attribute for recording the supernode groupings of nodes
            superedge_attribute: str
                The edge attribute for recording the edge types represented by superedges
            Returns
            -------
            summary graph: Networkx graph
            """
            output = G.__class__()
            node_label_lookup = {}
            for index, group_id in enumerate(groups):
                group_set = groups[group_id]
                supernode = f"{prefix}{index}"
                node_label_lookup[group_id] = supernode
                supernode_attributes = {
                    attr: G.nodes[next(iter(group_set))][attr] for attr in node_attributes
                }
                supernode_attributes[supernode_attribute] = group_set
                output.add_node(supernode, **supernode_attributes)
            for group_id in groups:
                group_set = groups[group_id]
                source_supernode = node_label_lookup[group_id]
                for other_group, group_edge_types in neighbor_info[
                    next(iter(group_set))
                ].items():
                    if group_edge_types:
                        target_supernode = node_label_lookup[other_group]
                        summary_graph_edge = (source_supernode, target_supernode)
                        edge_types = [
                            dict(zip(edge_attributes, edge_type))
                            for edge_type in group_edge_types
                        ]
                        has_edge = output.has_edge(*summary_graph_edge)
                        if output.is_multigraph():
                            if not has_edge:
                                for edge_type in edge_types:
                                    output.add_edge(*summary_graph_edge, **edge_type)
                            elif not output.is_directed():
                                existing_edge_data = output.get_edge_data(*summary_graph_edge)
                                for edge_type in edge_types:
                                    if edge_type not in existing_edge_data.values():
                                        output.add_edge(*summary_graph_edge, **edge_type)
                        else:
                            superedge_attributes = {superedge_attribute: edge_types}
                            output.add_edge(*summary_graph_edge, **superedge_attributes)
            return output
        def _snap_eligible_group(G, groups, group_lookup, edge_types):
            """
            Determines if a group is eligible to be split.
            A group is eligible to be split if all nodes in the group have edges of the same type(s)
            with the same other groups.
            Parameters
            ----------
```

```
    G: graph
        graph to be summarized
    groups: dict
        A dictionary of unique group IDs and their corresponding node groups
    group_lookup: dict
        dictionary of nodes and their current corresponding group ID
    edge_types: dict
        dictionary of edges in the graph and their corresponding attributes recognized
        in the summarization
    Returns
    -------
    tuple: group ID to split, and neighbor-groups participation_counts data structure
    """
    nbr_info = {node: {gid: Counter() for gid in groups} for node in group_lookup}
    for group_id in groups:
        current_group = groups[group_id]
        for node in current_group:
            nbr_info[node] = {group_id: Counter() for group_id in groups}
            edges = G.edges(node, keys=True) if G.is_multigraph() else G.edges(node)
            for edge in edges:
                neighbor = edge[1]
                edge_type = edge_types[edge]
                neighbor_group_id = group_lookup[neighbor]
                nbr_info[node][neighbor_group_id][edge_type] += 1
        group_size = len(current_group)
        for other_group_id in groups:
            edge_counts = Counter()
            for node in current_group:
                edge_counts.update(nbr_info[node][other_group_id].keys())
            if not all(count == group_size for count in edge_counts.values()):
                return group_id, nbr_info
    return None, nbr_info
def _snap_split(groups, neighbor_info, group_lookup, group_id):
    """
    Splits a group based on edge types and updates the groups accordingly
    Splits the group with the given group_id based on the edge types
    of the nodes so that each new grouping will all have the same
    edges with other nodes.
    Parameters
    ----------
    groups: dict
        A dictionary of unique group IDs and their corresponding node groups
    neighbor_info: dict
        A data structure indicating the number of edges a node has with the
        groups in the current summarization of each edge type
    edge_types: dict
        dictionary of edges in the graph and their corresponding attributes recognized
        in the summarization
    group_lookup: dict
        dictionary of nodes and their current corresponding group ID
    group_id: object
        ID of group to be split
    Returns
    -------
    dict
        The updated groups based on the split
    """
    new_group_mappings = defaultdict(set)
    for node in groups[group_id]:
        signature = tuple(
            frozenset(edge_types) for edge_types in neighbor_info[node].values()
        )
        new_group_mappings[signature].add(node)
```

```
340: (4)                new_groups = sorted(new_group_mappings.values(), key=len)
341: (4)                for new_group in new_groups[:-1]:
342: (8)                    new_group_id = len(groups)
343: (8)                    groups[new_group_id] = new_group
344: (8)                    groups[group_id] -= new_group
345: (8)                    for node in new_group:
346: (12)                       group_lookup[node] = new_group_id
347: (4)                return groups
348: (0)            @nx._dispatchable(
349: (4)                node_attrs="[node_attributes]", edge_attrs="[edge_attributes]",
returns_graph=True
350: (0)            )
351: (0)            def snap_aggregation(
352: (4)                G,
353: (4)                node_attributes,
354: (4)                edge_attributes=(),
355: (4)                prefix="Supernode-",
356: (4)                supernode_attribute="group",
357: (4)                superedge_attribute="types",
358: (0)            ):
359: (4)                """Creates a summary graph based on attributes and connectivity.
360: (4)                This function uses the Summarization by Grouping Nodes on Attributes
361: (4)                and Pairwise edges (SNAP) algorithm for summarizing a given
362: (4)                graph by grouping nodes by node attributes and their edge attributes
363: (4)                into supernodes in a summary graph.  This name SNAP should not be
364: (4)                confused with the Stanford Network Analysis Project (SNAP).
365: (4)                Here is a high-level view of how this algorithm works:
366: (4)                1) Group nodes by node attribute values.
367: (4)                2) Iteratively split groups until all nodes in each group have edges
368: (4)                to nodes in the same groups. That is, until all the groups are homogeneous
369: (4)                in their member nodes' edges to other groups.  For example,
370: (4)                if all the nodes in group A only have edge to nodes in group B, then the
371: (4)                group is homogeneous and does not need to be split. If all nodes in group
B
372: (4)                have edges with nodes in groups {A, C}, but some also have edges with
other
373: (4)                nodes in B, then group B is not homogeneous and needs to be split into
374: (4)                groups have edges with {A, C} and a group of nodes having
375: (4)                edges with {A, B, C}.  This way, viewers of the summary graph can
376: (4)                assume that all nodes in the group have the exact same node attributes and
377: (4)                the exact same edges.
378: (4)                3) Build the output summary graph, where the groups are represented by
379: (4)                super-nodes. Edges represent the edges shared between all the nodes in
each
380: (4)                respective groups.
381: (4)                A SNAP summary graph can be used to visualize graphs that are too large to
display
382: (4)                or visually analyze, or to efficiently identify sets of similar nodes with
similar connectivity
383: (4)                patterns to other sets of similar nodes based on specified node and/or
edge attributes in a graph.
384: (4)                Parameters
385: (4)                ----------
386: (4)                G: graph
387: (8)                    Networkx Graph to be summarized
388: (4)                node_attributes: iterable, required
389: (8)                    An iterable of the node attributes used to group nodes in the
summarization process. Nodes
390: (8)                    with the same values for these attributes will be grouped together in
the summary graph.
391: (4)                edge_attributes: iterable, optional
392: (8)                    An iterable of the edge attributes considered in the summarization
process.  If provided, unique
393: (8)                    combinations of the attribute values found in the graph are used to
394: (8)                    determine the edge types in the graph.  If not provided, all edges
395: (8)                    are considered to be of the same type.
396: (4)                prefix: str
397: (8)                    The prefix used to denote supernodes in the summary graph. Defaults to
'Supernode-'.
```

```
    supernode_attribute: str
        The node attribute for recording the supernode groupings of nodes. Defaults to 'group'.
    superedge_attribute: str
        The edge attribute for recording the edge types of multiple edges. Defaults to 'types'.
    Returns
    -------
    networkx.Graph: summary graph
    Examples
    --------
    SNAP aggregation takes a graph and summarizes it in the context of user-provided
    node and edge attributes such that a viewer can more easily extract and
    analyze the information represented by the graph
    >>> nodes = {
    ...     "A": dict(color="Red"),
    ...     "B": dict(color="Red"),
    ...     "C": dict(color="Red"),
    ...     "D": dict(color="Red"),
    ...     "E": dict(color="Blue"),
    ...     "F": dict(color="Blue"),
    ... }
    >>> edges = [
    ...     ("A", "E", "Strong"),
    ...     ("B", "F", "Strong"),
    ...     ("C", "E", "Weak"),
    ...     ("D", "F", "Weak"),
    ... ]
    >>> G = nx.Graph()
    >>> for node in nodes:
    ...     attributes = nodes[node]
    ...     G.add_node(node, **attributes)
    >>> for source, target, type in edges:
    ...     G.add_edge(source, target, type=type)
    >>> node_attributes = ("color",)
    >>> edge_attributes = ("type",)
    >>> summary_graph = nx.snap_aggregation(
    ...     G, node_attributes=node_attributes, edge_attributes=edge_attributes
    ... )
    Notes
    -----
    The summary graph produced is called a maximum Attribute-edge
    compatible (AR-compatible) grouping.  According to [1]_, an
    AR-compatible grouping means that all nodes in each group have the same
    exact node attribute values and the same exact edges and
    edge types to one or more nodes in the same groups.  The maximal
    AR-compatible grouping is the grouping with the minimal cardinality.
    The AR-compatible grouping is the most detailed grouping provided by
    any of the SNAP algorithms.
    References
    ----------
    .. [1] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation
       for graph summarization. In Proc. 2008 ACM-SIGMOD Int. Conf.
       Management of Data (SIGMOD’08), pages 567–580, Vancouver, Canada,
       June 2008.
    """
    edge_types = {
        edge: tuple(attrs.get(attr) for attr in edge_attributes)
        for edge, attrs in G.edges.items()
    }
    if not G.is_directed():
        if G.is_multigraph():
            edges = [((v, u, k), etype) for (u, v, k), etype in edge_types.items()]
        else:
            edges = [((v, u), etype) for (u, v), etype in edge_types.items()]
        edge_types.update(edges)
```

```
462: (4)                group_lookup = {
463: (8)                    node: tuple(attrs[attr] for attr in node_attributes)
464: (8)                    for node, attrs in G.nodes.items()
465: (4)                }
466: (4)                groups = defaultdict(set)
467: (4)                for node, node_type in group_lookup.items():
468: (8)                    groups[node_type].add(node)
469: (4)                eligible_group_id, nbr_info = _snap_eligible_group(
470: (8)                    G, groups, group_lookup, edge_types
471: (4)                )
472: (4)                while eligible_group_id:
473: (8)                    groups = _snap_split(groups, nbr_info, group_lookup,
eligible_group_id)
474: (8)                    eligible_group_id, nbr_info = _snap_eligible_group(
475: (12)                       G, groups, group_lookup, edge_types
476: (8)                    )
477: (4)                return _snap_build_graph(
478: (8)                    G,
479: (8)                    groups,
480: (8)                    node_attributes,
481: (8)                    edge_attributes,
482: (8)                    nbr_info,
483: (8)                    edge_types,
484: (8)                    prefix,
485: (8)                    supernode_attribute,
486: (8)                    superedge_attribute,
487: (4)                )
```

File 55 - swap.py:

```
1: (0)              """Swap edges in a graph.
2: (0)              """
3: (0)              import math
4: (0)              import networkx as nx
5: (0)              from networkx.utils import py_random_state
6: (0)              __all__ = ["double_edge_swap", "connected_double_edge_swap",
"directed_edge_swap"]
7: (0)              @nx.utils.not_implemented_for("undirected")
8: (0)              @py_random_state(3)
9: (0)              @nx._dispatchable(mutates_input=True, returns_graph=True)
10: (0)             def directed_edge_swap(G, *, nswap=1, max_tries=100, seed=None):
11: (4)                 """Swap three edges in a directed graph while keeping the node degrees
fixed.
12: (4)                 A directed edge swap swaps three edges such that a -> b -> c -> d becomes
13: (4)                 a -> c -> b -> d. This pattern of swapping allows all possible states with
the
14: (4)                 same in- and out-degree distribution in a directed graph to be reached.
15: (4)                 If the swap would create parallel edges (e.g. if a -> c already existed in
the
16: (4)                 previous example), another attempt is made to find a suitable trio of
edges.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : DiGraph
20: (7)                    A directed graph
21: (4)                 nswap : integer (optional, default=1)
22: (7)                    Number of three-edge (directed) swaps to perform
23: (4)                 max_tries : integer (optional, default=100)
24: (7)                    Maximum number of attempts to swap edges
25: (4)                 seed : integer, random_state, or None (default)
26: (8)                     Indicator of random number generation state.
27: (8)                     See :ref:`Randomness<randomness>`.
28: (4)                 Returns
29: (4)                 -------
30: (4)                 G : DiGraph
31: (7)                    The graph after the edges are swapped.
32: (4)                 Raises
```

```
(4)                 ------
(4)                 NetworkXError
(8)                     If `G` is not directed, or
(8)                     If nswap > max_tries, or
(8)                     If there are fewer than 4 nodes or 3 edges in `G`.
(4)                 NetworkXAlgorithmError
(8)                     If the number of swap attempts exceeds `max_tries` before `nswap`
swaps are made
(4)                 Notes
(4)                 -----
(4)                 Does not enforce any connectivity constraints.
(4)                 The graph G is modified in place.
(4)                 A later swap is allowed to undo a previous swap.
(4)                 References
(4)                 ----------
(4)                 .. [1] Erdős, Péter L., et al. “A Simple Havel-Hakimi Type Algorithm to
Realize
(11)                       Graphical Degree Sequences of Directed Graphs.” ArXiv:0905.4913
[Math],
(11)                       Jan. 2010. https://doi.org/10.48550/arXiv.0905.4913.
(11)                       Published  2010 in Elec. J. Combinatorics (17(1)). R66.
(11)                       http://www.combinatorics.org/Volume_17/PDF/v17i1r66.pdf
(4)                 .. [2] “Combinatorics - Reaching All Possible Simple Directed Graphs with
a given
(11)                       Degree Sequence with 2-Edge Swaps.” Mathematics Stack Exchange,
(11)                       https://math.stackexchange.com/questions/22272/. Accessed 30 May
2022.
(4)                 """
(4)                 if nswap > max_tries:
(8)                     raise nx.NetworkXError("Number of swaps > number of tries allowed.")
(4)                 if len(G) < 4:
(8)                     raise nx.NetworkXError("DiGraph has fewer than four nodes.")
(4)                 if len(G.edges) < 3:
(8)                     raise nx.NetworkXError("DiGraph has fewer than 3 edges")
(4)                 tries = 0
(4)                 swapcount = 0
(4)                 keys, degrees = zip(*G.degree())  # keys, degree
(4)                 cdf = nx.utils.cumulative_distribution(degrees)  # cdf of degree
(4)                 discrete_sequence = nx.utils.discrete_sequence
(4)                 while swapcount < nswap:
(8)                     start_index = discrete_sequence(1, cdistribution=cdf, seed=seed)[0]
(8)                     start = keys[start_index]
(8)                     tries += 1
(8)                     if tries > max_tries:
(12)                        msg = f"Maximum number of swap attempts ({tries}) exceeded before
desired swaps achieved ({nswap})."
(12)                        raise nx.NetworkXAlgorithmError(msg)
(8)                     if G.out_degree(start) == 0:
(12)                        continue
(8)                     second = seed.choice(list(G.succ[start]))
(8)                     if start == second:
(12)                        continue
(8)                     if G.out_degree(second) == 0:
(12)                        continue
(8)                     third = seed.choice(list(G.succ[second]))
(8)                     if second == third:
(12)                        continue
(8)                     if G.out_degree(third) == 0:
(12)                        continue
(8)                     fourth = seed.choice(list(G.succ[third]))
(8)                     if third == fourth:
(12)                        continue
(8)                     if (
(12)                        third not in G.succ[start]
(12)                        and fourth not in G.succ[second]
(12)                        and second not in G.succ[third]
(8)                     ):
(12)                        G.add_edge(start, third)
(12)                        G.add_edge(third, second)
```

```
96: (12)                        G.add_edge(second, fourth)
97: (12)                        G.remove_edge(start, second)
98: (12)                        G.remove_edge(second, third)
99: (12)                        G.remove_edge(third, fourth)
100: (12)                       swapcount += 1
101: (4)                return G
102: (0)            @py_random_state(3)
103: (0)            @nx._dispatchable(mutates_input=True, returns_graph=True)
104: (0)            def double_edge_swap(G, nswap=1, max_tries=100, seed=None):
105: (4)                """Swap two edges in the graph while keeping the node degrees fixed.
106: (4)                A double-edge swap removes two randomly chosen edges u-v and x-y
107: (4)                and creates the new edges u-x and v-y::
108: (5)                 u--v            u  v
109: (12)                       becomes  |  |
110: (5)                 x--y            x  y
111: (4)                If either the edge u-x or v-y already exist no swap is performed
112: (4)                and another attempt is made to find a suitable edge pair.
113: (4)                Parameters
114: (4)                ----------
115: (4)                G : graph
116: (7)                   An undirected graph
117: (4)                nswap : integer (optional, default=1)
118: (7)                   Number of double-edge swaps to perform
119: (4)                max_tries : integer (optional)
120: (7)                   Maximum number of attempts to swap edges
121: (4)                seed : integer, random_state, or None (default)
122: (8)                    Indicator of random number generation state.
123: (8)                    See :ref:`Randomness<randomness>`.
124: (4)                Returns
125: (4)                -------
126: (4)                G : graph
127: (7)                   The graph after double edge swaps.
128: (4)                Raises
129: (4)                ------
130: (4)                NetworkXError
131: (8)                    If `G` is directed, or
132: (8)                    If `nswap` > `max_tries`, or
133: (8)                    If there are fewer than 4 nodes or 2 edges in `G`.
134: (4)                NetworkXAlgorithmError
135: (8)                    If the number of swap attempts exceeds `max_tries` before `nswap`
swaps are made
136: (4)                Notes
137: (4)                -----
138: (4)                Does not enforce any connectivity constraints.
139: (4)                The graph G is modified in place.
140: (4)                """
141: (4)                if G.is_directed():
142: (8)                    raise nx.NetworkXError(
143: (12)                       "double_edge_swap() not defined for directed graphs. Use
directed_edge_swap instead."
144: (8)                    )
145: (4)                if nswap > max_tries:
146: (8)                    raise nx.NetworkXError("Number of swaps > number of tries allowed.")
147: (4)                if len(G) < 4:
148: (8)                    raise nx.NetworkXError("Graph has fewer than four nodes.")
149: (4)                if len(G.edges) < 2:
150: (8)                    raise nx.NetworkXError("Graph has fewer than 2 edges")
151: (4)                n = 0
152: (4)                swapcount = 0
153: (4)                keys, degrees = zip(*G.degree())  # keys, degree
154: (4)                cdf = nx.utils.cumulative_distribution(degrees)  # cdf of degree
155: (4)                discrete_sequence = nx.utils.discrete_sequence
156: (4)                while swapcount < nswap:
157: (8)                    (ui, xi) = discrete_sequence(2, cdistribution=cdf, seed=seed)
158: (8)                    if ui == xi:
159: (12)                       continue  # same source, skip
160: (8)                    u = keys[ui]  # convert index to label
161: (8)                    x = keys[xi]
162: (8)                    v = seed.choice(list(G[u]))
```

```
(8)                     y = seed.choice(list(G[x]))
(8)                     if v == y:
(12)                        continue  # same target, skip
(8)                     if (x not in G[u]) and (y not in G[v]):  # don't create parallel edges
(12)                        G.add_edge(u, x)
(12)                        G.add_edge(v, y)
(12)                        G.remove_edge(u, v)
(12)                        G.remove_edge(x, y)
(12)                        swapcount += 1
(8)                     if n >= max_tries:
(12)                        e = (
(16)                            f"Maximum number of swap attempts ({n}) exceeded "
(16)                            f"before desired swaps achieved ({nswap})."
(12)                        )
(12)                        raise nx.NetworkXAlgorithmError(e)
(8)                     n += 1
(4)                 return G
(0)             @py_random_state(3)
(0)             @nx._dispatchable(mutates_input=True)
(0)             def connected_double_edge_swap(G, nswap=1, _window_threshold=3, seed=None):
(4)                 """Attempts the specified number of double-edge swaps in the graph `G`.
(4)                 A double-edge swap removes two randomly chosen edges `(u, v)` and `(x,
(4)                 y)` and creates the new edges `(u, x)` and `(v, y)`::
(5)                  u--v            u  v
(12)                        becomes  |  |
(5)                  x--y            x  y
(4)                 If either `(u, x)` or `(v, y)` already exist, then no swap is performed
(4)                 so the actual number of swapped edges is always *at most* `nswap`.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    An undirected graph
(4)                 nswap : integer (optional, default=1)
(7)                    Number of double-edge swaps to perform
(4)                 _window_threshold : integer
(7)                    The window size below which connectedness of the graph will be checked
(7)                    after each swap.
(7)                    The "window" in this function is a dynamically updated integer that
(7)                    represents the number of swap attempts to make before checking if the
(7)                    graph remains connected. It is an optimization used to decrease the
(7)                    running time of the algorithm in exchange for increased complexity of
(7)                    implementation.
(7)                    If the window size is below this threshold, then the algorithm checks
(7)                    after each swap if the graph remains connected by checking if there is
a
(7)                    path joining the two nodes whose edge was just removed. If the window
(7)                    size is above this threshold, then the algorithm performs do all the
(7)                    swaps in the window and only then check if the graph is still
connected.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 int
(7)                    The number of successful swaps
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(7)                    If the input graph is not connected, or if the graph has fewer than
four
(7)                    nodes.
(4)                 Notes
(4)                 -----
(4)                 The initial graph `G` must be connected, and the resulting graph is
(4)                 connected. The graph `G` is modified in place.
(4)                 References
(4)                 ----------
(4)                 .. [1] C. Gkantsidis and M. Mihail and E. Zegura,
```

```
(11)                        The Markov chain simulation method for generating connected
(11)                        power law random graphs, 2003.
(11)                        http://citeseer.ist.psu.edu/gkantsidis03markov.html
(4)                 """
(4)                 if not nx.is_connected(G):
(8)                     raise nx.NetworkXError("Graph not connected")
(4)                 if len(G) < 4:
(8)                     raise nx.NetworkXError("Graph has fewer than four nodes.")
(4)                 n = 0
(4)                 swapcount = 0
(4)                 deg = G.degree()
(4)                 dk = [n for n, d in G.degree()]
(4)                 cdf = nx.utils.cumulative_distribution([d for n, d in G.degree()])
(4)                 discrete_sequence = nx.utils.discrete_sequence
(4)                 window = 1
(4)                 while n < nswap:
(8)                     wcount = 0
(8)                     swapped = []
(8)                     if window < _window_threshold:
(12)                        fail = False
(12)                        while wcount < window and n < nswap:
(16)                            (ui, xi) = discrete_sequence(2, cdistribution=cdf, seed=seed)
(16)                            if ui == xi:
(20)                                continue
(16)                            u = dk[ui]
(16)                            x = dk[xi]
(16)                            v = seed.choice(list(G.neighbors(u)))
(16)                            y = seed.choice(list(G.neighbors(x)))
(16)                            if v == y:
(20)                                continue
(16)                            if x not in G[u] and y not in G[v]:
(20)                                G.remove_edge(u, v)
(20)                                G.remove_edge(x, y)
(20)                                G.add_edge(u, x)
(20)                                G.add_edge(v, y)
(20)                                swapped.append((u, v, x, y))
(20)                                swapcount += 1
(16)                            n += 1
(16)                            if nx.has_path(G, u, v):
(20)                                wcount += 1
(16)                            else:
(20)                                G.add_edge(u, v)
(20)                                G.add_edge(x, y)
(20)                                G.remove_edge(u, x)
(20)                                G.remove_edge(v, y)
(20)                                swapcount -= 1
(20)                                fail = True
(12)                        if fail:
(16)                            window = math.ceil(window / 2)
(12)                        else:
(16)                            window += 1
(8)                     else:
(12)                        while wcount < window and n < nswap:
(16)                            (ui, xi) = discrete_sequence(2, cdistribution=cdf, seed=seed)
(16)                            if ui == xi:
(20)                                continue
(16)                            u = dk[ui]
(16)                            x = dk[xi]
(16)                            v = seed.choice(list(G.neighbors(u)))
(16)                            y = seed.choice(list(G.neighbors(x)))
(16)                            if v == y:
(20)                                continue
(16)                            if x not in G[u] and y not in G[v]:
(20)                                G.remove_edge(u, v)
(20)                                G.remove_edge(x, y)
(20)                                G.add_edge(u, x)
(20)                                G.add_edge(v, y)
(20)                                swapped.append((u, v, x, y))
(20)                                swapcount += 1
```

```
298: (16)                            n += 1
299: (16)                            wcount += 1
300: (12)                        if nx.is_connected(G):
301: (16)                            window += 1
302: (12)                        else:
303: (16)                            while swapped:
304: (20)                                (u, v, x, y) = swapped.pop()
305: (20)                                G.add_edge(u, v)
306: (20)                                G.add_edge(x, y)
307: (20)                                G.remove_edge(u, x)
308: (20)                                G.remove_edge(v, y)
309: (20)                                swapcount -= 1
310: (16)                            window = math.ceil(window / 2)
311: (4)                 return swapcount
```

File 56 - threshold.py:

```
1: (0)               """
2: (0)               Threshold Graphs - Creation, manipulation and identification.
3: (0)               """
4: (0)               from math import sqrt
5: (0)               import networkx as nx
6: (0)               from networkx.utils import py_random_state
7: (0)               __all__ = ["is_threshold_graph", "find_threshold_graph"]
8: (0)               @nx._dispatchable
9: (0)               def is_threshold_graph(G):
10: (4)                  """
11: (4)                  Returns `True` if `G` is a threshold graph.
12: (4)                  Parameters
13: (4)                  ----------
14: (4)                  G : NetworkX graph instance
15: (8)                      An instance of `Graph`, `DiGraph`, `MultiGraph` or `MultiDiGraph`
16: (4)                  Returns
17: (4)                  -------
18: (4)                  bool
19: (8)                      `True` if `G` is a threshold graph, `False` otherwise.
20: (4)                  Examples
21: (4)                  --------
22: (4)                  >>> from networkx.algorithms.threshold import is_threshold_graph
23: (4)                  >>> G = nx.path_graph(3)
24: (4)                  >>> is_threshold_graph(G)
25: (4)                  True
26: (4)                  >>> G = nx.barbell_graph(3, 3)
27: (4)                  >>> is_threshold_graph(G)
28: (4)                  False
29: (4)                  References
30: (4)                  ----------
31: (4)                  .. [1] Threshold graphs: https://en.wikipedia.org/wiki/Threshold_graph
32: (4)                  """
33: (4)                  return is_threshold_sequence([d for n, d in G.degree()])
34: (0)               def is_threshold_sequence(degree_sequence):
35: (4)                  """
36: (4)                  Returns True if the sequence is a threshold degree sequence.
37: (4)                  Uses the property that a threshold graph must be constructed by
38: (4)                  adding either dominating or isolated nodes. Thus, it can be
39: (4)                  deconstructed iteratively by removing a node of degree zero or a
40: (4)                  node that connects to the remaining nodes.  If this deconstruction
41: (4)                  fails then the sequence is not a threshold sequence.
42: (4)                  """
43: (4)                  ds = degree_sequence[:]  # get a copy so we don't destroy original
44: (4)                  ds.sort()
45: (4)                  while ds:
46: (8)                      if ds[0] == 0:  # if isolated node
47: (12)                         ds.pop(0)  # remove it
48: (12)                         continue
49: (8)                      if ds[-1] != len(ds) - 1:  # is the largest degree node dominating?
50: (12)                         return False  # no, not a threshold degree sequence
```

```
51: (8)                     ds.pop()  # yes, largest is the dominating node
52: (8)                     ds = [d - 1 for d in ds]  # remove it and decrement all degrees
53: (4)                 return True
54: (0)             def creation_sequence(degree_sequence, with_labels=False, compact=False):
55: (4)                 """
56: (4)                 Determines the creation sequence for the given threshold degree sequence.
57: (4)                 The creation sequence is a list of single characters 'd'
58: (4)                 or 'i': 'd' for dominating or 'i' for isolated vertices.
59: (4)                 Dominating vertices are connected to all vertices present when it
60: (4)                 is added.  The first node added is by convention 'd'.
61: (4)                 This list can be converted to a string if desired using "".join(cs)
62: (4)                 If with_labels==True:
63: (4)                 Returns a list of 2-tuples containing the vertex number
64: (4)                 and a character 'd' or 'i' which describes the type of vertex.
65: (4)                 If compact==True:
66: (4)                 Returns the creation sequence in a compact form that is the number
67: (4)                 of 'i's and 'd's alternating.
68: (4)                 Examples:
69: (4)                 [1,2,2,3] represents d,i,i,d,d,i,i,i
70: (4)                 [3,1,2] represents d,d,d,i,d,d
71: (4)                 Notice that the first number is the first vertex to be used for
72: (4)                 construction and so is always 'd'.
73: (4)                 with_labels and compact cannot both be True.
74: (4)                 Returns None if the sequence is not a threshold sequence
75: (4)                 """
76: (4)                 if with_labels and compact:
77: (8)                     raise ValueError("compact sequences cannot be labeled")
78: (4)                 if isinstance(degree_sequence, dict):  # labeled degree sequence
79: (8)                     ds = [[degree, label] for (label, degree) in degree_sequence.items()]
80: (4)                 else:
81: (8)                     ds = [[d, i] for i, d in enumerate(degree_sequence)]
82: (4)                 ds.sort()
83: (4)                 cs = []  # creation sequence
84: (4)                 while ds:
85: (8)                     if ds[0][0] == 0:  # isolated node
86: (12)                        (d, v) = ds.pop(0)
87: (12)                        if len(ds) > 0:  # make sure we start with a d
88: (16)                            cs.insert(0, (v, "i"))
89: (12)                        else:
90: (16)                            cs.insert(0, (v, "d"))
91: (12)                        continue
92: (8)                     if ds[-1][0] != len(ds) - 1:  # Not dominating node
93: (12)                        return None  # not a threshold degree sequence
94: (8)                     (d, v) = ds.pop()
95: (8)                     cs.insert(0, (v, "d"))
96: (8)                     ds = [[d[0] - 1, d[1]] for d in ds]  # decrement due to removing node
97: (4)                 if with_labels:
98: (8)                     return cs
99: (4)                 if compact:
100: (8)                    return make_compact(cs)
101: (4)                return [v[1] for v in cs]  # not labeled
102: (0)            def make_compact(creation_sequence):
103: (4)                """
104: (4)                Returns the creation sequence in a compact form
105: (4)                that is the number of 'i's and 'd's alternating.
106: (4)                Examples
107: (4)                --------
108: (4)                >>> from networkx.algorithms.threshold import make_compact
109: (4)                >>> make_compact(["d", "i", "i", "d", "d", "i", "i", "i"])
110: (4)                [1, 2, 2, 3]
111: (4)                >>> make_compact(["d", "d", "d", "i", "d", "d"])
112: (4)                [3, 1, 2]
113: (4)                Notice that the first number is the first vertex
114: (4)                to be used for construction and so is always 'd'.
115: (4)                Labeled creation sequences lose their labels in the
116: (4)                compact representation.
117: (4)                >>> make_compact([3, 1, 2])
118: (4)                [3, 1, 2]
119: (4)                """
```

```
(4)                 first = creation_sequence[0]
(4)                 if isinstance(first, str):  # creation sequence
(8)                     cs = creation_sequence[:]
(4)                 elif isinstance(first, tuple):  # labeled creation sequence
(8)                     cs = [s[1] for s in creation_sequence]
(4)                 elif isinstance(first, int):  # compact creation sequence
(8)                     return creation_sequence
(4)                 else:
(8)                     raise TypeError("Not a valid creation sequence type")
(4)                 ccs = []
(4)                 count = 1  # count the run lengths of d's or i's.
(4)                 for i in range(1, len(cs)):
(8)                     if cs[i] == cs[i - 1]:
(12)                        count += 1
(8)                     else:
(12)                        ccs.append(count)
(12)                        count = 1
(4)                 ccs.append(count)  # don't forget the last one
(4)                 return ccs
(0)             def uncompact(creation_sequence):
(4)                 """
(4)                 Converts a compact creation sequence for a threshold
(4)                 graph to a standard creation sequence (unlabeled).
(4)                 If the creation_sequence is already standard, return it.
(4)                 See creation_sequence.
(4)                 """
(4)                 first = creation_sequence[0]
(4)                 if isinstance(first, str):  # creation sequence
(8)                     return creation_sequence
(4)                 elif isinstance(first, tuple):  # labeled creation sequence
(8)                     return creation_sequence
(4)                 elif isinstance(first, int):  # compact creation sequence
(8)                     ccscopy = creation_sequence[:]
(4)                 else:
(8)                     raise TypeError("Not a valid creation sequence type")
(4)                 cs = []
(4)                 while ccscopy:
(8)                     cs.extend(ccscopy.pop(0) * ["d"])
(8)                     if ccscopy:
(12)                        cs.extend(ccscopy.pop(0) * ["i"])
(4)                 return cs
(0)             def creation_sequence_to_weights(creation_sequence):
(4)                 """
(4)                 Returns a list of node weights which create the threshold
(4)                 graph designated by the creation sequence.  The weights
(4)                 are scaled so that the threshold is 1.0.  The order of the
(4)                 nodes is the same as that in the creation sequence.
(4)                 """
(4)                 first = creation_sequence[0]
(4)                 if isinstance(first, str):  # creation sequence
(8)                     if isinstance(creation_sequence, list):
(12)                        wseq = creation_sequence[:]
(8)                     else:
(12)                        wseq = list(creation_sequence)  # string like 'ddididid'
(4)                 elif isinstance(first, tuple):  # labeled creation sequence
(8)                     wseq = [v[1] for v in creation_sequence]
(4)                 elif isinstance(first, int):  # compact creation sequence
(8)                     wseq = uncompact(creation_sequence)
(4)                 else:
(8)                     raise TypeError("Not a valid creation sequence type")
(4)                 wseq.reverse()
(4)                 w = 0
(4)                 prev = "i"
(4)                 for j, s in enumerate(wseq):
(8)                     if s == "i":
(12)                        wseq[j] = w
(12)                        prev = s
(8)                     elif prev == "i":
(12)                        prev = s
```

```
(12)                    w += 1
(4)                wseq.reverse()  # now pass through forwards
(4)                for j, s in enumerate(wseq):
(8)                    if s == "d":
(12)                       wseq[j] = w
(12)                       prev = s
(8)                    elif prev == "d":
(12)                       prev = s
(12)                       w += 1
(4)                if prev == "d":
(8)                    w += 1
(4)                wscale = 1 / w
(4)                return [ww * wscale for ww in wseq]
(0)            def weights_to_creation_sequence(
(4)                weights, threshold=1, with_labels=False, compact=False
(0)            ):
(4)                """
(4)                Returns a creation sequence for a threshold graph
(4)                determined by the weights and threshold given as input.
(4)                If the sum of two node weights is greater than the
(4)                threshold value, an edge is created between these nodes.
(4)                The creation sequence is a list of single characters 'd'
(4)                or 'i': 'd' for dominating or 'i' for isolated vertices.
(4)                Dominating vertices are connected to all vertices present
(4)                when it is added.  The first node added is by convention 'd'.
(4)                If with_labels==True:
(4)                Returns a list of 2-tuples containing the vertex number
(4)                and a character 'd' or 'i' which describes the type of vertex.
(4)                If compact==True:
(4)                Returns the creation sequence in a compact form that is the number
(4)                of 'i's and 'd's alternating.
(4)                Examples:
(4)                [1,2,2,3] represents d,i,i,d,d,i,i,i
(4)                [3,1,2] represents d,d,d,i,d,d
(4)                Notice that the first number is the first vertex to be used for
(4)                construction and so is always 'd'.
(4)                with_labels and compact cannot both be True.
(4)                """
(4)                if with_labels and compact:
(8)                    raise ValueError("compact sequences cannot be labeled")
(4)                if isinstance(weights, dict):  # labeled weights
(8)                    wseq = [[w, label] for (label, w) in weights.items()]
(4)                else:
(8)                    wseq = [[w, i] for i, w in enumerate(weights)]
(4)                wseq.sort()
(4)                cs = []  # creation sequence
(4)                cutoff = threshold - wseq[-1][0]
(4)                while wseq:
(8)                    if wseq[0][0] < cutoff:  # isolated node
(12)                       (w, label) = wseq.pop(0)
(12)                       cs.append((label, "i"))
(8)                    else:
(12)                       (w, label) = wseq.pop()
(12)                       cs.append((label, "d"))
(12)                       cutoff = threshold - wseq[-1][0]
(8)                    if len(wseq) == 1:  # make sure we start with a d
(12)                       (w, label) = wseq.pop()
(12)                       cs.append((label, "d"))
(4)                cs.reverse()
(4)                if with_labels:
(8)                    return cs
(4)                if compact:
(8)                    return make_compact(cs)
(4)                return [v[1] for v in cs]  # not labeled
(0)            @nx._dispatchable(graphs=None, returns_graph=True)
(0)            def threshold_graph(creation_sequence, create_using=None):
(4)                """
(4)                Create a threshold graph from the creation sequence or compact
(4)                creation_sequence.
```

```
(4)                 The input sequence can be a
(4)                 creation sequence (e.g. ['d','i','d','d','d','i'])
(4)                 labeled creation sequence (e.g. [(0,'d'),(2,'d'),(1,'i')])
(4)                 compact creation sequence (e.g. [2,1,1,2,0])
(4)                 Use cs=creation_sequence(degree_sequence,labeled=True)
(4)                 to convert a degree sequence to a creation sequence.
(4)                 Returns None if the sequence is not valid
(4)                 """
(4)                 first = creation_sequence[0]
(4)                 if isinstance(first, str):  # creation sequence
(8)                     ci = list(enumerate(creation_sequence))
(4)                 elif isinstance(first, tuple):  # labeled creation sequence
(8)                     ci = creation_sequence[:]
(4)                 elif isinstance(first, int):  # compact creation sequence
(8)                     cs = uncompact(creation_sequence)
(8)                     ci = list(enumerate(cs))
(4)                 else:
(8)                     print("not a valid creation sequence type")
(8)                     return None
(4)                 G = nx.empty_graph(0, create_using)
(4)                 if G.is_directed():
(8)                     raise nx.NetworkXError("Directed Graph not supported")
(4)                 G.name = "Threshold Graph"
(4)                 while ci:
(8)                     (v, node_type) = ci.pop(0)
(8)                     if node_type == "d":  # dominating type, connect to all existing nodes
(12)                        for u in list(G):
(16)                            G.add_edge(v, u)
(8)                     G.add_node(v)
(4)                 return G
(0)             @nx._dispatchable
(0)             def find_alternating_4_cycle(G):
(4)                 """
(4)                 Returns False if there aren't any alternating 4 cycles.
(4)                 Otherwise returns the cycle as [a,b,c,d] where (a,b)
(4)                 and (c,d) are edges and (a,c) and (b,d) are not.
(4)                 """
(4)                 for u, v in G.edges():
(8)                     for w in G.nodes():
(12)                        if not G.has_edge(u, w) and u != w:
(16)                            for x in G.neighbors(w):
(20)                                if not G.has_edge(v, x) and v != x:
(24)                                    return [u, v, w, x]
(4)                 return False
(0)             @nx._dispatchable(returns_graph=True)
(0)             def find_threshold_graph(G, create_using=None):
(4)                 """
(4)                 Returns a threshold subgraph that is close to largest in `G`.
(4)                 The threshold graph will contain the largest degree node in G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph instance
(8)                     An instance of `Graph`, or `MultiDiGraph`
(4)                 create_using : NetworkX graph class or `None` (default), optional
(8)                     Type of graph to use when constructing the threshold graph.
(8)                     If `None`, infer the appropriate graph type from the input.
(4)                 Returns
(4)                 -------
(4)                 graph :
(8)                     A graph instance representing the threshold graph
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms.threshold import find_threshold_graph
(4)                 >>> G = nx.barbell_graph(3, 3)
(4)                 >>> T = find_threshold_graph(G)
(4)                 >>> T.nodes  # may vary
(4)                 NodeView((7, 8, 5, 6))
(4)                 References
(4)                 ----------
```

```
327: (4)                .. [1] Threshold graphs: https://en.wikipedia.org/wiki/Threshold_graph
328: (4)                """
329: (4)                return threshold_graph(find_creation_sequence(G), create_using)
330: (0)            @nx._dispatchable
331: (0)            def find_creation_sequence(G):
332: (4)                """
333: (4)                Find a threshold subgraph that is close to largest in G.
334: (4)                Returns the labeled creation sequence of that threshold graph.
335: (4)                """
336: (4)                cs = []
337: (4)                H = G
338: (4)                while H.order() > 0:
339: (8)                    dsdict = dict(H.degree())
340: (8)                    ds = [(d, v) for v, d in dsdict.items()]
341: (8)                    ds.sort()
342: (8)                    if ds[-1][0] == 0:  # all are isolated
343: (12)                       cs.extend(zip(dsdict, ["i"] * (len(ds) - 1) + ["d"]))
344: (12)                       break  # Done!
345: (8)                    while ds[0][0] == 0:
346: (12)                       (d, iso) = ds.pop(0)
347: (12)                       cs.append((iso, "i"))
348: (8)                    (d, bigv) = ds.pop()
349: (8)                    cs.append((bigv, "d"))
350: (8)                    H = H.subgraph(H.neighbors(bigv))
351: (4)                cs.reverse()
352: (4)                return cs
353: (0)            def triangles(creation_sequence):
354: (4)                """
355: (4)                Compute number of triangles in the threshold graph with the
356: (4)                given creation sequence.
357: (4)                """
358: (4)                cs = creation_sequence  # alias
359: (4)                dr = cs.count("d")  # number of d's in sequence
360: (4)                ntri = dr * (dr - 1) * (dr - 2) / 6  # number of triangles in clique of nd
d's
361: (4)                for i, typ in enumerate(cs):
362: (8)                    if typ == "i":
363: (12)                       ntri += dr * (dr - 1) / 2
364: (8)                    else:
365: (12)                       dr -= 1
366: (4)                return ntri
367: (0)            def triangle_sequence(creation_sequence):
368: (4)                """
369: (4)                Return triangle sequence for the given threshold graph creation sequence.
370: (4)                """
371: (4)                cs = creation_sequence
372: (4)                seq = []
373: (4)                dr = cs.count("d")  # number of d's to the right of the current pos
374: (4)                dcur = (dr - 1) * (dr - 2) // 2  # number of triangles through a node of
clique dr
375: (4)                irun = 0  # number of i's in the last run
376: (4)                drun = 0  # number of d's in the last run
377: (4)                for i, sym in enumerate(cs):
378: (8)                    if sym == "d":
379: (12)                       drun += 1
380: (12)                       tri = dcur + (dr - 1) * irun  # new triangles at this d
381: (8)                    else:  # cs[i]="i":
382: (12)                       if prevsym == "d":  # new string of i's
383: (16)                           dcur += (dr - 1) * irun  # accumulate shared shortest paths
384: (16)                           irun = 0  # reset i run counter
385: (16)                           dr -= drun  # reduce number of d's to right
386: (16)                           drun = 0  # reset d run counter
387: (12)                       irun += 1
388: (12)                       tri = dr * (dr - 1) // 2  # new triangles at this i
389: (8)                    seq.append(tri)
390: (8)                    prevsym = sym
391: (4)                return seq
392: (0)            def cluster_sequence(creation_sequence):
393: (4)                """
```

```
(4)                 Return cluster sequence for the given threshold graph creation sequence.
(4)                 """
(4)                 triseq = triangle_sequence(creation_sequence)
(4)                 degseq = degree_sequence(creation_sequence)
(4)                 cseq = []
(4)                 for i, deg in enumerate(degseq):
(8)                     tri = triseq[i]
(8)                     if deg <= 1:  # isolated vertex or single pair gets cc 0
(12)                        cseq.append(0)
(12)                        continue
(8)                     max_size = (deg * (deg - 1)) // 2
(8)                     cseq.append(tri / max_size)
(4)                 return cseq
(0)             def degree_sequence(creation_sequence):
(4)                 """
(4)                 Return degree sequence for the threshold graph with the given
(4)                 creation sequence
(4)                 """
(4)                 cs = creation_sequence  # alias
(4)                 seq = []
(4)                 rd = cs.count("d")  # number of d to the right
(4)                 for i, sym in enumerate(cs):
(8)                     if sym == "d":
(12)                        rd -= 1
(12)                        seq.append(rd + i)
(8)                     else:
(12)                        seq.append(rd)
(4)                 return seq
(0)             def density(creation_sequence):
(4)                 """
(4)                 Return the density of the graph with this creation_sequence.
(4)                 The density is the fraction of possible edges present.
(4)                 """
(4)                 N = len(creation_sequence)
(4)                 two_size = sum(degree_sequence(creation_sequence))
(4)                 two_possible = N * (N - 1)
(4)                 den = two_size / two_possible
(4)                 return den
(0)             def degree_correlation(creation_sequence):
(4)                 """
(4)                 Return the degree-degree correlation over all edges.
(4)                 """
(4)                 cs = creation_sequence
(4)                 s1 = 0  # deg_i*deg_j
(4)                 s2 = 0  # deg_i^2+deg_j^2
(4)                 s3 = 0  # deg_i+deg_j
(4)                 m = 0  # number of edges
(4)                 rd = cs.count("d")  # number of d nodes to the right
(4)                 rdi = [i for i, sym in enumerate(cs) if sym == "d"]  # index of "d"s
(4)                 ds = degree_sequence(cs)
(4)                 for i, sym in enumerate(cs):
(8)                     if sym == "d":
(12)                        if i != rdi[0]:
(16)                            print("Logic error in degree_correlation", i, rdi)
(16)                            raise ValueError
(12)                        rdi.pop(0)
(8)                     degi = ds[i]
(8)                     for dj in rdi:
(12)                        degj = ds[dj]
(12)                        s1 += degj * degi
(12)                        s2 += degi**2 + degj**2
(12)                        s3 += degi + degj
(12)                        m += 1
(4)                 denom = 2 * m * s2 - s3 * s3
(4)                 numer = 4 * m * s1 - s3 * s3
(4)                 if denom == 0:
(8)                     if numer == 0:
(12)                        return 1
(8)                     raise ValueError(f"Zero Denominator but Numerator is {numer}")
```

```
463: (4)                return numer / denom
464: (0)            def shortest_path(creation_sequence, u, v):
465: (4)                """
466: (4)                Find the shortest path between u and v in a
467: (4)                threshold graph G with the given creation_sequence.
468: (4)                For an unlabeled creation_sequence, the vertices
469: (4)                u and v must be integers in (0,len(sequence)) referring
470: (4)                to the position of the desired vertices in the sequence.
471: (4)                For a labeled creation_sequence, u and v are labels of vertices.
472: (4)                Use cs=creation_sequence(degree_sequence,with_labels=True)
473: (4)                to convert a degree sequence to a creation sequence.
474: (4)                Returns a list of vertices from u to v.
475: (4)                Example: if they are neighbors, it returns [u,v]
476: (4)                """
477: (4)                first = creation_sequence[0]
478: (4)                if isinstance(first, str):  # creation sequence
479: (8)                    cs = [(i, creation_sequence[i]) for i in
range(len(creation_sequence))]
480: (4)                elif isinstance(first, tuple):  # labeled creation sequence
481: (8)                    cs = creation_sequence[:]
482: (4)                elif isinstance(first, int):  # compact creation sequence
483: (8)                    ci = uncompact(creation_sequence)
484: (8)                    cs = [(i, ci[i]) for i in range(len(ci))]
485: (4)                else:
486: (8)                    raise TypeError("Not a valid creation sequence type")
487: (4)                verts = [s[0] for s in cs]
488: (4)                if v not in verts:
489: (8)                    raise ValueError(f"Vertex {v} not in graph from creation_sequence")
490: (4)                if u not in verts:
491: (8)                    raise ValueError(f"Vertex {u} not in graph from creation_sequence")
492: (4)                if u == v:
493: (8)                    return [u]
494: (4)                uindex = verts.index(u)
495: (4)                vindex = verts.index(v)
496: (4)                bigind = max(uindex, vindex)
497: (4)                if cs[bigind][1] == "d":
498: (8)                    return [u, v]
499: (4)                cs = cs[bigind:]
500: (4)                while cs:
501: (8)                    vert = cs.pop()
502: (8)                    if vert[1] == "d":
503: (12)                       return [u, vert[0], v]
504: (4)                return -1
505: (0)            def shortest_path_length(creation_sequence, i):
506: (4)                """
507: (4)                Return the shortest path length from indicated node to
508: (4)                every other node for the threshold graph with the given
509: (4)                creation sequence.
510: (4)                Node is indicated by index i in creation_sequence unless
511: (4)                creation_sequence is labeled in which case, i is taken to
512: (4)                be the label of the node.
513: (4)                Paths lengths in threshold graphs are at most 2.
514: (4)                Length to unreachable nodes is set to -1.
515: (4)                """
516: (4)                first = creation_sequence[0]
517: (4)                if isinstance(first, str):  # creation sequence
518: (8)                    if isinstance(creation_sequence, list):
519: (12)                       cs = creation_sequence[:]
520: (8)                    else:
521: (12)                       cs = list(creation_sequence)
522: (4)                elif isinstance(first, tuple):  # labeled creation sequence
523: (8)                    cs = [v[1] for v in creation_sequence]
524: (8)                    i = [v[0] for v in creation_sequence].index(i)
525: (4)                elif isinstance(first, int):  # compact creation sequence
526: (8)                    cs = uncompact(creation_sequence)
527: (4)                else:
528: (8)                    raise TypeError("Not a valid creation sequence type")
529: (4)                N = len(cs)
530: (4)                spl = [2] * N  # length 2 to every node
```

```
(4)                 spl[i] = 0  # except self which is 0
(4)                 for j in range(i + 1, N):
(8)                     if cs[j] == "d":
(12)                        spl[j] = 1
(4)                 if cs[i] == "d":  # 1 for all nodes to the left
(8)                     for j in range(i):
(12)                        spl[j] = 1
(4)                 for j in range(N - 1, 0, -1):
(8)                     if cs[j] == "d":
(12)                        break
(8)                     spl[j] = -1
(4)                 return spl
(0)             def betweenness_sequence(creation_sequence, normalized=True):
(4)                 """
(4)                 Return betweenness for the threshold graph with the given creation
(4)                 sequence.  The result is unscaled.  To scale the values
(4)                 to the interval [0,1] divide by (n-1)*(n-2).
(4)                 """
(4)                 cs = creation_sequence
(4)                 seq = []  # betweenness
(4)                 lastchar = "d"  # first node is always a 'd'
(4)                 dr = float(cs.count("d"))  # number of d's to the right of current pos
(4)                 irun = 0  # number of i's in the last run
(4)                 drun = 0  # number of d's in the last run
(4)                 dlast = 0.0  # betweenness of last d
(4)                 for i, c in enumerate(cs):
(8)                     if c == "d":  # cs[i]=="d":
(12)                        b = dlast + (irun - 1) * irun / dr + 2 * irun * (i - drun - irun)
/ dr
(12)                        drun += 1  # update counter
(8)                     else:  # cs[i]="i":
(12)                        if lastchar == "d":  # if this is a new run of i's
(16)                            dlast = b  # accumulate betweenness
(16)                            dr -= drun  # update number of d's to the right
(16)                            drun = 0  # reset d counter
(16)                            irun = 0  # reset i counter
(12)                        b = 0  # isolated nodes have zero betweenness
(12)                        irun += 1  # add another i to the run
(8)                     seq.append(float(b))
(8)                     lastchar = c
(4)                 if normalized:
(8)                     order = len(cs)
(8)                     scale = 1.0 / ((order - 1) * (order - 2))
(8)                     seq = [s * scale for s in seq]
(4)                 return seq
(0)             def eigenvectors(creation_sequence):
(4)                 """
(4)                 Return a 2-tuple of Laplacian eigenvalues and eigenvectors
(4)                 for the threshold network with creation_sequence.
(4)                 The first value is a list of eigenvalues.
(4)                 The second value is a list of eigenvectors.
(4)                 The lists are in the same order so corresponding eigenvectors
(4)                 and eigenvalues are in the same position in the two lists.
(4)                 Notice that the order of the eigenvalues returned by eigenvalues(cs)
(4)                 may not correspond to the order of these eigenvectors.
(4)                 """
(4)                 ccs = make_compact(creation_sequence)
(4)                 N = sum(ccs)
(4)                 vec = [0] * N
(4)                 val = vec[:]
(4)                 dr = sum(ccs[::2])
(4)                 nn = ccs[0]
(4)                 vec[0] = [1.0 / sqrt(N)] * N
(4)                 val[0] = 0
(4)                 e = dr
(4)                 dr -= nn
(4)                 type_d = True
(4)                 i = 1
(4)                 dd = 1
```

```
599: (4)                while dd < nn:
600: (8)                    scale = 1.0 / sqrt(dd * dd + i)
601: (8)                    vec[i] = i * [-scale] + [dd * scale] + [0] * (N - i - 1)
602: (8)                    val[i] = e
603: (8)                    i += 1
604: (8)                    dd += 1
605: (4)                if len(ccs) == 1:
606: (8)                    return (val, vec)
607: (4)                for nn in ccs[1:]:
608: (8)                    scale = 1.0 / sqrt(nn * i * (i + nn))
609: (8)                    vec[i] = i * [-nn * scale] + nn * [i * scale] + [0] * (N - i - nn)
610: (8)                    type_d = not type_d
611: (8)                    if type_d:
612: (12)                       e = i + dr
613: (12)                       dr -= nn
614: (8)                    else:
615: (12)                       e = dr
616: (8)                    val[i] = e
617: (8)                    st = i
618: (8)                    i += 1
619: (8)                    dd = 1
620: (8)                    while dd < nn:
621: (12)                       scale = 1.0 / sqrt(i - st + dd * dd)
622: (12)                       vec[i] = [0] * st + (i - st) * [-scale] + [dd * scale] + [0] * (N
- i - 1)
623: (12)                       val[i] = e
624: (12)                       i += 1
625: (12)                       dd += 1
626: (4)                return (val, vec)
627: (0)            def spectral_projection(u, eigenpairs):
628: (4)                """
629: (4)                Returns the coefficients of each eigenvector
630: (4)                in a projection of the vector u onto the normalized
631: (4)                eigenvectors which are contained in eigenpairs.
632: (4)                eigenpairs should be a list of two objects.  The
633: (4)                first is a list of eigenvalues and the second a list
634: (4)                of eigenvectors.  The eigenvectors should be lists.
635: (4)                There's not a lot of error checking on lengths of
636: (4)                arrays, etc. so be careful.
637: (4)                """
638: (4)                coeff = []
639: (4)                evect = eigenpairs[1]
640: (4)                for ev in evect:
641: (8)                    c = sum(evv * uv for (evv, uv) in zip(ev, u))
642: (8)                    coeff.append(c)
643: (4)                return coeff
644: (0)            def eigenvalues(creation_sequence):
645: (4)                """
646: (4)                Return sequence of eigenvalues of the Laplacian of the threshold
647: (4)                graph for the given creation_sequence.
648: (4)                Based on the Ferrer's diagram method.  The spectrum is integral
649: (4)                and is the conjugate of the degree sequence.
650: (4)                See::
651: (6)                  @Article{degree-merris-1994,
652: (7)                   author = {Russel Merris},
653: (7)                   title = {Degree maximal graphs are Laplacian integral},
654: (7)                   journal = {Linear Algebra Appl.},
655: (7)                   year = {1994},
656: (7)                   volume = {199},
657: (7)                   pages = {381--389},
658: (6)                  }
659: (4)                """
660: (4)                degseq = degree_sequence(creation_sequence)
661: (4)                degseq.sort()
662: (4)                eiglist = []  # zero is always one eigenvalue
663: (4)                eig = 0
664: (4)                row = len(degseq)
665: (4)                bigdeg = degseq.pop()
666: (4)                while row:
```

```
(8)                 if bigdeg < row:
(12)                    eiglist.append(eig)
(12)                    row -= 1
(8)                 else:
(12)                    eig += 1
(12)                    if degseq:
(16)                        bigdeg = degseq.pop()
(12)                    else:
(16)                        bigdeg = 0
(4)             return eiglist
(0)         @py_random_state(2)
(0)         def random_threshold_sequence(n, p, seed=None):
(4)             """
(4)             Create a random threshold sequence of size n.
(4)             A creation sequence is built by randomly choosing d's with
(4)             probability p and i's with probability 1-p.
(4)             s=nx.random_threshold_sequence(10,0.5)
(4)             returns a threshold sequence of length 10 with equal
(4)             probably of an i or a d at each position.
(4)             A "random" threshold graph can be built with
(4)             G=nx.threshold_graph(s)
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             """
(4)             if not (0 <= p <= 1):
(8)                 raise ValueError("p must be in [0,1]")
(4)             cs = ["d"]  # threshold sequences always start with a d
(4)             for i in range(1, n):
(8)                 if seed.random() < p:
(12)                    cs.append("d")
(8)                 else:
(12)                    cs.append("i")
(4)             return cs
(0)         def right_d_threshold_sequence(n, m):
(4)             """
(4)             Create a skewed threshold graph with a given number
(4)             of vertices (n) and a given number of edges (m).
(4)             The routine returns an unlabeled creation sequence
(4)             for the threshold graph.
(4)             FIXME: describe algorithm
(4)             """
(4)             cs = ["d"] + ["i"] * (n - 1)  # create sequence with n insolated nodes
(4)             if m < n:
(8)                 cs[m] = "d"
(8)                 return cs
(4)             if m > n * (n - 1) / 2:
(8)                 raise ValueError("Too many edges for this many nodes.")
(4)             ind = n - 1
(4)             sum = n - 1
(4)             while sum < m:
(8)                 cs[ind] = "d"
(8)                 ind -= 1
(8)                 sum += ind
(4)             ind = m - (sum - ind)
(4)             cs[ind] = "d"
(4)             return cs
(0)         def left_d_threshold_sequence(n, m):
(4)             """
(4)             Create a skewed threshold graph with a given number
(4)             of vertices (n) and a given number of edges (m).
(4)             The routine returns an unlabeled creation sequence
(4)             for the threshold graph.
(4)             FIXME: describe algorithm
(4)             """
(4)             cs = ["d"] + ["i"] * (n - 1)  # create sequence with n insolated nodes
(4)             if m < n:
(8)                 cs[m] = "d"
(8)                 return cs
```

```
736: (4)                if m > n * (n - 1) / 2:
737: (8)                    raise ValueError("Too many edges for this many nodes.")
738: (4)                cs[n - 1] = "d"
739: (4)                sum = n - 1
740: (4)                ind = 1
741: (4)                while sum < m:
742: (8)                    cs[ind] = "d"
743: (8)                    sum += ind
744: (8)                    ind += 1
745: (4)                if sum > m:  # be sure not to change the first vertex
746: (8)                    cs[sum - m] = "i"
747: (4)                return cs
748: (0)            @py_random_state(3)
749: (0)            def swap_d(cs, p_split=1.0, p_combine=1.0, seed=None):
750: (4)                """
751: (4)                Perform a "swap" operation on a threshold sequence.
752: (4)                The swap preserves the number of nodes and edges
753: (4)                in the graph for the given sequence.
754: (4)                The resulting sequence is still a threshold sequence.
755: (4)                Perform one split and one combine operation on the
756: (4)                'd's of a creation sequence for a threshold graph.
757: (4)                This operation maintains the number of nodes and edges
758: (4)                in the graph, but shifts the edges from node to node
759: (4)                maintaining the threshold quality of the graph.
760: (4)                seed : integer, random_state, or None (default)
761: (8)                    Indicator of random number generation state.
762: (8)                    See :ref:`Randomness<randomness>`.
763: (4)                """
764: (4)                dlist = [i for (i, node_type) in enumerate(cs[1:-1]) if node_type == "d"]
765: (4)                if seed.random() < p_split:
766: (8)                    choice = seed.choice(dlist)
767: (8)                    split_to = seed.choice(range(choice))
768: (8)                    flip_side = choice - split_to
769: (8)                    if split_to != flip_side and cs[split_to] == "i" and cs[flip_side] ==
"i":
770: (12)                       cs[choice] = "i"
771: (12)                       cs[split_to] = "d"
772: (12)                       cs[flip_side] = "d"
773: (12)                       dlist.remove(choice)
774: (4)                if seed.random() < p_combine and dlist:
775: (8)                    first_choice = seed.choice(dlist)
776: (8)                    second_choice = seed.choice(dlist)
777: (8)                    target = first_choice + second_choice
778: (8)                    if target >= len(cs) or cs[target] == "d" or first_choice ==
second_choice:
779: (12)                       return cs
780: (8)                    cs[first_choice] = "i"
781: (8)                    cs[second_choice] = "i"
782: (8)                    cs[target] = "d"
783: (4)                return cs
```

File 57 - time_dependent.py:

```
1: (0)              """Time dependent algorithms."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for
4: (0)              __all__ = ["cd_index"]
5: (0)              @not_implemented_for("undirected")
6: (0)              @not_implemented_for("multigraph")
7: (0)              @nx._dispatchable(node_attrs={"time": None, "weight": 1})
8: (0)              def cd_index(G, node, time_delta, *, time="time", weight=None):
9: (4)                  r"""Compute the CD index for `node` within the graph `G`.
10: (4)                 Calculates the CD index for the given node of the graph,
11: (4)                 considering only its predecessors who have the `time` attribute
12: (4)                 smaller than or equal to the `time` attribute of the `node`
13: (4)                 plus `time_delta`.
14: (4)                 Parameters
```

```
15: (4)                 ----------
16: (4)                 G : graph
17: (7)                    A directed networkx graph whose nodes have `time` attributes and
optionally
18: (7)                    `weight` attributes (if a weight is not given, it is considered 1).
19: (4)                 node : node
20: (7)                    The node for which the CD index is calculated.
21: (4)                 time_delta : numeric or timedelta
22: (7)                    Amount of time after the `time` attribute of the `node`. The value of
23: (7)                    `time_delta` must support comparison with the `time` node attribute.
For
24: (7)                    example, if the `time` attribute of the nodes are `datetime.datetime`
25: (7)                    objects, then `time_delta` should be a `datetime.timedelta` object.
26: (4)                 time : string (Optional, default is "time")
27: (8)                     The name of the node attribute that will be used for the calculations.
28: (4)                 weight : string (Optional, default is None)
29: (8)                     The name of the node attribute used as weight.
30: (4)                 Returns
31: (4)                 -------
32: (4)                 float
33: (7)                    The CD index calculated for the node `node` within the graph `G`.
34: (4)                 Raises
35: (4)                 ------
36: (4)                 NetworkXError
37: (7)                    If not all nodes have a `time` attribute or
38: (7)                    `time_delta` and `time` attribute types are not compatible or
39: (7)                    `n` equals 0.
40: (4)                 NetworkXNotImplemented
41: (8)                     If `G` is a non-directed graph or a multigraph.
42: (4)                 Examples
43: (4)                 --------
44: (4)                 >>> from datetime import datetime, timedelta
45: (4)                 >>> G = nx.DiGraph()
46: (4)                 >>> nodes = {
47: (4)                 ...     1: {"time": datetime(2015, 1, 1)},
48: (4)                 ...     2: {"time": datetime(2012, 1, 1), "weight": 4},
49: (4)                 ...     3: {"time": datetime(2010, 1, 1)},
50: (4)                 ...     4: {"time": datetime(2008, 1, 1)},
51: (4)                 ...     5: {"time": datetime(2014, 1, 1)},
52: (4)                 ... }
53: (4)                 >>> G.add_nodes_from([(n, nodes[n]) for n in nodes])
54: (4)                 >>> edges = [(1, 3), (1, 4), (2, 3), (3, 4), (3, 5)]
55: (4)                 >>> G.add_edges_from(edges)
56: (4)                 >>> delta = timedelta(days=5 * 365)
57: (4)                 >>> nx.cd_index(G, 3, time_delta=delta, time="time")
58: (4)                 0.5
59: (4)                 >>> nx.cd_index(G, 3, time_delta=delta, time="time", weight="weight")
60: (4)                 0.12
61: (4)                 Integers can also be used for the time values:
62: (4)                 >>> node_times = {1: 2015, 2: 2012, 3: 2010, 4: 2008, 5: 2014}
63: (4)                 >>> nx.set_node_attributes(G, node_times, "new_time")
64: (4)                 >>> nx.cd_index(G, 3, time_delta=4, time="new_time")
65: (4)                 0.5
66: (4)                 >>> nx.cd_index(G, 3, time_delta=4, time="new_time", weight="weight")
67: (4)                 0.12
68: (4)                 Notes
69: (4)                 -----
70: (4)                 This method implements the algorithm for calculating the CD index,
71: (4)                 as described in the paper by Funk and Owen-Smith [1]_. The CD index
72: (4)                 is used in order to check how consolidating or destabilizing a patent
73: (4)                 is, hence the nodes of the graph represent patents and the edges show
74: (4)                 the citations between these patents. The mathematical model is given
75: (4)                 below:
76: (4)                 .. math::
77: (8)                     CD_{t}=\frac{1}{n_{t}}\sum_{i=1}^{n}\frac{-2f_{it}b_{it}+f_{it}}
{w_{it}},
78: (4)                 where `f_{it}` equals 1 if `i` cites the focal patent else 0, `b_{it}`
equals
79: (4)                 1 if `i` cites any of the focal patents successors else 0, `n_{t}` is the
```

```
number
80: (4)                 of forward citations in `i` and `w_{it}` is a matrix of weight for patent
`i`
81: (4)                 at time `t`.
82: (4)                 The `datetime.timedelta` package can lead to off-by-one issues when
converting
83: (4)                 from years to days. In the example above `timedelta(days=5 * 365)` looks
like
84: (4)                 5 years, but it isn't because of leap year days. So it gives the same
result
85: (4)                 as `timedelta(days=4 * 365)`. But using `timedelta(days=5 * 365 + 1)`
gives
86: (4)                 a 5 year delta **for this choice of years** but may not if the 5 year gap
has
87: (4)                 more than 1 leap year. To avoid these issues, use integers to represent
years,
88: (4)                 or be very careful when you convert units of time.
89: (4)                 References
90: (4)                 ----------
91: (4)                 .. [1] Funk, Russell J., and Jason Owen-Smith.
92: (11)                       "A dynamic network measure of technological change."
93: (11)                       Management science 63, no. 3 (2017): 791-817.
94: (11)                       http://russellfunk.org/cdindex/static/papers/funk_ms_2017.pdf
95: (4)                 """
96: (4)                 if not all(time in G.nodes[n] for n in G):
97: (8)                     raise nx.NetworkXError("Not all nodes have a 'time' attribute.")
98: (4)                 try:
99: (8)                     target_date = G.nodes[node][time] + time_delta
100: (8)                    pred = {i for i in G.pred[node] if G.nodes[i][time] <= target_date}
101: (4)                except:
102: (8)                    raise nx.NetworkXError(
103: (12)                       "Addition and comparison are not supported between 'time_delta' "
104: (12)                       "and 'time' types."
105: (8)                    )
106: (4)                b = [-1 if any(j in G[i] for j in G[node]) else 1 for i in pred]
107: (4)                n = len(pred.union(*(G.pred[s].keys() - {node} for s in G[node])))
108: (4)                if n == 0:
109: (8)                    raise nx.NetworkXError("The cd index cannot be defined.")
110: (4)                if weight is None:
111: (8)                    return round(sum(bi for bi in b) / n, 2)
112: (4)                else:
113: (8)                    weights = [G.nodes[i].get(weight, 1) for i in pred]
114: (8)                    return round(sum(bi / wt for bi, wt in zip(b, weights)) / n, 2)

----------------------------------------

File 58 - tournament.py:

1: (0)              """Functions concerning tournament graphs.
2: (0)              A `tournament graph`_ is a complete oriented graph. In other words, it
3: (0)              is a directed graph in which there is exactly one directed edge joining
4: (0)              each pair of distinct nodes. For each function in this module that
5: (0)              accepts a graph as input, you must provide a tournament graph. The
6: (0)              responsibility is on the caller to ensure that the graph is a tournament
7: (0)              graph:
8: (4)                  >>> G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
9: (4)                  >>> nx.is_tournament(G)
10: (4)                 True
11: (0)             To access the functions in this module, you must access them through the
12: (0)             :mod:`networkx.tournament` module::
13: (4)                 >>> nx.tournament.is_reachable(G, 0, 1)
14: (4)                 True
15: (0)             .. _tournament graph:
https://en.wikipedia.org/wiki/Tournament_%28graph_theory%29
16: (0)             """
17: (0)             from itertools import combinations
18: (0)             import networkx as nx
19: (0)             from networkx.algorithms.simple_paths import is_simple_path as is_path
20: (0)             from networkx.utils import arbitrary_element, not_implemented_for,
```

```
py_random_state
__all__ = [
    "hamiltonian_path",
    "is_reachable",
    "is_strongly_connected",
    "is_tournament",
    "random_tournament",
    "score_sequence",
]
def index_satisfying(iterable, condition):
    """Returns the index of the first element in `iterable` that
    satisfies the given condition.
    If no such element is found (that is, when the iterable is
    exhausted), this returns the length of the iterable (that is, one
    greater than the last index of the iterable).
    `iterable` must not be empty. If `iterable` is empty, this
    function raises :exc:`ValueError`.
    """
    for i, x in enumerate(iterable):
        if condition(x):
            return i
    try:
        return i + 1
    except NameError as err:
        raise ValueError("iterable must be non-empty") from err
@not_implemented_for("undirected")
@not_implemented_for("multigraph")
@nx._dispatchable
def is_tournament(G):
    """Returns True if and only if `G` is a tournament.
    A tournament is a directed graph, with neither self-loops nor
    multi-edges, in which there is exactly one directed edge joining
    each pair of distinct nodes.
    Parameters
    ----------
    G : NetworkX graph
        A directed graph representing a tournament.
    Returns
    -------
    bool
        Whether the given graph is a tournament graph.
    Examples
    --------
    >>> G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
    >>> nx.is_tournament(G)
    True
    Notes
    -----
    Some definitions require a self-loop on each node, but that is not
    the convention used here.
    """
    return (
        all((v in G[u]) ^ (u in G[v]) for u, v in combinations(G, 2))
        and nx.number_of_selfloops(G) == 0
    )
@not_implemented_for("undirected")
@not_implemented_for("multigraph")
@nx._dispatchable
def hamiltonian_path(G):
    """Returns a Hamiltonian path in the given tournament graph.
    Each tournament has a Hamiltonian path. If furthermore, the
    tournament is strongly connected, then the returned Hamiltonian path
    is a Hamiltonian cycle (by joining the endpoints of the path).
    Parameters
    ----------
    G : NetworkX graph
        A directed graph representing a tournament.
    Returns
    -------
```

```
(4)                 path : list
(8)                     A list of nodes which form a Hamiltonian path in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)])
(4)                 >>> nx.is_tournament(G)
(4)                 True
(4)                 >>> nx.tournament.hamiltonian_path(G)
(4)                 [0, 1, 2, 3]
(4)                 Notes
(4)                 -----
(4)                 This is a recursive implementation with an asymptotic running time
(4)                 of $O(n^2)$, ignoring multiplicative polylogarithmic factors, where
(4)                 $n$ is the number of nodes in the graph.
(4)                 """
(4)                 if len(G) == 0:
(8)                     return []
(4)                 if len(G) == 1:
(8)                     return [arbitrary_element(G)]
(4)                 v = arbitrary_element(G)
(4)                 hampath = hamiltonian_path(G.subgraph(set(G) - {v}))
(4)                 index = index_satisfying(hampath, lambda u: v not in G[u])
(4)                 hampath.insert(index, v)
(4)                 return hampath
(0)             @py_random_state(1)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_tournament(n, seed=None):
(4)                 r"""Returns a random tournament graph on `n` nodes.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes in the returned graph.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 G : DiGraph
(8)                     A tournament on `n` nodes, with exactly one directed edge joining
(8)                     each pair of distinct nodes.
(4)                 Notes
(4)                 -----
(4)                 This algorithm adds, for each pair of distinct nodes, an edge with
(4)                 uniformly random orientation. In other words, `\binom{n}{2}` flips
(4)                 of an unbiased coin decide the orientations of the edges in the
(4)                 graph.
(4)                 """
(4)                 coins = (seed.random() for i in range((n * (n - 1)) // 2))
(4)                 pairs = combinations(range(n), 2)
(4)                 edges = ((u, v) if r < 0.5 else (v, u) for (u, v), r in zip(pairs, coins))
(4)                 return nx.DiGraph(edges)
(0)             @not_implemented_for("undirected")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable
(0)             def score_sequence(G):
(4)                 """Returns the score sequence for the given tournament graph.
(4)                 The score sequence is the sorted list of the out-degrees of the
(4)                 nodes of the graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     A directed graph representing a tournament.
(4)                 Returns
(4)                 -------
(4)                 list
(8)                     A sorted list of the out-degrees of the nodes of `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph([(1, 0), (1, 3), (0, 2), (0, 3), (2, 1), (3, 2)])
```

```
158: (4)                >>> nx.is_tournament(G)
159: (4)                True
160: (4)                >>> nx.tournament.score_sequence(G)
161: (4)                [1, 1, 2, 2]
162: (4)                """
163: (4)                return sorted(d for v, d in G.out_degree())
164: (0)            @not_implemented_for("undirected")
165: (0)            @not_implemented_for("multigraph")
166: (0)            @nx._dispatchable(preserve_edge_attrs={"G": {"weight": 1}})
167: (0)            def tournament_matrix(G):
168: (4)                r"""Returns the tournament matrix for the given tournament graph.
169: (4)                This function requires SciPy.
170: (4)                The *tournament matrix* of a tournament graph with edge set *E* is
171: (4)                the matrix *T* defined by
172: (4)                .. math::
173: (7)                   T_{i j} =
174: (7)                   \begin{cases}
175: (7)                   +1 & \text{if } (i, j) \in E \\
176: (7)                   -1 & \text{if } (j, i) \in E \\
177: (7)                   0 & \text{if } i == j.
178: (7)                   \end{cases}
179: (4)                An equivalent definition is `T = A - A^T`, where *A* is the
180: (4)                adjacency matrix of the graph `G`.
181: (4)                Parameters
182: (4)                ----------
183: (4)                G : NetworkX graph
184: (8)                    A directed graph representing a tournament.
185: (4)                Returns
186: (4)                -------
187: (4)                SciPy sparse array
188: (8)                    The tournament matrix of the tournament graph `G`.
189: (4)                Raises
190: (4)                ------
191: (4)                ImportError
192: (8)                    If SciPy is not available.
193: (4)                """
194: (4)                A = nx.adjacency_matrix(G)
195: (4)                return A - A.T
196: (0)            @not_implemented_for("undirected")
197: (0)            @not_implemented_for("multigraph")
198: (0)            @nx._dispatchable
199: (0)            def is_reachable(G, s, t):
200: (4)                """Decides whether there is a path from `s` to `t` in the
201: (4)                tournament.
202: (4)                This function is more theoretically efficient than the reachability
203: (4)                checks than the shortest path algorithms in
204: (4)                :mod:`networkx.algorithms.shortest_paths`.
205: (4)                The given graph **must** be a tournament, otherwise this function's
206: (4)                behavior is undefined.
207: (4)                Parameters
208: (4)                ----------
209: (4)                G : NetworkX graph
210: (8)                    A directed graph representing a tournament.
211: (4)                s : node
212: (8)                    A node in the graph.
213: (4)                t : node
214: (8)                    A node in the graph.
215: (4)                Returns
216: (4)                -------
217: (4)                bool
218: (8)                    Whether there is a path from `s` to `t` in `G`.
219: (4)                Examples
220: (4)                --------
221: (4)                >>> G = nx.DiGraph([(1, 0), (1, 3), (1, 2), (2, 3), (2, 0), (3, 0)])
222: (4)                >>> nx.is_tournament(G)
223: (4)                True
224: (4)                >>> nx.tournament.is_reachable(G, 1, 3)
225: (4)                True
226: (4)                >>> nx.tournament.is_reachable(G, 3, 2)
```

```
227: (4)                False
228: (4)                Notes
229: (4)                -----
230: (4)                Although this function is more theoretically efficient than the
231: (4)                generic shortest path functions, a speedup requires the use of
232: (4)                parallelism. Though it may in the future, the current implementation
233: (4)                does not use parallelism, thus you may not see much of a speedup.
234: (4)                This algorithm comes from [1].
235: (4)                References
236: (4)                ----------
237: (4)                .. [1] Tantau, Till.
238: (11)                      "A note on the complexity of the reachability problem for
239: (11)                      tournaments."
240: (11)                      *Electronic Colloquium on Computational Complexity*. 2001.
241: (11)                      <http://eccc.hpi-web.de/report/2001/092/>
242: (4)                """
243: (4)                def two_neighborhood(G, v):
244: (8)                    """Returns the set of nodes at distance at most two from `v`.
245: (8)                    `G` must be a graph and `v` a node in that graph.
246: (8)                    The returned set includes the nodes at distance zero (that is,
247: (8)                    the node `v` itself), the nodes at distance one (that is, the
248: (8)                    out-neighbors of `v`), and the nodes at distance two.
249: (8)                    """
250: (8)                    return {
251: (12)                       x for x in G if x == v or x in G[v] or any(is_path(G, [v, z, x])
for z in G)
252: (8)                    }
253: (4)                def is_closed(G, nodes):
254: (8)                    """Decides whether the given set of nodes is closed.
255: (8)                    A set *S* of nodes is *closed* if for each node *u* in the graph
256: (8)                    not in *S* and for each node *v* in *S*, there is an edge from
257: (8)                    *u* to *v*.
258: (8)                    """
259: (8)                    return all(v in G[u] for u in set(G) - nodes for v in nodes)
260: (4)                neighborhoods = [two_neighborhood(G, v) for v in G]
261: (4)                return all(not (is_closed(G, S) and s in S and t not in S) for S in
neighborhoods)
262: (0)            @not_implemented_for("undirected")
263: (0)            @not_implemented_for("multigraph")
264: (0)            @nx._dispatchable(name="tournament_is_strongly_connected")
265: (0)            def is_strongly_connected(G):
266: (4)                """Decides whether the given tournament is strongly connected.
267: (4)                This function is more theoretically efficient than the
268: (4)                :func:`~networkx.algorithms.components.is_strongly_connected`
269: (4)                function.
270: (4)                The given graph **must** be a tournament, otherwise this function's
271: (4)                behavior is undefined.
272: (4)                Parameters
273: (4)                ----------
274: (4)                G : NetworkX graph
275: (8)                    A directed graph representing a tournament.
276: (4)                Returns
277: (4)                -------
278: (4)                bool
279: (8)                    Whether the tournament is strongly connected.
280: (4)                Examples
281: (4)                --------
282: (4)                >>> G = nx.DiGraph([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 0)])
283: (4)                >>> nx.is_tournament(G)
284: (4)                True
285: (4)                >>> nx.tournament.is_strongly_connected(G)
286: (4)                True
287: (4)                >>> G.remove_edge(3, 0)
288: (4)                >>> G.add_edge(0, 3)
289: (4)                >>> nx.is_tournament(G)
290: (4)                True
291: (4)                >>> nx.tournament.is_strongly_connected(G)
292: (4)                False
293: (4)                Notes
```

```
294: (4)                -----
295: (4)                Although this function is more theoretically efficient than the
296: (4)                generic strong connectivity function, a speedup requires the use of
297: (4)                parallelism. Though it may in the future, the current implementation
298: (4)                does not use parallelism, thus you may not see much of a speedup.
299: (4)                This algorithm comes from [1].
300: (4)                References
301: (4)                ----------
302: (4)                .. [1] Tantau, Till.
303: (11)                      "A note on the complexity of the reachability problem for
304: (11)                      tournaments."
305: (11)                      *Electronic Colloquium on Computational Complexity*. 2001.
306: (11)                      <http://eccc.hpi-web.de/report/2001/092/>
307: (4)                """
308: (4)                return all(is_reachable(G, u, v) for u in G for v in G)
```

File 59 - triads.py:

```
1: (0)              """Functions for analyzing triads of a graph."""
2: (0)              from collections import defaultdict
3: (0)              from itertools import combinations, permutations
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for, py_random_state
6: (0)              __all__ = [
7: (4)                  "triadic_census",
8: (4)                  "is_triad",
9: (4)                  "all_triplets",
10: (4)                 "all_triads",
11: (4)                 "triads_by_type",
12: (4)                 "triad_type",
13: (4)                 "random_triad",
14: (0)             ]
15: (0)             TRICODES = (
16: (4)                 1,
17: (4)                 2,
18: (4)                 2,
19: (4)                 3,
20: (4)                 2,
21: (4)                 4,
22: (4)                 6,
23: (4)                 8,
24: (4)                 2,
25: (4)                 6,
26: (4)                 5,
27: (4)                 7,
28: (4)                 3,
29: (4)                 8,
30: (4)                 7,
31: (4)                 11,
32: (4)                 2,
33: (4)                 6,
34: (4)                 4,
35: (4)                 8,
36: (4)                 5,
37: (4)                 9,
38: (4)                 9,
39: (4)                 13,
40: (4)                 6,
41: (4)                 10,
42: (4)                 9,
43: (4)                 14,
44: (4)                 7,
45: (4)                 14,
46: (4)                 12,
47: (4)                 15,
48: (4)                 2,
49: (4)                 5,
```

```
50: (4)                 6,
51: (4)                 7,
52: (4)                 6,
53: (4)                 9,
54: (4)                 10,
55: (4)                 14,
56: (4)                 4,
57: (4)                 9,
58: (4)                 9,
59: (4)                 12,
60: (4)                 8,
61: (4)                 13,
62: (4)                 14,
63: (4)                 15,
64: (4)                 3,
65: (4)                 7,
66: (4)                 8,
67: (4)                 11,
68: (4)                 7,
69: (4)                 12,
70: (4)                 14,
71: (4)                 15,
72: (4)                 8,
73: (4)                 14,
74: (4)                 13,
75: (4)                 15,
76: (4)                 11,
77: (4)                 15,
78: (4)                 15,
79: (4)                 16,
80: (0)             )
81: (0)             TRIAD_NAMES = (
82: (4)                 "003",
83: (4)                 "012",
84: (4)                 "102",
85: (4)                 "021D",
86: (4)                 "021U",
87: (4)                 "021C",
88: (4)                 "111D",
89: (4)                 "111U",
90: (4)                 "030T",
91: (4)                 "030C",
92: (4)                 "201",
93: (4)                 "120D",
94: (4)                 "120U",
95: (4)                 "120C",
96: (4)                 "210",
97: (4)                 "300",
98: (0)             )
99: (0)             TRICODE_TO_NAME = {i: TRIAD_NAMES[code - 1] for i, code in
enumerate(TRICODES)}
100: (0)            def _tricode(G, v, u, w):
101: (4)                """Returns the integer code of the given triad.
102: (4)                This is some fancy magic that comes from Batagelj and Mrvar's paper. It
103: (4)                treats each edge joining a pair of `v`, `u`, and `w` as a bit in
104: (4)                the binary representation of an integer.
105: (4)                """
106: (4)                combos = ((v, u, 1), (u, v, 2), (v, w, 4), (w, v, 8), (u, w, 16), (w, u,
32))
107: (4)                return sum(x for u, v, x in combos if v in G[u])
108: (0)            @not_implemented_for("undirected")
109: (0)            @nx._dispatchable
110: (0)            def triadic_census(G, nodelist=None):
111: (4)                """Determines the triadic census of a directed graph.
112: (4)                The triadic census is a count of how many of the 16 possible types of
113: (4)                triads are present in a directed graph. If a list of nodes is passed, then
114: (4)                only those triads are taken into account which have elements of nodelist
in them.
115: (4)                Parameters
```

```
    ----------
    G : digraph
       A NetworkX DiGraph
    nodelist : list
        List of nodes for which you want to calculate triadic census
    Returns
    -------
    census : dict
       Dictionary with triad type as keys and number of occurrences as values.
    Examples
    --------
    >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1), (3, 4), (4, 1), (4, 2)])
    >>> triadic_census = nx.triadic_census(G)
    >>> for key, value in triadic_census.items():
    ...     print(f"{key}: {value}")
    003: 0
    012: 0
    102: 0
    021D: 0
    021U: 0
    021C: 0
    111D: 0
    111U: 0
    030T: 2
    030C: 2
    201: 0
    120D: 0
    120U: 0
    120C: 0
    210: 0
    300: 0
    Notes
    -----
    This algorithm has complexity $O(m)$ where $m$ is the number of edges in
    the graph.
    For undirected graphs, the triadic census can be computed by first converting
    the graph into a directed graph using the ``G.to_directed()`` method.
    After this conversion, only the triad types 003, 102, 201 and 300 will be
    present in the undirected scenario.
    Raises
    ------
    ValueError
        If `nodelist` contains duplicate nodes or nodes not in `G`.
        If you want to ignore this you can preprocess with `set(nodelist) & G.nodes`
    See also
    --------
    triad_graph
    References
    ----------
    .. [1] Vladimir Batagelj and Andrej Mrvar, A subquadratic triad census
        algorithm for large sparse networks with small maximum degree,
        University of Ljubljana,
        http://vlado.fmf.uni-lj.si/pub/networks/doc/triads/triads.pdf
    """
    nodeset = set(G.nbunch_iter(nodelist))
    if nodelist is not None and len(nodelist) != len(nodeset):
        raise ValueError("nodelist includes duplicate nodes or nodes not in G")
    N = len(G)
    Nnot = N - len(nodeset)  # can signal special counting for subset of nodes
    m = {n: i for i, n in enumerate(nodeset)}
    if Nnot:
        not_nodeset = G.nodes - nodeset
        m.update((n, i + N) for i, n in enumerate(not_nodeset))
    nbrs = {n: G.pred[n].keys() | G.succ[n].keys() for n in G}
    dbl_nbrs = {n: G.pred[n].keys() & G.succ[n].keys() for n in G}
    if Nnot:
```

```
182: (8)                    sgl_nbrs = {n: G.pred[n].keys() ^ G.succ[n].keys() for n in
not_nodeset}
183: (8)                    sgl = sum(1 for n in not_nodeset for nbr in sgl_nbrs[n] if nbr not in
nodeset)
184: (8)                    sgl_edges_outside = sgl // 2
185: (8)                    dbl = sum(1 for n in not_nodeset for nbr in dbl_nbrs[n] if nbr not in
nodeset)
186: (8)                    dbl_edges_outside = dbl // 2
187: (4)                census = {name: 0 for name in TRIAD_NAMES}
188: (4)                for v in nodeset:
189: (8)                    vnbrs = nbrs[v]
190: (8)                    dbl_vnbrs = dbl_nbrs[v]
191: (8)                    if Nnot:
192: (12)                       sgl_unbrs_bdy = sgl_unbrs_out = dbl_unbrs_bdy = dbl_unbrs_out = 0
193: (8)                    for u in vnbrs:
194: (12)                       if m[u] <= m[v]:
195: (16)                           continue
196: (12)                       unbrs = nbrs[u]
197: (12)                       neighbors = (vnbrs | unbrs) - {u, v}
198: (12)                       for w in neighbors:
199: (16)                           if m[u] < m[w] or (m[v] < m[w] < m[u] and v not in nbrs[w]):
200: (20)                               code = _tricode(G, v, u, w)
201: (20)                               census[TRICODE_TO_NAME[code]] += 1
202: (12)                       if u in dbl_vnbrs:
203: (16)                           census["102"] += N - len(neighbors) - 2
204: (12)                       else:
205: (16)                           census["012"] += N - len(neighbors) - 2
206: (12)                       if Nnot and u not in nodeset:
207: (16)                           sgl_unbrs = sgl_nbrs[u]
208: (16)                           sgl_unbrs_bdy += len(sgl_unbrs & vnbrs - nodeset)
209: (16)                           sgl_unbrs_out += len(sgl_unbrs - vnbrs - nodeset)
210: (16)                           dbl_unbrs = dbl_nbrs[u]
211: (16)                           dbl_unbrs_bdy += len(dbl_unbrs & vnbrs - nodeset)
212: (16)                           dbl_unbrs_out += len(dbl_unbrs - vnbrs - nodeset)
213: (8)                    if Nnot:
214: (12)                       census["012"] += sgl_edges_outside - (sgl_unbrs_out +
sgl_unbrs_bdy // 2)
215: (12)                       census["102"] += dbl_edges_outside - (dbl_unbrs_out +
dbl_unbrs_bdy // 2)
216: (4)                total_triangles = (N * (N - 1) * (N - 2)) // 6
217: (4)                triangles_without_nodeset = (Nnot * (Nnot - 1) * (Nnot - 2)) // 6
218: (4)                total_census = total_triangles - triangles_without_nodeset
219: (4)                census["003"] = total_census - sum(census.values())
220: (4)                return census
221: (0)            @nx._dispatchable
222: (0)            def is_triad(G):
223: (4)                """Returns True if the graph G is a triad, else False.
224: (4)                Parameters
225: (4)                ----------
226: (4)                G : graph
227: (7)                   A NetworkX Graph
228: (4)                Returns
229: (4)                -------
230: (4)                istriad : boolean
231: (7)                   Whether G is a valid triad
232: (4)                Examples
233: (4)                --------
234: (4)                >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
235: (4)                >>> nx.is_triad(G)
236: (4)                True
237: (4)                >>> G.add_edge(0, 1)
238: (4)                >>> nx.is_triad(G)
239: (4)                False
240: (4)                """
241: (4)                if isinstance(G, nx.Graph):
242: (8)                    if G.order() == 3 and nx.is_directed(G):
243: (12)                       if not any((n, n) in G.edges() for n in G.nodes()):
244: (16)                           return True
245: (4)                return False
```

```
(0)            @not_implemented_for("undirected")
(0)            @nx._dispatchable
(0)            def all_triplets(G):
(4)                """Returns a generator of all possible sets of 3 nodes in a DiGraph.
(4)                .. deprecated:: 3.3
(7)                   all_triplets is deprecated and will be removed in NetworkX version 3.5.
(7)                   Use `itertools.combinations` instead::
(10)                     all_triplets = itertools.combinations(G, 3)
(4)                Parameters
(4)                ----------
(4)                G : digraph
(7)                   A NetworkX DiGraph
(4)                Returns
(4)                -------
(4)                triplets : generator of 3-tuples
(7)                   Generator of tuples of 3 nodes
(4)                Examples
(4)                --------
(4)                >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
(4)                >>> list(nx.all_triplets(G))
(4)                [(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
(4)                """
(4)                import warnings
(4)                warnings.warn(
(8)                    (
(12)                       "\n\nall_triplets is deprecated and will be rmoved in v3.5.\n"
(12)                       "Use `itertools.combinations(G, 3)` instead."
(8)                    ),
(8)                    category=DeprecationWarning,
(8)                    stacklevel=4,
(4)                )
(4)                triplets = combinations(G.nodes(), 3)
(4)                return triplets
(0)            @not_implemented_for("undirected")
(0)            @nx._dispatchable(returns_graph=True)
(0)            def all_triads(G):
(4)                """A generator of all possible triads in G.
(4)                Parameters
(4)                ----------
(4)                G : digraph
(7)                   A NetworkX DiGraph
(4)                Returns
(4)                -------
(4)                all_triads : generator of DiGraphs
(7)                   Generator of triads (order-3 DiGraphs)
(4)                Examples
(4)                --------
(4)                >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1), (3, 4), (4, 1), (4, 2)])
(4)                >>> for triad in nx.all_triads(G):
(4)                ...     print(triad.edges)
(4)                [(1, 2), (2, 3), (3, 1)]
(4)                [(1, 2), (4, 1), (4, 2)]
(4)                [(3, 1), (3, 4), (4, 1)]
(4)                [(2, 3), (3, 4), (4, 2)]
(4)                """
(4)                triplets = combinations(G.nodes(), 3)
(4)                for triplet in triplets:
(8)                    yield G.subgraph(triplet).copy()
(0)            @not_implemented_for("undirected")
(0)            @nx._dispatchable
(0)            def triads_by_type(G):
(4)                """Returns a list of all triads for each triad type in a directed graph.
(4)                There are exactly 16 different types of triads possible. Suppose 1, 2, 3
are three
(4)                nodes, they will be classified as a particular triad type if their
connections
(4)                are as follows:
(4)                - 003: 1, 2, 3
(4)                - 012: 1 -> 2, 3
```

```
(4)                - 102: 1 <-> 2, 3
(4)                - 021D: 1 <- 2 -> 3
(4)                - 021U: 1 -> 2 <- 3
(4)                - 021C: 1 -> 2 -> 3
(4)                - 111D: 1 <-> 2 <- 3
(4)                - 111U: 1 <-> 2 -> 3
(4)                - 030T: 1 -> 2 -> 3, 1 -> 3
(4)                - 030C: 1 <- 2 <- 3, 1 -> 3
(4)                - 201: 1 <-> 2 <-> 3
(4)                - 120D: 1 <- 2 -> 3, 1 <-> 3
(4)                - 120U: 1 -> 2 <- 3, 1 <-> 3
(4)                - 120C: 1 -> 2 -> 3, 1 <-> 3
(4)                - 210: 1 -> 2 <-> 3, 1 <-> 3
(4)                - 300: 1 <-> 2 <-> 3, 1 <-> 3
(4)                Refer to the :doc:`example gallery
</auto_examples/graph/plot_triad_types>`
(4)                for visual examples of the triad types.
(4)                Parameters
(4)                ----------
(4)                G : digraph
(7)                   A NetworkX DiGraph
(4)                Returns
(4)                -------
(4)                tri_by_type : dict
(7)                   Dictionary with triad types as keys and lists of triads as values.
(4)                Examples
(4)                --------
(4)                >>> G = nx.DiGraph([(1, 2), (1, 3), (2, 3), (3, 1), (5, 6), (5, 4), (6,
7)])
(4)                >>> dict = nx.triads_by_type(G)
(4)                >>> dict["120C"][0].edges()
(4)                OutEdgeView([(1, 2), (1, 3), (2, 3), (3, 1)])
(4)                >>> dict["012"][0].edges()
(4)                OutEdgeView([(1, 2)])
(4)                References
(4)                ----------
(4)                .. [1] Snijders, T. (2012). "Transitivity and triads." University of
(8)                    Oxford.
(8)
https://web.archive.org/web/20170830032057/http://www.stats.ox.ac.uk/~snijders/Trans_Triads_ha.pdf
(4)                """
(4)                all_tri = all_triads(G)
(4)                tri_by_type = defaultdict(list)
(4)                for triad in all_tri:
(8)                    name = triad_type(triad)
(8)                    tri_by_type[name].append(triad)
(4)                return tri_by_type
(0)            @not_implemented_for("undirected")
(0)            @nx._dispatchable
(0)            def triad_type(G):
(4)                """Returns the sociological triad type for a triad.
(4)                Parameters
(4)                ----------
(4)                G : digraph
(7)                   A NetworkX DiGraph with 3 nodes
(4)                Returns
(4)                -------
(4)                triad_type : str
(7)                   A string identifying the triad type
(4)                Examples
(4)                --------
(4)                >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
(4)                >>> nx.triad_type(G)
(4)                '030C'
(4)                >>> G.add_edge(1, 3)
(4)                >>> nx.triad_type(G)
(4)                '120C'
(4)                Notes
(4)                -----
```

```
379: (4)                There can be 6 unique edges in a triad (order-3 DiGraph) (so 2^^6=64
unique
380: (4)                triads given 3 nodes). These 64 triads each display exactly 1 of 16
381: (4)                topologies of triads (topologies can be permuted). These topologies are
382: (4)                identified by the following notation:
383: (4)                {m}{a}{n}{type} (for example: 111D, 210, 102)
384: (4)                Here:
385: (4)                {m}     = number of mutual ties (takes 0, 1, 2, 3); a mutual tie is (0,1)
386: (14)                         AND (1,0)
387: (4)                {a}     = number of asymmetric ties (takes 0, 1, 2, 3); an asymmetric tie
388: (14)                         is (0,1) BUT NOT (1,0) or vice versa
389: (4)                {n}     = number of null ties (takes 0, 1, 2, 3); a null tie is NEITHER
390: (14)                         (0,1) NOR (1,0)
391: (4)                {type}  = a letter (takes U, D, C, T) corresponding to up, down, cyclical
392: (14)                         and transitive. This is only used for topologies that can have
393: (14)                         more than one form (eg: 021D and 021U).
394: (4)                References
395: (4)                ----------
396: (4)                .. [1] Snijders, T. (2012). "Transitivity and triads." University of
397: (8)                    Oxford.
398: (8)
https://web.archive.org/web/20170830032057/http://www.stats.ox.ac.uk/~snijders/Trans_Triads_ha.pdf
399: (4)                """
400: (4)                if not is_triad(G):
401: (8)                    raise nx.NetworkXAlgorithmError("G is not a triad (order-3 DiGraph)")
402: (4)                num_edges = len(G.edges())
403: (4)                if num_edges == 0:
404: (8)                    return "003"
405: (4)                elif num_edges == 1:
406: (8)                    return "012"
407: (4)                elif num_edges == 2:
408: (8)                    e1, e2 = G.edges()
409: (8)                    if set(e1) == set(e2):
410: (12)                       return "102"
411: (8)                    elif e1[0] == e2[0]:
412: (12)                       return "021D"
413: (8)                    elif e1[1] == e2[1]:
414: (12)                       return "021U"
415: (8)                    elif e1[1] == e2[0] or e2[1] == e1[0]:
416: (12)                       return "021C"
417: (4)                elif num_edges == 3:
418: (8)                    for e1, e2, e3 in permutations(G.edges(), 3):
419: (12)                       if set(e1) == set(e2):
420: (16)                           if e3[0] in e1:
421: (20)                               return "111U"
422: (16)                           return "111D"
423: (12)                       elif set(e1).symmetric_difference(set(e2)) == set(e3):
424: (16)                           if {e1[0], e2[0], e3[0]} == {e1[0], e2[0], e3[0]} ==
set(G.nodes()):
425: (20)                               return "030C"
426: (16)                           return "030T"
427: (4)                elif num_edges == 4:
428: (8)                    for e1, e2, e3, e4 in permutations(G.edges(), 4):
429: (12)                       if set(e1) == set(e2):
430: (16)                           if set(e3) == set(e4):
431: (20)                               return "201"
432: (16)                           if {e3[0]} == {e4[0]} == set(e3).intersection(set(e4)):
433: (20)                               return "120D"
434: (16)                           if {e3[1]} == {e4[1]} == set(e3).intersection(set(e4)):
435: (20)                               return "120U"
436: (16)                           if e3[1] == e4[0]:
437: (20)                               return "120C"
438: (4)                elif num_edges == 5:
439: (8)                    return "210"
440: (4)                elif num_edges == 6:
441: (8)                    return "300"
442: (0)            @not_implemented_for("undirected")
443: (0)            @py_random_state(1)
444: (0)            @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
```

```
def random_triad(G, seed=None):
    """Returns a random triad from a directed graph.
    .. deprecated:: 3.3
       random_triad is deprecated and will be removed in version 3.5.
       Use random sampling directly instead::
          G.subgraph(random.sample(list(G), 3))
    Parameters
    ----------
    G : digraph
       A NetworkX DiGraph
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    Returns
    -------
    G2 : subgraph
       A randomly selected triad (order-3 NetworkX DiGraph)
    Raises
    ------
    NetworkXError
        If the input Graph has less than 3 nodes.
    Examples
    --------
    >>> G = nx.DiGraph([(1, 2), (1, 3), (2, 3), (3, 1), (5, 6), (5, 4), (6, 7)])
    >>> triad = nx.random_triad(G, seed=1)
    >>> triad.edges
    OutEdgeView([(1, 2)])
    """
    import warnings
    warnings.warn(
        (
            "\n\nrandom_triad is deprecated and will be removed in NetworkX v3.5.\n"
            "Use random.sample instead, e.g.::\n\n"
            "\tG.subgraph(random.sample(list(G), 3))\n"
        ),
        category=DeprecationWarning,
        stacklevel=5,
    )
    if len(G) < 3:
        raise nx.NetworkXError(
            f"G needs at least 3 nodes to form a triad; (it has {len(G)} nodes)"
        )
    nodes = seed.sample(list(G.nodes()), 3)
    G2 = G.subgraph(nodes)
    return G2
```

File 60 - vitality.py:

```
"""
Vitality measures.
"""
from functools import partial
import networkx as nx
__all__ = ["closeness_vitality"]
@nx._dispatchable(edge_attrs="weight")
def closeness_vitality(G, node=None, weight=None, wiener_index=None):
    """Returns the closeness vitality for nodes in the graph.
    The *closeness vitality* of a node, defined in Section 3.6.2 of [1],
    is the change in the sum of distances between all node pairs when
    excluding that node.
    Parameters
    ----------
    G : NetworkX graph
        A strongly-connected graph.
```

```
17: (4)                 weight : string
18: (8)                     The name of the edge attribute used as weight. This is passed
19: (8)                     directly to the :func:`~networkx.wiener_index` function.
20: (4)                 node : object
21: (8)                     If specified, only the closeness vitality for this node will be
22: (8)                     returned. Otherwise, a dictionary mapping each node to its
23: (8)                     closeness vitality will be returned.
24: (4)                 Other parameters
25: (4)                 ----------------
26: (4)                 wiener_index : number
27: (8)                     If you have already computed the Wiener index of the graph
28: (8)                     `G`, you can provide that value here. Otherwise, it will be
29: (8)                     computed for you.
30: (4)                 Returns
31: (4)                 -------
32: (4)                 dictionary or float
33: (8)                     If `node` is None, this function returns a dictionary
34: (8)                     with nodes as keys and closeness vitality as the
35: (8)                     value. Otherwise, it returns only the closeness vitality for the
36: (8)                     specified `node`.
37: (8)                     The closeness vitality of a node may be negative infinity if
38: (8)                     removing that node would disconnect the graph.
39: (4)                 Examples
40: (4)                 --------
41: (4)                 >>> G = nx.cycle_graph(3)
42: (4)                 >>> nx.closeness_vitality(G)
43: (4)                 {0: 2.0, 1: 2.0, 2: 2.0}
44: (4)                 See Also
45: (4)                 --------
46: (4)                 closeness_centrality
47: (4)                 References
48: (4)                 ----------
49: (4)                 .. [1] Ulrik Brandes, Thomas Erlebach (eds.).
50: (11)                       *Network Analysis: Methodological Foundations*.
51: (11)                       Springer, 2005.
52: (11)                       <http://books.google.com/books?id=TTNhSm7HYrIC>
53: (4)                 """
54: (4)                 if wiener_index is None:
55: (8)                     wiener_index = nx.wiener_index(G, weight=weight)
56: (4)                 if node is not None:
57: (8)                     after = nx.wiener_index(G.subgraph(set(G) - {node}), weight=weight)
58: (8)                     return wiener_index - after
59: (4)                 vitality = partial(closeness_vitality, G, weight=weight,
wiener_index=wiener_index)
60: (4)                 return {v: vitality(node=v) for v in G}

----------------------------------------

File 61 - voronoi.py:

1: (0)              """Functions for computing the Voronoi cells of a graph."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import groups
4: (0)              __all__ = ["voronoi_cells"]
5: (0)              @nx._dispatchable(edge_attrs="weight")
6: (0)              def voronoi_cells(G, center_nodes, weight="weight"):
7: (4)                  """Returns the Voronoi cells centered at `center_nodes` with respect
8: (4)                  to the shortest-path distance metric.
9: (4)                  If $C$ is a set of nodes in the graph and $c$ is an element of $C$,
10: (4)                 the *Voronoi cell* centered at a node $c$ is the set of all nodes
11: (4)                 $v$ that are closer to $c$ than to any other center node in $C$ with
12: (4)                 respect to the shortest-path distance metric. [1]_
13: (4)                 For directed graphs, this will compute the "outward" Voronoi cells,
14: (4)                 as defined in [1]_, in which distance is measured from the center
15: (4)                 nodes to the target node. For the "inward" Voronoi cells, use the
16: (4)                 :meth:`DiGraph.reverse` method to reverse the orientation of the
17: (4)                 edges before invoking this function on the directed graph.
18: (4)                 Parameters
19: (4)                 ----------
```

```
20: (4)                 G : NetworkX graph
21: (4)                 center_nodes : set
22: (8)                     A nonempty set of nodes in the graph `G` that represent the
23: (8)                     center of the Voronoi cells.
24: (4)                 weight : string or function
25: (8)                     The edge attribute (or an arbitrary function) representing the
26: (8)                     weight of an edge. This keyword argument is as described in the
27: (8)                     documentation for :func:`~networkx.multi_source_dijkstra_path`,
28: (8)                     for example.
29: (4)                 Returns
30: (4)                 -------
31: (4)                 dictionary
32: (8)                     A mapping from center node to set of all nodes in the graph
33: (8)                     closer to that center node than to any other center node. The
34: (8)                     keys of the dictionary are the element of `center_nodes`, and
35: (8)                     the values of the dictionary form a partition of the nodes of
36: (8)                     `G`.
37: (4)                 Examples
38: (4)                 --------
39: (4)                 To get only the partition of the graph induced by the Voronoi cells,
40: (4)                 take the collection of all values in the returned dictionary::
41: (8)                     >>> G = nx.path_graph(6)
42: (8)                     >>> center_nodes = {0, 3}
43: (8)                     >>> cells = nx.voronoi_cells(G, center_nodes)
44: (8)                     >>> partition = set(map(frozenset, cells.values()))
45: (8)                     >>> sorted(map(sorted, partition))
46: (8)                     [[0, 1], [2, 3, 4, 5]]
47: (4)                 Raises
48: (4)                 ------
49: (4)                 ValueError
50: (8)                     If `center_nodes` is empty.
51: (4)                 References
52: (4)                 ----------
53: (4)                 .. [1] Erwig, Martin. (2000),"The graph Voronoi diagram with
applications."
54: (8)                     *Networks*, 36: 156--163.
55: (8)                     https://doi.org/10.1002/1097-0037(200010)36:3<156::AID-NET2>3.0.CO;2-L
56: (4)                 """
57: (4)                 paths = nx.multi_source_dijkstra_path(G, center_nodes, weight=weight)
58: (4)                 nearest = {v: p[0] for v, p in paths.items()}
59: (4)                 cells = groups(nearest)
60: (4)                 unreachable = set(G) - set(nearest)
61: (4)                 if unreachable:
62: (8)                     cells["unreachable"] = unreachable
63: (4)                 return cells
```

File 62 - walks.py:

```
1: (0)              """Function for computing walks in a graph.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              __all__ = ["number_of_walks"]
5: (0)              @nx._dispatchable
6: (0)              def number_of_walks(G, walk_length):
7: (4)                  """Returns the number of walks connecting each pair of nodes in `G`
8: (4)                  A *walk* is a sequence of nodes in which each adjacent pair of nodes
9: (4)                  in the sequence is adjacent in the graph. A walk can repeat the same
10: (4)                 edge and go in the opposite direction just as people can walk on a
11: (4)                 set of paths, but standing still is not counted as part of the walk.
12: (4)                 This function only counts the walks with `walk_length` edges. Note that
13: (4)                 the number of nodes in the walk sequence is one more than `walk_length`.
14: (4)                 The number of walks can grow very quickly on a larger graph
15: (4)                 and with a larger walk length.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G : NetworkX graph
19: (4)                 walk_length : int
```

```
20: (8)                 A nonnegative integer representing the length of a walk.
21: (4)             Returns
22: (4)             -------
23: (4)             dict
24: (8)                 A dictionary of dictionaries in which outer keys are source
25: (8)                 nodes, inner keys are target nodes, and inner values are the
26: (8)                 number of walks of length `walk_length` connecting those nodes.
27: (4)             Raises
28: (4)             ------
29: (4)             ValueError
30: (8)                 If `walk_length` is negative
31: (4)             Examples
32: (4)             --------
33: (4)             >>> G = nx.Graph([(0, 1), (1, 2)])
34: (4)             >>> walks = nx.number_of_walks(G, 2)
35: (4)             >>> walks
36: (4)             {0: {0: 1, 1: 0, 2: 1}, 1: {0: 0, 1: 2, 2: 0}, 2: {0: 1, 1: 0, 2: 1}}
37: (4)             >>> total_walks = sum(sum(tgts.values()) for _, tgts in walks.items())
38: (4)             You can also get the number of walks from a specific source node using the
39: (4)             returned dictionary. For example, number of walks of length 1 from node 0
40: (4)             can be found as follows:
41: (4)             >>> walks = nx.number_of_walks(G, 1)
42: (4)             >>> walks[0]
43: (4)             {0: 0, 1: 1, 2: 0}
44: (4)             >>> sum(walks[0].values())  # walks from 0 of length 1
45: (4)             1
46: (4)             Similarly, a target node can also be specified:
47: (4)             >>> walks[0][1]
48: (4)             1
49: (4)             """
50: (4)             import numpy as np
51: (4)             if walk_length < 0:
52: (8)                 raise ValueError(f"`walk_length` cannot be negative: {walk_length}")
53: (4)             A = nx.adjacency_matrix(G, weight=None)
54: (4)             power = np.linalg.matrix_power(A.toarray(), walk_length)
55: (4)             result = {
56: (8)                 u: {v: power.item(u_idx, v_idx) for v_idx, v in enumerate(G)}
57: (8)                 for u_idx, u in enumerate(G)
58: (4)             }
59: (4)             return result
```

File 63 - wiener.py:

```
1: (0)          """Functions related to the Wiener Index of a graph.
2: (0)          The Wiener Index is a topological measure of a graph
3: (0)          related to the distance between nodes and their degree.
4: (0)          The Schultz Index and Gutman Index are similar measures.
5: (0)          They are used categorize molecules via the network of
6: (0)          atoms connected by chemical bonds. The indices are
7: (0)          correlated with functional aspects of the molecules.
8: (0)          References
9: (0)          ----------
10: (0)         .. [1] `Wikipedia: Wiener Index <https://en.wikipedia.org/wiki/Wiener_index>`_
11: (0)         .. [2] M.V. Diudeaa and I. Gutman, Wiener-Type Topological Indices,
12: (7)                Croatica Chemica Acta, 71 (1998), 21-51.
13: (7)                https://hrcak.srce.hr/132323
14: (0)         """
15: (0)         import itertools as it
16: (0)         import networkx as nx
17: (0)         __all__ = ["wiener_index", "schultz_index", "gutman_index"]
18: (0)         @nx._dispatchable(edge_attrs="weight")
19: (0)         def wiener_index(G, weight=None):
20: (4)             """Returns the Wiener index of the given graph.
21: (4)             The *Wiener index* of a graph is the sum of the shortest-path
22: (4)             (weighted) distances between each pair of reachable nodes.
23: (4)             For pairs of nodes in undirected graphs, only one orientation
24: (4)             of the pair is counted.
```

```
 25: (4)                Parameters
 26: (4)                ----------
 27: (4)                G : NetworkX graph
 28: (4)                weight : string or None, optional (default: None)
 29: (8)                    If None, every edge has weight 1.
 30: (8)                    If a string, use this edge attribute as the edge weight.
 31: (8)                    Any edge attribute not present defaults to 1.
 32: (8)                    The edge weights are used to computing shortest-path distances.
 33: (4)                Returns
 34: (4)                -------
 35: (4)                number
 36: (8)                    The Wiener index of the graph `G`.
 37: (4)                Raises
 38: (4)                ------
 39: (4)                NetworkXError
 40: (8)                    If the graph `G` is not connected.
 41: (4)                Notes
 42: (4)                -----
 43: (4)                If a pair of nodes is not reachable, the distance is assumed to be
 44: (4)                infinity. This means that for graphs that are not
 45: (4)                strongly-connected, this function returns ``inf``.
 46: (4)                The Wiener index is not usually defined for directed graphs, however
 47: (4)                this function uses the natural generalization of the Wiener index to
 48: (4)                directed graphs.
 49: (4)                Examples
 50: (4)                --------
 51: (4)                The Wiener index of the (unweighted) complete graph on *n* nodes
 52: (4)                equals the number of pairs of the *n* nodes, since each pair of
 53: (4)                nodes is at distance one::
 54: (8)                    >>> n = 10
 55: (8)                    >>> G = nx.complete_graph(n)
 56: (8)                    >>> nx.wiener_index(G) == n * (n - 1) / 2
 57: (8)                    True
 58: (4)                Graphs that are not strongly-connected have infinite Wiener index::
 59: (8)                    >>> G = nx.empty_graph(2)
 60: (8)                    >>> nx.wiener_index(G)
 61: (8)                    inf
 62: (4)                References
 63: (4)                ----------
 64: (4)                .. [1] `Wikipedia: Wiener Index
<https://en.wikipedia.org/wiki/Wiener_index>`_
 65: (4)                """
 66: (4)                connected = nx.is_strongly_connected(G) if G.is_directed() else
nx.is_connected(G)
 67: (4)                if not connected:
 68: (8)                    return float("inf")
 69: (4)                spl = nx.shortest_path_length(G, weight=weight)
 70: (4)                total = sum(it.chain.from_iterable(nbrs.values() for node, nbrs in spl))
 71: (4)                return total if G.is_directed() else total / 2
 72: (0)            @nx.utils.not_implemented_for("directed")
 73: (0)            @nx.utils.not_implemented_for("multigraph")
 74: (0)            @nx._dispatchable(edge_attrs="weight")
 75: (0)            def schultz_index(G, weight=None):
 76: (4)                r"""Returns the Schultz Index (of the first kind) of `G`
 77: (4)                The *Schultz Index* [3]_ of a graph is the sum over all node pairs of
 78: (4)                distances times the sum of degrees. Consider an undirected graph `G`.
 79: (4)                For each node pair ``(u, v)`` compute ``dist(u, v) * (deg(u) + deg(v)``
 80: (4)                where ``dist`` is the shortest path length between two nodes and ``deg``
 81: (4)                is the degree of a node.
 82: (4)                The Schultz Index is the sum of these quantities over all (unordered)
 83: (4)                pairs of nodes.
 84: (4)                Parameters
 85: (4)                ----------
 86: (4)                G : NetworkX graph
 87: (8)                    The undirected graph of interest.
 88: (4)                weight : string or None, optional (default: None)
 89: (8)                    If None, every edge has weight 1.
 90: (8)                    If a string, use this edge attribute as the edge weight.
 91: (8)                    Any edge attribute not present defaults to 1.
```

```
92: (8)                     The edge weights are used to computing shortest-path distances.
93: (4)                 Returns
94: (4)                 -------
95: (4)                 number
96: (8)                     The first kind of Schultz Index of the graph `G`.
97: (4)                 Examples
98: (4)                 --------
99: (4)                 The Schultz Index of the (unweighted) complete graph on *n* nodes
100: (4)                equals the number of pairs of the *n* nodes times ``2 * (n - 1)``,
101: (4)                since each pair of nodes is at distance one and the sum of degree
102: (4)                of two nodes is ``2 * (n - 1)``.
103: (4)                >>> n = 10
104: (4)                >>> G = nx.complete_graph(n)
105: (4)                >>> nx.schultz_index(G) == (n * (n - 1) / 2) * (2 * (n - 1))
106: (4)                True
107: (4)                Graph that is disconnected
108: (4)                >>> nx.schultz_index(nx.empty_graph(2))
109: (4)                inf
110: (4)                References
111: (4)                ----------
112: (4)                .. [1] I. Gutman, Selected properties of the Schultz molecular topological
index,
113: (11)                       J. Chem. Inf. Comput. Sci. 34 (1994), 1087–1089.
114: (11)                       https://doi.org/10.1021/ci00021a009
115: (4)                .. [2] M.V. Diudeaa and I. Gutman, Wiener-Type Topological Indices,
116: (11)                       Croatica Chemica Acta, 71 (1998), 21-51.
117: (11)                       https://hrcak.srce.hr/132323
118: (4)                .. [3] H. P. Schultz, Topological organic chemistry. 1.
119: (11)                       Graph theory and topological indices of alkanes,i
120: (11)                       J. Chem. Inf. Comput. Sci. 29 (1989), 239–257.
121: (4)                """
122: (4)                if not nx.is_connected(G):
123: (8)                    return float("inf")
124: (4)                spl = nx.shortest_path_length(G, weight=weight)
125: (4)                d = dict(G.degree, weight=weight)
126: (4)                return sum(dist * (d[u] + d[v]) for u, info in spl for v, dist in
info.items()) / 2
127: (0)            @nx.utils.not_implemented_for("directed")
128: (0)            @nx.utils.not_implemented_for("multigraph")
129: (0)            @nx._dispatchable(edge_attrs="weight")
130: (0)            def gutman_index(G, weight=None):
131: (4)                r"""Returns the Gutman Index for the graph `G`.
132: (4)                The *Gutman Index* measures the topology of networks, especially for
molecule
133: (4)                networks of atoms connected by bonds [1]_. It is also called the Schultz
Index
134: (4)                of the second kind [2]_.
135: (4)                Consider an undirected graph `G` with node set ``V``.
136: (4)                The Gutman Index of a graph is the sum over all (unordered) pairs of nodes
137: (4)                of nodes ``(u, v)``, with distance ``dist(u, v)`` and degrees ``deg(u)``
138: (4)                and ``deg(v)``, of ``dist(u, v) * deg(u) * deg(v)``
139: (4)                Parameters
140: (4)                ----------
141: (4)                G : NetworkX graph
142: (4)                weight : string or None, optional (default: None)
143: (8)                    If None, every edge has weight 1.
144: (8)                    If a string, use this edge attribute as the edge weight.
145: (8)                    Any edge attribute not present defaults to 1.
146: (8)                    The edge weights are used to computing shortest-path distances.
147: (4)                Returns
148: (4)                -------
149: (4)                number
150: (8)                    The Gutman Index of the graph `G`.
151: (4)                Examples
152: (4)                --------
153: (4)                The Gutman Index of the (unweighted) complete graph on *n* nodes
154: (4)                equals the number of pairs of the *n* nodes times ``(n - 1) * (n - 1)``,
155: (4)                since each pair of nodes is at distance one and the product of degree of
two
```

```
156: (4)                vertices is ``(n - 1) * (n - 1)``.
157: (4)                >>> n = 10
158: (4)                >>> G = nx.complete_graph(n)
159: (4)                >>> nx.gutman_index(G) == (n * (n - 1) / 2) * ((n - 1) * (n - 1))
160: (4)                True
161: (4)                Graphs that are disconnected
162: (4)                >>> G = nx.empty_graph(2)
163: (4)                >>> nx.gutman_index(G)
164: (4)                inf
165: (4)                References
166: (4)                ----------
167: (4)                .. [1] M.V. Diudeaa and I. Gutman, Wiener-Type Topological Indices,
168: (11)                      Croatica Chemica Acta, 71 (1998), 21-51.
169: (11)                      https://hrcak.srce.hr/132323
170: (4)                .. [2] I. Gutman, Selected properties of the Schultz molecular topological
index,
171: (11)                      J. Chem. Inf. Comput. Sci. 34 (1994), 1087–1089.
172: (11)                      https://doi.org/10.1021/ci00021a009
173: (4)                """
174: (4)                if not nx.is_connected(G):
175: (8)                    return float("inf")
176: (4)                spl = nx.shortest_path_length(G, weight=weight)
177: (4)                d = dict(G.degree, weight=weight)
178: (4)                return sum(dist * d[u] * d[v] for u, vinfo in spl for v, dist in
vinfo.items()) / 2
```

File 64 - __init__.py:

```
1: (0)              """Approximations of graph properties and Heuristic methods for optimization.
2: (0)              The functions in this class are not imported into the top-level ``networkx``
3: (0)              namespace so the easiest way to use them is with::
4: (4)                  >>> from networkx.algorithms import approximation
5: (0)              Another option is to import the specific function with
6: (0)              ``from networkx.algorithms.approximation import function_name``.
7: (0)              """
8: (0)              from networkx.algorithms.approximation.clustering_coefficient import *
9: (0)              from networkx.algorithms.approximation.clique import *
10: (0)             from networkx.algorithms.approximation.connectivity import *
11: (0)             from networkx.algorithms.approximation.distance_measures import *
12: (0)             from networkx.algorithms.approximation.dominating_set import *
13: (0)             from networkx.algorithms.approximation.kcomponents import *
14: (0)             from networkx.algorithms.approximation.matching import *
15: (0)             from networkx.algorithms.approximation.ramsey import *
16: (0)             from networkx.algorithms.approximation.steinertree import *
17: (0)             from networkx.algorithms.approximation.traveling_salesman import *
18: (0)             from networkx.algorithms.approximation.treewidth import *
19: (0)             from networkx.algorithms.approximation.vertex_cover import *
20: (0)             from networkx.algorithms.approximation.maxcut import *
```

File 65 - clique.py:

```
1: (0)              """Functions for computing large cliques and maximum independent sets."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.approximation import ramsey
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = [
6: (4)                  "clique_removal",
7: (4)                  "max_clique",
8: (4)                  "large_clique_size",
9: (4)                  "maximum_independent_set",
10: (0)             ]
11: (0)             @not_implemented_for("directed")
12: (0)             @not_implemented_for("multigraph")
13: (0)             @nx._dispatchable
14: (0)             def maximum_independent_set(G):
```

```
 15: (4)                 """Returns an approximate maximum independent set.
 16: (4)                 Independent set or stable set is a set of vertices in a graph, no two of
 17: (4)                 which are adjacent. That is, it is a set I of vertices such that for every
 18: (4)                 two vertices in I, there is no edge connecting the two. Equivalently, each
 19: (4)                 edge in the graph has at most one endpoint in I. The size of an
independent
 20: (4)                 set is the number of vertices it contains [1]_.
 21: (4)                 A maximum independent set is a largest independent set for a given graph G
 22: (4)                 and its size is denoted $\\alpha(G)$. The problem of finding such a set is
called
 23: (4)                 the maximum independent set problem and is an NP-hard optimization
problem.
 24: (4)                 As such, it is unlikely that there exists an efficient algorithm for
finding
 25: (4)                 a maximum independent set of a graph.
 26: (4)                 The Independent Set algorithm is based on [2]_.
 27: (4)                 Parameters
 28: (4)                 ----------
 29: (4)                 G : NetworkX graph
 30: (8)                     Undirected graph
 31: (4)                 Returns
 32: (4)                 -------
 33: (4)                 iset : Set
 34: (8)                     The apx-maximum independent set
 35: (4)                 Examples
 36: (4)                 --------
 37: (4)                 >>> G = nx.path_graph(10)
 38: (4)                 >>> nx.approximation.maximum_independent_set(G)
 39: (4)                 {0, 2, 4, 6, 9}
 40: (4)                 Raises
 41: (4)                 ------
 42: (4)                 NetworkXNotImplemented
 43: (8)                     If the graph is directed or is a multigraph.
 44: (4)                 Notes
 45: (4)                 -----
 46: (4)                 Finds the $O(|V|/(log|V|)^2)$ apx of independent set in the worst case.
 47: (4)                 References
 48: (4)                 ----------
 49: (4)                 .. [1] `Wikipedia: Independent set
 50: (8)                     <https://en.wikipedia.org/wiki/Independent_set_(graph_theory)>`_
 51: (4)                 .. [2] Boppana, R., & Halldórsson, M. M. (1992).
 52: (7)                    Approximating maximum independent sets by excluding subgraphs.
 53: (7)                    BIT Numerical Mathematics, 32(2), 180–196. Springer.
 54: (4)                 """
 55: (4)                 iset, _ = clique_removal(G)
 56: (4)                 return iset
 57: (0)             @not_implemented_for("directed")
 58: (0)             @not_implemented_for("multigraph")
 59: (0)             @nx._dispatchable
 60: (0)             def max_clique(G):
 61: (4)                 r"""Find the Maximum Clique
 62: (4)                 Finds the $O(|V|/(log|V|)^2)$ apx of maximum clique/independent set
 63: (4)                 in the worst case.
 64: (4)                 Parameters
 65: (4)                 ----------
 66: (4)                 G : NetworkX graph
 67: (8)                     Undirected graph
 68: (4)                 Returns
 69: (4)                 -------
 70: (4)                 clique : set
 71: (8)                     The apx-maximum clique of the graph
 72: (4)                 Examples
 73: (4)                 --------
 74: (4)                 >>> G = nx.path_graph(10)
 75: (4)                 >>> nx.approximation.max_clique(G)
 76: (4)                 {8, 9}
 77: (4)                 Raises
 78: (4)                 ------
 79: (4)                 NetworkXNotImplemented
```

```
80: (8)                     If the graph is directed or is a multigraph.
81: (4)                 Notes
82: (4)                 -----
83: (4)                 A clique in an undirected graph G = (V, E) is a subset of the vertex set
84: (4)                 `C \subseteq V` such that for every two vertices in C there exists an edge
85: (4)                 connecting the two. This is equivalent to saying that the subgraph
86: (4)                 induced by C is complete (in some cases, the term clique may also refer
87: (4)                 to the subgraph).
88: (4)                 A maximum clique is a clique of the largest possible size in a given
graph.
89: (4)                 The clique number `\omega(G)` of a graph G is the number of
90: (4)                 vertices in a maximum clique in G. The intersection number of
91: (4)                 G is the smallest number of cliques that together cover all edges of G.
92: (4)                 https://en.wikipedia.org/wiki/Maximum_clique
93: (4)                 References
94: (4)                 ----------
95: (4)                 .. [1] Boppana, R., & Halldórsson, M. M. (1992).
96: (8)                     Approximating maximum independent sets by excluding subgraphs.
97: (8)                     BIT Numerical Mathematics, 32(2), 180–196. Springer.
98: (8)                     doi:10.1007/BF01994876
99: (4)                 """
100: (4)                cgraph = nx.complement(G)
101: (4)                iset, _ = clique_removal(cgraph)
102: (4)                return iset
103: (0)            @not_implemented_for("directed")
104: (0)            @not_implemented_for("multigraph")
105: (0)            @nx._dispatchable
106: (0)            def clique_removal(G):
107: (4)                r"""Repeatedly remove cliques from the graph.
108: (4)                Results in a $O(|V|/(\log |V|)^2)$ approximation of maximum clique
109: (4)                and independent set. Returns the largest independent set found, along
110: (4)                with found maximal cliques.
111: (4)                Parameters
112: (4)                ----------
113: (4)                G : NetworkX graph
114: (8)                    Undirected graph
115: (4)                Returns
116: (4)                -------
117: (4)                max_ind_cliques : (set, list) tuple
118: (8)                    2-tuple of Maximal Independent Set and list of maximal cliques (sets).
119: (4)                Examples
120: (4)                --------
121: (4)                >>> G = nx.path_graph(10)
122: (4)                >>> nx.approximation.clique_removal(G)
123: (4)                ({0, 2, 4, 6, 9}, [{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}])
124: (4)                Raises
125: (4)                ------
126: (4)                NetworkXNotImplemented
127: (8)                    If the graph is directed or is a multigraph.
128: (4)                References
129: (4)                ----------
130: (4)                .. [1] Boppana, R., & Halldórsson, M. M. (1992).
131: (8)                    Approximating maximum independent sets by excluding subgraphs.
132: (8)                    BIT Numerical Mathematics, 32(2), 180–196. Springer.
133: (4)                """
134: (4)                graph = G.copy()
135: (4)                c_i, i_i = ramsey.ramsey_R2(graph)
136: (4)                cliques = [c_i]
137: (4)                isets = [i_i]
138: (4)                while graph:
139: (8)                    graph.remove_nodes_from(c_i)
140: (8)                    c_i, i_i = ramsey.ramsey_R2(graph)
141: (8)                    if c_i:
142: (12)                       cliques.append(c_i)
143: (8)                    if i_i:
144: (12)                       isets.append(i_i)
145: (4)                maxiset = max(isets, key=len)
146: (4)                return maxiset, cliques
147: (0)            @not_implemented_for("directed")
```

```
148: (0)            @not_implemented_for("multigraph")
149: (0)            @nx._dispatchable
150: (0)            def large_clique_size(G):
151: (4)                """Find the size of a large clique in a graph.
152: (4)                A *clique* is a subset of nodes in which each pair of nodes is
153: (4)                adjacent. This function is a heuristic for finding the size of a
154: (4)                large clique in the graph.
155: (4)                Parameters
156: (4)                ----------
157: (4)                G : NetworkX graph
158: (4)                Returns
159: (4)                -------
160: (4)                k: integer
161: (7)                   The size of a large clique in the graph.
162: (4)                Examples
163: (4)                --------
164: (4)                >>> G = nx.path_graph(10)
165: (4)                >>> nx.approximation.large_clique_size(G)
166: (4)                2
167: (4)                Raises
168: (4)                ------
169: (4)                NetworkXNotImplemented
170: (8)                    If the graph is directed or is a multigraph.
171: (4)                Notes
172: (4)                -----
173: (4)                This implementation is from [1]_. Its worst case time complexity is
174: (4)                :math:`O(n d^2)`, where *n* is the number of nodes in the graph and
175: (4)                *d* is the maximum degree.
176: (4)                This function is a heuristic, which means it may work well in
177: (4)                practice, but there is no rigorous mathematical guarantee on the
178: (4)                ratio between the returned number and the actual largest clique size
179: (4)                in the graph.
180: (4)                References
181: (4)                ----------
182: (4)                .. [1] Pattabiraman, Bharath, et al.
183: (7)                   "Fast Algorithms for the Maximum Clique Problem on Massive Graphs
184: (7)                   with Applications to Overlapping Community Detection."
185: (7)                   *Internet Mathematics* 11.4-5 (2015): 421--448.
186: (7)                   <https://doi.org/10.1080/15427951.2014.986778>
187: (4)                See also
188: (4)                --------
189: (4)                :func:`networkx.algorithms.approximation.clique.max_clique`
190: (8)                    A function that returns an approximate maximum clique with a
191: (8)                    guarantee on the approximation ratio.
192: (4)                :mod:`networkx.algorithms.clique`
193: (8)                    Functions for finding the exact maximum clique in a graph.
194: (4)                """
195: (4)                degrees = G.degree
196: (4)                def _clique_heuristic(G, U, size, best_size):
197: (8)                    if not U:
198: (12)                       return max(best_size, size)
199: (8)                    u = max(U, key=degrees)
200: (8)                    U.remove(u)
201: (8)                    N_prime = {v for v in G[u] if degrees[v] >= best_size}
202: (8)                    return _clique_heuristic(G, U & N_prime, size + 1, best_size)
203: (4)                best_size = 0
204: (4)                nodes = (u for u in G if degrees[u] >= best_size)
205: (4)                for u in nodes:
206: (8)                    neighbors = {v for v in G[u] if degrees[v] >= best_size}
207: (8)                    best_size = _clique_heuristic(G, neighbors, 1, best_size)
208: (4)                return best_size

----------------------------------------

File 66 - connectivity.py:

1: (0)              """ Fast approximation for node connectivity
2: (0)              """
3: (0)              import itertools
```

```
4: (0)              from operator import itemgetter
5: (0)              import networkx as nx
6: (0)              __all__ = [
7: (4)                  "local_node_connectivity",
8: (4)                  "node_connectivity",
9: (4)                  "all_pairs_node_connectivity",
10: (0)             ]
11: (0)             @nx._dispatchable(name="approximate_local_node_connectivity")
12: (0)             def local_node_connectivity(G, source, target, cutoff=None):
13: (4)                 """Compute node connectivity between source and target.
14: (4)                 Pairwise or local node connectivity between two distinct and nonadjacent
15: (4)                 nodes is the minimum number of nodes that must be removed (minimum
16: (4)                 separating cutset) to disconnect them. By Menger's theorem, this is equal
17: (4)                 to the number of node independent paths (paths that share no nodes other
18: (4)                 than source and target). Which is what we compute in this function.
19: (4)                 This algorithm is a fast approximation that gives an strict lower
20: (4)                 bound on the actual number of node independent paths between two nodes
[1]_.
21: (4)                 It works for both directed and undirected graphs.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 G : NetworkX graph
25: (4)                 source : node
26: (8)                     Starting node for node connectivity
27: (4)                 target : node
28: (8)                     Ending node for node connectivity
29: (4)                 cutoff : integer
30: (8)                     Maximum node connectivity to consider. If None, the minimum degree
31: (8)                     of source or target is used as a cutoff. Default value None.
32: (4)                 Returns
33: (4)                 -------
34: (4)                 k: integer
35: (7)                    pairwise node connectivity
36: (4)                 Examples
37: (4)                 --------
38: (4)                 >>> # Platonic octahedral graph has node connectivity 4
39: (4)                 >>> # for each non adjacent node pair
40: (4)                 >>> from networkx.algorithms import approximation as approx
41: (4)                 >>> G = nx.octahedral_graph()
42: (4)                 >>> approx.local_node_connectivity(G, 0, 5)
43: (4)                 4
44: (4)                 Notes
45: (4)                 -----
46: (4)                 This algorithm [1]_ finds node independents paths between two nodes by
47: (4)                 computing their shortest path using BFS, marking the nodes of the path
48: (4)                 found as 'used' and then searching other shortest paths excluding the
49: (4)                 nodes marked as used until no more paths exist. It is not exact because
50: (4)                 a shortest path could use nodes that, if the path were longer, may belong
51: (4)                 to two different node independent paths. Thus it only guarantees an
52: (4)                 strict lower bound on node connectivity.
53: (4)                 Note that the authors propose a further refinement, losing accuracy and
54: (4)                 gaining speed, which is not implemented yet.
55: (4)                 See also
56: (4)                 --------
57: (4)                 all_pairs_node_connectivity
58: (4)                 node_connectivity
59: (4)                 References
60: (4)                 ----------
61: (4)                 .. [1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for
62: (8)                     Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035
63: (8)                     http://eclectic.ss.uci.edu/~drwhite/working.pdf
64: (4)                 """
65: (4)                 if target == source:
66: (8)                     raise nx.NetworkXError("source and target have to be different
nodes.")
67: (4)                 if G.is_directed():
68: (8)                     possible = min(G.out_degree(source), G.in_degree(target))
69: (4)                 else:
70: (8)                     possible = min(G.degree(source), G.degree(target))
```

```
(4)                 K = 0
(4)                 if not possible:
(8)                     return K
(4)                 if cutoff is None:
(8)                     cutoff = float("inf")
(4)                 exclude = set()
(4)                 for i in range(min(possible, cutoff)):
(8)                     try:
(12)                        path = _bidirectional_shortest_path(G, source, target, exclude)
(12)                        exclude.update(set(path))
(12)                        K += 1
(8)                     except nx.NetworkXNoPath:
(12)                        break
(4)                 return K
(0)             @nx._dispatchable(name="approximate_node_connectivity")
(0)             def node_connectivity(G, s=None, t=None):
(4)                 r"""Returns an approximation for node connectivity for a graph or digraph
G.
(4)                 Node connectivity is equal to the minimum number of nodes that
(4)                 must be removed to disconnect G or render it trivial. By Menger's theorem,
(4)                 this is equal to the number of node independent paths (paths that
(4)                 share no nodes other than source and target).
(4)                 If source and target nodes are provided, this function returns the
(4)                 local node connectivity: the minimum number of nodes that must be
(4)                 removed to break all paths from source to target in G.
(4)                 This algorithm is based on a fast approximation that gives an strict lower
(4)                 bound on the actual number of node independent paths between two nodes
[1]_.
(4)                 It works for both directed and undirected graphs.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Undirected graph
(4)                 s : node
(8)                     Source node. Optional. Default value: None.
(4)                 t : node
(8)                     Target node. Optional. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 K : integer
(8)                     Node connectivity of G, or local node connectivity if source
(8)                     and target are provided.
(4)                 Examples
(4)                 --------
(4)                 >>> # Platonic octahedral graph is 4-node-connected
(4)                 >>> from networkx.algorithms import approximation as approx
(4)                 >>> G = nx.octahedral_graph()
(4)                 >>> approx.node_connectivity(G)
(4)                 4
(4)                 Notes
(4)                 -----
(4)                 This algorithm [1]_ finds node independents paths between two nodes by
(4)                 computing their shortest path using BFS, marking the nodes of the path
(4)                 found as 'used' and then searching other shortest paths excluding the
(4)                 nodes marked as used until no more paths exist. It is not exact because
(4)                 a shortest path could use nodes that, if the path were longer, may belong
(4)                 to two different node independent paths. Thus it only guarantees an
(4)                 strict lower bound on node connectivity.
(4)                 See also
(4)                 --------
(4)                 all_pairs_node_connectivity
(4)                 local_node_connectivity
(4)                 References
(4)                 ----------
(4)                 .. [1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for
(8)                     Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035
(8)                     http://eclectic.ss.uci.edu/~drwhite/working.pdf
(4)                 """
(4)                 if (s is not None and t is None) or (s is None and t is not None):
```

```
(8)                 raise nx.NetworkXError("Both source and target must be specified.")
(4)             if s is not None and t is not None:
(8)                 if s not in G:
(12)                    raise nx.NetworkXError(f"node {s} not in graph")
(8)                 if t not in G:
(12)                    raise nx.NetworkXError(f"node {t} not in graph")
(8)                 return local_node_connectivity(G, s, t)
(4)             if G.is_directed():
(8)                 connected_func = nx.is_weakly_connected
(8)                 iter_func = itertools.permutations
(8)                 def neighbors(v):
(12)                    return itertools.chain(G.predecessors(v), G.successors(v))
(4)             else:
(8)                 connected_func = nx.is_connected
(8)                 iter_func = itertools.combinations
(8)                 neighbors = G.neighbors
(4)             if not connected_func(G):
(8)                 return 0
(4)             v, minimum_degree = min(G.degree(), key=itemgetter(1))
(4)             K = minimum_degree
(4)             for w in set(G) - set(neighbors(v)) - {v}:
(8)                 K = min(K, local_node_connectivity(G, v, w, cutoff=K))
(4)             for x, y in iter_func(neighbors(v), 2):
(8)                 if y not in G[x] and x != y:
(12)                    K = min(K, local_node_connectivity(G, x, y, cutoff=K))
(4)             return K
(0)         @nx._dispatchable(name="approximate_all_pairs_node_connectivity")
(0)         def all_pairs_node_connectivity(G, nbunch=None, cutoff=None):
(4)             """Compute node connectivity between all pairs of nodes.
(4)             Pairwise or local node connectivity between two distinct and nonadjacent
(4)             nodes is the minimum number of nodes that must be removed (minimum
(4)             separating cutset) to disconnect them. By Menger's theorem, this is equal
(4)             to the number of node independent paths (paths that share no nodes other
(4)             than source and target). Which is what we compute in this function.
(4)             This algorithm is a fast approximation that gives an strict lower
(4)             bound on the actual number of node independent paths between two nodes
[1]_.
(4)             It works for both directed and undirected graphs.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             nbunch: container
(8)                 Container of nodes. If provided node connectivity will be computed
(8)                 only over pairs of nodes in nbunch.
(4)             cutoff : integer
(8)                 Maximum node connectivity to consider. If None, the minimum degree
(8)                 of source or target is used as a cutoff in each pair of nodes.
(8)                 Default value None.
(4)             Returns
(4)             -------
(4)             K : dictionary
(8)                 Dictionary, keyed by source and target, of pairwise node connectivity
(4)             Examples
(4)             --------
(4)             A 3 node cycle with one extra node attached has connectivity 2 between all
(4)             nodes in the cycle and connectivity 1 between the extra node and the rest:
(4)             >>> G = nx.cycle_graph(3)
(4)             >>> G.add_edge(2, 3)
(4)             >>> import pprint  # for nice dictionary formatting
(4)             >>> pprint.pprint(nx.all_pairs_node_connectivity(G))
(4)             {0: {1: 2, 2: 2, 3: 1},
(5)              1: {0: 2, 2: 2, 3: 1},
(5)              2: {0: 2, 1: 2, 3: 1},
(5)              3: {0: 1, 1: 1, 2: 1}}
(4)             See Also
(4)             --------
(4)             local_node_connectivity
(4)             node_connectivity
(4)             References
```

```
(4)                 ----------
(4)                 .. [1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for
(8)                     Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035
(8)                     http://eclectic.ss.uci.edu/~drwhite/working.pdf
(4)                 """
(4)                 if nbunch is None:
(8)                     nbunch = G
(4)                 else:
(8)                     nbunch = set(nbunch)
(4)                 directed = G.is_directed()
(4)                 if directed:
(8)                     iter_func = itertools.permutations
(4)                 else:
(8)                     iter_func = itertools.combinations
(4)                 all_pairs = {n: {} for n in nbunch}
(4)                 for u, v in iter_func(nbunch, 2):
(8)                     k = local_node_connectivity(G, u, v, cutoff=cutoff)
(8)                     all_pairs[u][v] = k
(8)                     if not directed:
(12)                        all_pairs[v][u] = k
(4)                 return all_pairs
(0)             def _bidirectional_shortest_path(G, source, target, exclude):
(4)                 """Returns shortest path between source and target ignoring nodes in the
(4)                 container 'exclude'.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node
(8)                     Starting node for path
(4)                 target : node
(8)                     Ending node for path
(4)                 exclude: container
(8)                     Container for nodes to exclude from the search for shortest paths
(4)                 Returns
(4)                 -------
(4)                 path: list
(8)                     Shortest path between source and target ignoring nodes in 'exclude'
(4)                 Raises
(4)                 ------
(4)                 NetworkXNoPath
(8)                     If there is no path or if nodes are adjacent and have only one path
(8)                     between them
(4)                 Notes
(4)                 -----
(4)                 This function and its helper are originally from
(4)                 networkx.algorithms.shortest_paths.unweighted and are modified to
(4)                 accept the extra parameter 'exclude', which is a container for nodes
(4)                 already used in other paths that should be ignored.
(4)                 References
(4)                 ----------
(4)                 .. [1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for
(8)                     Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035
(8)                     http://eclectic.ss.uci.edu/~drwhite/working.pdf
(4)                 """
(4)                 results = _bidirectional_pred_succ(G, source, target, exclude)
(4)                 pred, succ, w = results
(4)                 path = []
(4)                 while w is not None:
(8)                     path.append(w)
(8)                     w = pred[w]
(4)                 path.reverse()
(4)                 w = succ[path[-1]]
(4)                 while w is not None:
(8)                     path.append(w)
(8)                     w = succ[w]
(4)                 return path
(0)             def _bidirectional_pred_succ(G, source, target, exclude):
(4)                 if G.is_directed():
(8)                     Gpred = G.predecessors
```

```
275: (8)                    Gsucc = G.successors
276: (4)                else:
277: (8)                    Gpred = G.neighbors
278: (8)                    Gsucc = G.neighbors
279: (4)                pred = {source: None}
280: (4)                succ = {target: None}
281: (4)                forward_fringe = [source]
282: (4)                reverse_fringe = [target]
283: (4)                level = 0
284: (4)                while forward_fringe and reverse_fringe:
285: (8)                    level += 1
286: (8)                    if level % 2 != 0:
287: (12)                       this_level = forward_fringe
288: (12)                       forward_fringe = []
289: (12)                       for v in this_level:
290: (16)                           for w in Gsucc(v):
291: (20)                               if w in exclude:
292: (24)                                   continue
293: (20)                               if w not in pred:
294: (24)                                   forward_fringe.append(w)
295: (24)                                   pred[w] = v
296: (20)                               if w in succ:
297: (24)                                   return pred, succ, w  # found path
298: (8)                    else:
299: (12)                       this_level = reverse_fringe
300: (12)                       reverse_fringe = []
301: (12)                       for v in this_level:
302: (16)                           for w in Gpred(v):
303: (20)                               if w in exclude:
304: (24)                                   continue
305: (20)                               if w not in succ:
306: (24)                                   succ[w] = v
307: (24)                                   reverse_fringe.append(w)
308: (20)                               if w in pred:
309: (24)                                   return pred, succ, w  # found path
310: (4)                raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
```

File 67 - clustering_coefficient.py:

```
1: (0)              import networkx as nx
2: (0)              from networkx.utils import not_implemented_for, py_random_state
3: (0)              __all__ = ["average_clustering"]
4: (0)              @not_implemented_for("directed")
5: (0)              @py_random_state(2)
6: (0)              @nx._dispatchable(name="approximate_average_clustering")
7: (0)              def average_clustering(G, trials=1000, seed=None):
8: (4)                  r"""Estimates the average clustering coefficient of G.
9: (4)                  The local clustering of each node in `G` is the fraction of triangles
10: (4)                 that actually exist over all possible triangles in its neighborhood.
11: (4)                 The average clustering coefficient of a graph `G` is the mean of
12: (4)                 local clusterings.
13: (4)                 This function finds an approximate average clustering coefficient
14: (4)                 for G by repeating `n` times (defined in `trials`) the following
15: (4)                 experiment: choose a node at random, choose two of its neighbors
16: (4)                 at random, and check if they are connected. The approximate
17: (4)                 coefficient is the fraction of triangles found over the number
18: (4)                 of trials [1]_.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : NetworkX graph
22: (4)                 trials : integer
23: (8)                     Number of trials to perform (default 1000).
24: (4)                 seed : integer, random_state, or None (default)
25: (8)                     Indicator of random number generation state.
26: (8)                     See :ref:`Randomness<randomness>`.
27: (4)                 Returns
28: (4)                 -------
```

```
 29: (4)                c : float
 30: (8)                    Approximated average clustering coefficient.
 31: (4)                Examples
 32: (4)                --------
 33: (4)                >>> from networkx.algorithms import approximation
 34: (4)                >>> G = nx.erdos_renyi_graph(10, 0.2, seed=10)
 35: (4)                >>> approximation.average_clustering(G, trials=1000, seed=10)
 36: (4)                0.214
 37: (4)                Raises
 38: (4)                ------
 39: (4)                NetworkXNotImplemented
 40: (8)                    If G is directed.
 41: (4)                References
 42: (4)                ----------
 43: (4)                .. [1] Schank, Thomas, and Dorothea Wagner. Approximating clustering
 44: (7)                   coefficient and transitivity. Universität Karlsruhe, Fakultät für
 45: (7)                   Informatik, 2004.
 46: (7)                   https://doi.org/10.5445/IR/1000001239
 47: (4)                """
 48: (4)                n = len(G)
 49: (4)                triangles = 0
 50: (4)                nodes = list(G)
 51: (4)                for i in [int(seed.random() * n) for i in range(trials)]:
 52: (8)                    nbrs = list(G[nodes[i]])
 53: (8)                    if len(nbrs) < 2:
 54: (12)                       continue
 55: (8)                    u, v = seed.sample(nbrs, 2)
 56: (8)                    if u in G[v]:
 57: (12)                       triangles += 1
 58: (4)                return triangles / trials

----------------------------------------

File 68 - dominating_set.py:

 1: (0)             """Functions for finding node and edge dominating sets.
 2: (0)             A `dominating set`_ for an undirected graph *G* with vertex set *V*
 3: (0)             and edge set *E* is a subset *D* of *V* such that every vertex not in
 4: (0)             *D* is adjacent to at least one member of *D*. An `edge dominating set`_
 5: (0)             is a subset *F* of *E* such that every edge not in *F* is
 6: (0)             incident to an endpoint of at least one edge in *F*.
 7: (0)             .. _dominating set: https://en.wikipedia.org/wiki/Dominating_set
 8: (0)             .. _edge dominating set: https://en.wikipedia.org/wiki/Edge_dominating_set
 9: (0)             """
10: (0)             import networkx as nx
11: (0)             from ...utils import not_implemented_for
12: (0)             from ..matching import maximal_matching
13: (0)             __all__ = ["min_weighted_dominating_set", "min_edge_dominating_set"]
14: (0)             @not_implemented_for("directed")
15: (0)             @nx._dispatchable(node_attrs="weight")
16: (0)             def min_weighted_dominating_set(G, weight=None):
17: (4)                 r"""Returns a dominating set that approximates the minimum weight node
18: (4)                 dominating set.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : NetworkX graph
22: (8)                     Undirected graph.
23: (4)                 weight : string
24: (8)                     The node attribute storing the weight of an node. If provided,
25: (8)                     the node attribute with this key must be a number for each
26: (8)                     node. If not provided, each node is assumed to have weight one.
27: (4)                 Returns
28: (4)                 -------
29: (4)                 min_weight_dominating_set : set
30: (8)                     A set of nodes, the sum of whose weights is no more than `(\log
31: (8)                     w(V)) w(V^*)`, where `w(V)` denotes the sum of the weights of
32: (8)                     each node in the graph and `w(V^*)` denotes the sum of the
33: (8)                     weights of each node in the minimum weight dominating set.
34: (4)                 Examples
```

```
    --------
    >>> G = nx.Graph([(0, 1), (0, 4), (1, 4), (1, 2), (2, 3), (3, 4), (2, 5)])
    >>> nx.approximation.min_weighted_dominating_set(G)
    {1, 2, 4}
    Raises
    ------
    NetworkXNotImplemented
        If G is directed.
    Notes
    -----
    This algorithm computes an approximate minimum weighted dominating
    set for the graph `G`. The returned solution has weight `(\log
    w(V)) w(V^*)`, where `w(V)` denotes the sum of the weights of each
    node in the graph and `w(V^*)` denotes the sum of the weights of
    each node in the minimum weight dominating set for the graph.
    This implementation of the algorithm runs in $O(m)$ time, where $m$
    is the number of edges in the graph.
    References
    ----------
    .. [1] Vazirani, Vijay V.
           *Approximation Algorithms*.
           Springer Science & Business Media, 2001.
    """
    if len(G) == 0:
        return set()
    dom_set = set()
    def _cost(node_and_neighborhood):
        """Returns the cost-effectiveness of greedily choosing the given
        node.
        `node_and_neighborhood` is a two-tuple comprising a node and its
        closed neighborhood.
        """
        v, neighborhood = node_and_neighborhood
        return G.nodes[v].get(weight, 1) / len(neighborhood - dom_set)
    vertices = set(G)
    neighborhoods = {v: {v} | set(G[v]) for v in G}
    while vertices:
        dom_node, min_set = min(neighborhoods.items(), key=_cost)
        dom_set.add(dom_node)
        del neighborhoods[dom_node]
        vertices -= min_set
    return dom_set
@nx._dispatchable
def min_edge_dominating_set(G):
    r"""Returns minimum cardinality edge dominating set.
    Parameters
    ----------
    G : NetworkX graph
      Undirected graph
    Returns
    -------
    min_edge_dominating_set : set
      Returns a set of dominating edges whose size is no more than 2 * OPT.
    Examples
    --------
    >>> G = nx.petersen_graph()
    >>> nx.approximation.min_edge_dominating_set(G)
    {(0, 1), (4, 9), (6, 8), (5, 7), (2, 3)}
    Raises
    ------
    ValueError
        If the input graph `G` is empty.
    Notes
    -----
    The algorithm computes an approximate solution to the edge dominating set
    problem. The result is no more than 2 * OPT in terms of size of the set.
    Runtime of the algorithm is $O(|E|)$.
    """
    if not G:
```

```
104: (8)                    raise ValueError("Expected non-empty NetworkX graph!")
105: (4)                return maximal_matching(G)
```

File 69 - distance_measures.py:

```
1: (0)              """Distance measures approximated metrics."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils.decorators import py_random_state
4: (0)              __all__ = ["diameter"]
5: (0)              @py_random_state(1)
6: (0)              @nx._dispatchable(name="approximate_diameter")
7: (0)              def diameter(G, seed=None):
8: (4)                  """Returns a lower bound on the diameter of the graph G.
9: (4)                  The function computes a lower bound on the diameter (i.e., the maximum
eccentricity)
10: (4)                 of a directed or undirected graph G. The procedure used varies depending
on the graph
11: (4)                 being directed or not.
12: (4)                 If G is an `undirected` graph, then the function uses the `2-sweep`
algorithm [1]_.
13: (4)                 The main idea is to pick the farthest node from a random node and return
its eccentricity.
14: (4)                 Otherwise, if G is a `directed` graph, the function uses the `2-dSweep`
algorithm [2]_,
15: (4)                 The procedure starts by selecting a random source node $s$ from which it
performs a
16: (4)                 forward and a backward BFS. Let $a_1$ and $a_2$ be the farthest nodes in
the forward and
17: (4)                 backward cases, respectively. Then, it computes the backward eccentricity
of $a_1$ using
18: (4)                 a backward BFS and the forward eccentricity of $a_2$ using a forward BFS.
19: (4)                 Finally, it returns the best lower bound between the two.
20: (4)                 In both cases, the time complexity is linear with respect to the size of
G.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : NetworkX graph
24: (4)                 seed : integer, random_state, or None (default)
25: (8)                     Indicator of random number generation state.
26: (8)                     See :ref:`Randomness<randomness>`.
27: (4)                 Returns
28: (4)                 -------
29: (4)                 d : integer
30: (7)                    Lower Bound on the Diameter of G
31: (4)                 Examples
32: (4)                 --------
33: (4)                 >>> G = nx.path_graph(10)  # undirected graph
34: (4)                 >>> nx.diameter(G)
35: (4)                 9
36: (4)                 >>> G = nx.cycle_graph(3, create_using=nx.DiGraph)  # directed graph
37: (4)                 >>> nx.diameter(G)
38: (4)                 2
39: (4)                 Raises
40: (4)                 ------
41: (4)                 NetworkXError
42: (8)                     If the graph is empty or
43: (8)                     If the graph is undirected and not connected or
44: (8)                     If the graph is directed and not strongly connected.
45: (4)                 See Also
46: (4)                 --------
47: (4)                 networkx.algorithms.distance_measures.diameter
48: (4)                 References
49: (4)                 ----------
50: (4)                 .. [1] Magnien, Clémence, Matthieu Latapy, and Michel Habib.
51: (7)                    *Fast computation of empirically tight bounds for the diameter of
massive graphs.*
52: (7)                    Journal of Experimental Algorithmics (JEA), 2009.
```

```
53: (7)                     https://arxiv.org/pdf/0904.2728.pdf
54: (4)                 .. [2] Crescenzi, Pierluigi, Roberto Grossi, Leonardo Lanzi, and Andrea
Marino.
55: (7)                     *On computing the diameter of real-world directed (weighted) graphs.*
56: (7)                     International Symposium on Experimental Algorithms. Springer, Berlin,
Heidelberg, 2012.
57: (7)
https://courses.cs.ut.ee/MTAT.03.238/2014_fall/uploads/Main/diameter.pdf
58: (4)                 """
59: (4)                 if not G:
60: (8)                     raise nx.NetworkXError("Expected non-empty NetworkX graph!")
61: (4)                 if G.number_of_nodes() == 1:
62: (8)                     return 0
63: (4)                 if G.is_directed():
64: (8)                     return _two_sweep_directed(G, seed)
65: (4)                 return _two_sweep_undirected(G, seed)
66: (0)             def _two_sweep_undirected(G, seed):
67: (4)                 """Helper function for finding a lower bound on the diameter
68: (8)                     for undirected Graphs.
69: (8)                     The idea is to pick the farthest node from a random node
70: (8)                     and return its eccentricity.
71: (8)                     ``G`` is a NetworkX undirected graph.
72: (4)                 .. note::
73: (8)                     ``seed`` is a random.Random or numpy.random.RandomState instance
74: (4)                 """
75: (4)                 source = seed.choice(list(G))
76: (4)                 distances = nx.shortest_path_length(G, source)
77: (4)                 if len(distances) != len(G):
78: (8)                     raise nx.NetworkXError("Graph not connected.")
79: (4)                 *_, node = distances
80: (4)                 return nx.eccentricity(G, node)
81: (0)             def _two_sweep_directed(G, seed):
82: (4)                 """Helper function for finding a lower bound on the diameter
83: (8)                     for directed Graphs.
84: (8)                     It implements 2-dSweep, the directed version of the 2-sweep algorithm.
85: (8)                     The algorithm follows the following steps.
86: (8)                     1. Select a source node $s$ at random.
87: (8)                     2. Perform a forward BFS from $s$ to select a node $a_1$ at the
maximum
88: (8)                     distance from the source, and compute $LB_1$, the backward
eccentricity of $a_1$.
89: (8)                     3. Perform a backward BFS from $s$ to select a node $a_2$ at the
maximum
90: (8)                     distance from the source, and compute $LB_2$, the forward eccentricity
of $a_2$.
91: (8)                     4. Return the maximum between $LB_1$ and $LB_2$.
92: (8)                     ``G`` is a NetworkX directed graph.
93: (4)                 .. note::
94: (8)                     ``seed`` is a random.Random or numpy.random.RandomState instance
95: (4)                 """
96: (4)                 G_reversed = G.reverse()
97: (4)                 source = seed.choice(list(G))
98: (4)                 forward_distances = nx.shortest_path_length(G, source)
99: (4)                 backward_distances = nx.shortest_path_length(G_reversed, source)
100: (4)                n = len(G)
101: (4)                if len(forward_distances) != n or len(backward_distances) != n:
102: (8)                    raise nx.NetworkXError("DiGraph not strongly connected.")
103: (4)                *_, a_1 = forward_distances
104: (4)                *_, a_2 = backward_distances
105: (4)                return max(nx.eccentricity(G_reversed, a_1), nx.eccentricity(G, a_2))
```

File 70 - kcomponents.py:

```
1: (0)              """ Fast approximation for k-component structure
2: (0)              """
3: (0)              import itertools
4: (0)              from collections import defaultdict
```

```
from collections.abc import Mapping
from functools import cached_property
import networkx as nx
from networkx.algorithms.approximation import local_node_connectivity
from networkx.exception import NetworkXError
from networkx.utils import not_implemented_for
__all__ = ["k_components"]
@not_implemented_for("directed")
@nx._dispatchable(name="approximate_k_components")
def k_components(G, min_density=0.95):
    r"""Returns the approximate k-component structure of a graph G.
    A `k`-component is a maximal subgraph of a graph G that has, at least,
    node connectivity `k`: we need to remove at least `k` nodes to break it
    into more components. `k`-components have an inherent hierarchical
    structure because they are nested in terms of connectivity: a connected
    graph can contain several 2-components, each of which can contain
    one or more 3-components, and so forth.
    This implementation is based on the fast heuristics to approximate
    the `k`-component structure of a graph [1]_. Which, in turn, it is based
on
    a fast approximation algorithm for finding good lower bounds of the number
    of node independent paths between two nodes [2]_.
    Parameters
    ----------
    G : NetworkX graph
        Undirected graph
    min_density : Float
        Density relaxation threshold. Default value 0.95
    Returns
    -------
    k_components : dict
        Dictionary with connectivity level `k` as key and a list of
        sets of nodes that form a k-component of level `k` as values.
    Raises
    ------
    NetworkXNotImplemented
        If G is directed.
    Examples
    --------
    >>> # Petersen graph has 10 nodes and it is triconnected, thus all
    >>> # nodes are in a single component on all three connectivity levels
    >>> from networkx.algorithms import approximation as apxa
    >>> G = nx.petersen_graph()
    >>> k_components = apxa.k_components(G)
    Notes
    -----
    The logic of the approximation algorithm for computing the `k`-component
    structure [1]_ is based on repeatedly applying simple and fast algorithms
    for `k`-cores and biconnected components in order to narrow down the
    number of pairs of nodes over which we have to compute White and Newman's
    approximation algorithm for finding node independent paths [2]_. More
    formally, this algorithm is based on Whitney's theorem, which states
    an inclusion relation among node connectivity, edge connectivity, and
    minimum degree for any graph G. This theorem implies that every
    `k`-component is nested inside a `k`-edge-component, which in turn,
    is contained in a `k`-core. Thus, this algorithm computes node independent
    paths among pairs of nodes in each biconnected part of each `k`-core,
    and repeats this procedure for each `k` from 3 to the maximal core number
    of a node in the input graph.
    Because, in practice, many nodes of the core of level `k` inside a
    bicomponent actually are part of a component of level k, the auxiliary
    graph needed for the algorithm is likely to be very dense. Thus, we use
    a complement graph data structure (see `AntiGraph`) to save memory.
    AntiGraph only stores information of the edges that are *not* present
    in the actual auxiliary graph. When applying algorithms to this
    complement graph data structure, it behaves as if it were the dense
    version.
    See also
    --------
```

```
(4)                 k_components
(4)                 References
(4)                 ----------
(4)                 .. [1]  Torrents, J. and F. Ferraro (2015) Structural Cohesion:
(12)                        Visualization and Heuristics for Fast Computation.
(12)                        https://arxiv.org/pdf/1503.04476v1
(4)                 .. [2]  White, Douglas R., and Mark Newman (2001) A Fast Algorithm for
(12)                        Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035
(12)                        https://www.santafe.edu/research/results/working-papers/fast-approximation-algorithms-for-finding-node-ind
(4)                 .. [3]  Moody, J. and D. White (2003). Social cohesion and embeddedness:
(12)                        A hierarchical conception of social groups.
(12)                        American Sociological Review 68(1), 103--28.
(12)                        https://doi.org/10.2307/3088904
(4)                 """
(4)                 k_components = defaultdict(list)
(4)                 node_connectivity = local_node_connectivity
(4)                 k_core = nx.k_core
(4)                 core_number = nx.core_number
(4)                 biconnected_components = nx.biconnected_components
(4)                 combinations = itertools.combinations
(4)                 for component in nx.connected_components(G):
(8)                     comp = set(component)
(8)                     if len(comp) > 1:
(12)                        k_components[1].append(comp)
(4)                 for bicomponent in nx.biconnected_components(G):
(8)                     bicomp = set(bicomponent)
(8)                     if len(bicomp) > 2:
(12)                        k_components[2].append(bicomp)
(4)                 g_cnumber = core_number(G)
(4)                 max_core = max(g_cnumber.values())
(4)                 for k in range(3, max_core + 1):
(8)                     C = k_core(G, k, core_number=g_cnumber)
(8)                     for nodes in biconnected_components(C):
(12)                        if len(nodes) < k:
(16)                            continue
(12)                        SG = G.subgraph(nodes)
(12)                        H = _AntiGraph()
(12)                        H.add_nodes_from(SG.nodes())
(12)                        for u, v in combinations(SG, 2):
(16)                            K = node_connectivity(SG, u, v, cutoff=k)
(16)                            if k > K:
(20)                                H.add_edge(u, v)
(12)                        for h_nodes in biconnected_components(H):
(16)                            if len(h_nodes) <= k:
(20)                                continue
(16)                            SH = H.subgraph(h_nodes)
(16)                            for Gc in _cliques_heuristic(SG, SH, k, min_density):
(20)                                for k_nodes in biconnected_components(Gc):
(24)                                    Gk = nx.k_core(SG.subgraph(k_nodes), k)
(24)                                    if len(Gk) <= k:
(28)                                        continue
(24)                                    k_components[k].append(set(Gk))
(4)                 return k_components
(0)             def _cliques_heuristic(G, H, k, min_density):
(4)                 h_cnumber = nx.core_number(H)
(4)                 for i, c_value in enumerate(sorted(set(h_cnumber.values()), reverse=True)):
(8)                     cands = {n for n, c in h_cnumber.items() if c == c_value}
(8)                     if i == 0:
(12)                        overlap = False
(8)                     else:
(12)                        overlap = set.intersection(
(16)                            *[{x for x in H[n] if x not in cands} for n in cands]
(12)                        )
(8)                     if overlap and len(overlap) < k:
(12)                        SH = H.subgraph(cands | overlap)
(8)                     else:
```

```
139: (12)                        SH = H.subgraph(cands)
140: (8)                    sh_cnumber = nx.core_number(SH)
141: (8)                    SG = nx.k_core(G.subgraph(SH), k)
142: (8)                    while not (_same(sh_cnumber) and nx.density(SH) >= min_density):
143: (12)                        SH = H.subgraph(SG).copy()
144: (12)                        if len(SH) <= k:
145: (16)                            break
146: (12)                        sh_cnumber = nx.core_number(SH)
147: (12)                        sh_deg = dict(SH.degree())
148: (12)                        min_deg = min(sh_deg.values())
149: (12)                        SH.remove_nodes_from(n for n, d in sh_deg.items() if d == min_deg)
150: (12)                        SG = nx.k_core(G.subgraph(SH), k)
151: (8)                    else:
152: (12)                        yield SG
153: (0)            def _same(measure, tol=0):
154: (4)                vals = set(measure.values())
155: (4)                if (max(vals) - min(vals)) <= tol:
156: (8)                    return True
157: (4)                return False
158: (0)            class _AntiGraph(nx.Graph):
159: (4)                """
160: (4)                Class for complement graphs.
161: (4)                The main goal is to be able to work with big and dense graphs with
162: (4)                a low memory footprint.
163: (4)                In this class you add the edges that *do not exist* in the dense graph,
164: (4)                the report methods of the class return the neighbors, the edges and
165: (4)                the degree as if it was the dense graph. Thus it's possible to use
166: (4)                an instance of this class with some of NetworkX functions. In this
167: (4)                case we only use k-core, connected_components, and biconnected_components.
168: (4)                """
169: (4)                all_edge_dict = {"weight": 1}
170: (4)                def single_edge_dict(self):
171: (8)                    return self.all_edge_dict
172: (4)                edge_attr_dict_factory = single_edge_dict  # type: ignore[assignment]
173: (4)                def __getitem__(self, n):
174: (8)                    """Returns a dict of neighbors of node n in the dense graph.
175: (8)                    Parameters
176: (8)                    ----------
177: (8)                    n : node
178: (11)                      A node in the graph.
179: (8)                    Returns
180: (8)                    -------
181: (8)                    adj_dict : dictionary
182: (11)                      The adjacency dictionary for nodes connected to n.
183: (8)                    """
184: (8)                    all_edge_dict = self.all_edge_dict
185: (8)                    return {
186: (12)                        node: all_edge_dict for node in set(self._adj) - set(self._adj[n])
- {n}
187: (8)                    }
188: (4)                def neighbors(self, n):
189: (8)                    """Returns an iterator over all neighbors of node n in the
190: (8)                    dense graph.
191: (8)                    """
192: (8)                    try:
193: (12)                        return iter(set(self._adj) - set(self._adj[n]) - {n})
194: (8)                    except KeyError as err:
195: (12)                        raise NetworkXError(f"The node {n} is not in the graph.") from err
196: (4)                class AntiAtlasView(Mapping):
197: (8)                    """An adjacency inner dict for AntiGraph"""
198: (8)                    def __init__(self, graph, node):
199: (12)                        self._graph = graph
200: (12)                        self._atlas = graph._adj[node]
201: (12)                        self._node = node
202: (8)                    def __len__(self):
203: (12)                        return len(self._graph) - len(self._atlas) - 1
204: (8)                    def __iter__(self):
205: (12)                        return (n for n in self._graph if n not in self._atlas and n !=
self._node)
```

```
206: (8)                    def __getitem__(self, nbr):
207: (12)                       nbrs = set(self._graph._adj) - set(self._atlas) - {self._node}
208: (12)                       if nbr in nbrs:
209: (16)                           return self._graph.all_edge_dict
210: (12)                       raise KeyError(nbr)
211: (4)                class AntiAdjacencyView(AntiAtlasView):
212: (8)                    """An adjacency outer dict for AntiGraph"""
213: (8)                    def __init__(self, graph):
214: (12)                       self._graph = graph
215: (12)                       self._atlas = graph._adj
216: (8)                    def __len__(self):
217: (12)                       return len(self._atlas)
218: (8)                    def __iter__(self):
219: (12)                       return iter(self._graph)
220: (8)                    def __getitem__(self, node):
221: (12)                       if node not in self._graph:
222: (16)                           raise KeyError(node)
223: (12)                       return self._graph.AntiAtlasView(self._graph, node)
224: (4)                @cached_property
225: (4)                def adj(self):
226: (8)                    return self.AntiAdjacencyView(self)
227: (4)                def subgraph(self, nodes):
228: (8)                    """This subgraph method returns a full AntiGraph. Not a View"""
229: (8)                    nodes = set(nodes)
230: (8)                    G = _AntiGraph()
231: (8)                    G.add_nodes_from(nodes)
232: (8)                    for n in G:
233: (12)                       Gnbrs = G.adjlist_inner_dict_factory()
234: (12)                       G._adj[n] = Gnbrs
235: (12)                       for nbr, d in self._adj[n].items():
236: (16)                           if nbr in G._adj:
237: (20)                               Gnbrs[nbr] = d
238: (20)                               G._adj[nbr][n] = d
239: (8)                    G.graph = self.graph
240: (8)                    return G
241: (4)                class AntiDegreeView(nx.reportviews.DegreeView):
242: (8)                    def __iter__(self):
243: (12)                       all_nodes = set(self._succ)
244: (12)                       for n in self._nodes:
245: (16)                           nbrs = all_nodes - set(self._succ[n]) - {n}
246: (16)                           yield (n, len(nbrs))
247: (8)                    def __getitem__(self, n):
248: (12)                       nbrs = set(self._succ) - set(self._succ[n]) - {n}
249: (12)                       return len(nbrs) + (n in nbrs)
250: (4)                @cached_property
251: (4)                def degree(self):
252: (8)                    """Returns an iterator for (node, degree) and degree for single node.
253: (8)                    The node degree is the number of edges adjacent to the node.
254: (8)                    Parameters
255: (8)                    ----------
256: (8)                    nbunch : iterable container, optional (default=all nodes)
257: (12)                       A container of nodes.  The container will be iterated
258: (12)                       through once.
259: (8)                    weight : string or None, optional (default=None)
260: (11)                      The edge attribute that holds the numerical value used
261: (11)                      as a weight.  If None, then each edge has weight 1.
262: (11)                      The degree is the sum of the edge weights adjacent to the node.
263: (8)                    Returns
264: (8)                    -------
265: (8)                    deg:
266: (12)                       Degree of the node, if a single node is passed as argument.
267: (8)                    nd_iter : an iterator
268: (12)                       The iterator returns two-tuples of (node, degree).
269: (8)                    See Also
270: (8)                    --------
271: (8)                    degree
272: (8)                    Examples
273: (8)                    --------
274: (8)                    >>> G = nx.path_graph(4)
```

```
275: (8)                    >>> G.degree(0)  # node 0 with degree 1
276: (8)                    1
277: (8)                    >>> list(G.degree([0, 1]))
278: (8)                    [(0, 1), (1, 2)]
279: (8)                    """
280: (8)                    return self.AntiDegreeView(self)
281: (4)                def adjacency(self):
282: (8)                    """Returns an iterator of (node, adjacency set) tuples for all nodes
283: (11)                      in the dense graph.
284: (8)                    This is the fastest way to look at every edge.
285: (8)                    For directed graphs, only outgoing adjacencies are included.
286: (8)                    Returns
287: (8)                    -------
288: (8)                    adj_iter : iterator
289: (11)                      An iterator of (node, adjacency set) for all nodes in
290: (11)                      the graph.
291: (8)                    """
292: (8)                    for n in self._adj:
293: (12)                       yield (n, set(self._adj) - set(self._adj[n]) - {n})
```

File 71 - matching.py:

```
1: (0)              """
2: (0)              **************
3: (0)              Graph Matching
4: (0)              **************
5: (0)              Given a graph G = (V,E), a matching M in G is a set of pairwise non-adjacent
6: (0)              edges; that is, no two edges share a common vertex.
7: (0)              `Wikipedia: Matching <https://en.wikipedia.org/wiki/Matching_(graph_theory)>`_
8: (0)              """
9: (0)              import networkx as nx
10: (0)             __all__ = ["min_maximal_matching"]
11: (0)             @nx._dispatchable
12: (0)             def min_maximal_matching(G):
13: (4)                 r"""Returns the minimum maximal matching of G. That is, out of all maximal
14: (4)                 matchings of the graph G, the smallest is returned.
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 G : NetworkX graph
18: (6)                   Undirected graph
19: (4)                 Returns
20: (4)                 -------
21: (4)                 min_maximal_matching : set
22: (6)                   Returns a set of edges such that no two edges share a common endpoint
23: (6)                   and every edge not in the set shares some common endpoint in the set.
24: (6)                   Cardinality will be 2*OPT in the worst case.
25: (4)                 Notes
26: (4)                 -----
27: (4)                 The algorithm computes an approximate solution for the minimum maximal
28: (4)                 cardinality matching problem. The solution is no more than 2 * OPT in
size.
29: (4)                 Runtime is $O(|E|)$.
30: (4)                 References
31: (4)                 ----------
32: (4)                 .. [1] Vazirani, Vijay Approximation Algorithms (2001)
33: (4)                 """
34: (4)                 return nx.maximal_matching(G)
```

File 72 - maxcut.py:

```
1: (0)              import networkx as nx
2: (0)              from networkx.utils.decorators import not_implemented_for, py_random_state
3: (0)              __all__ = ["randomized_partitioning", "one_exchange"]
4: (0)              @not_implemented_for("directed")
5: (0)              @not_implemented_for("multigraph")
```

```
@py_random_state(1)
@nx._dispatchable(edge_attrs="weight")
def randomized_partitioning(G, seed=None, p=0.5, weight=None):
    """Compute a random partitioning of the graph nodes and its cut value.
    A partitioning is calculated by observing each node
    and deciding to add it to the partition with probability `p`,
    returning a random cut and its corresponding value (the
    sum of weights of edges connecting different partitions).
    Parameters
    ----------
    G : NetworkX graph
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    p : scalar
        Probability for each node to be part of the first partition.
        Should be in [0,1]
    weight : object
        Edge attribute key to use as weight. If not specified, edges
        have weight one.
    Returns
    -------
    cut_size : scalar
        Value of the minimum cut.
    partition : pair of node sets
        A partitioning of the nodes that defines a minimum cut.
    Examples
    --------
    >>> G = nx.complete_graph(5)
    >>> cut_size, partition = nx.approximation.randomized_partitioning(G, seed=1)

    >>> cut_size
    6
    >>> partition
    ({0, 3, 4}, {1, 2})
    Raises
    ------
    NetworkXNotImplemented
        If the graph is directed or is a multigraph.
    """
    cut = {node for node in G.nodes() if seed.random() < p}
    cut_size = nx.algorithms.cut_size(G, cut, weight=weight)
    partition = (cut, G.nodes - cut)
    return cut_size, partition
def _swap_node_partition(cut, node):
    return cut - {node} if node in cut else cut.union({node})
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@py_random_state(2)
@nx._dispatchable(edge_attrs="weight")
def one_exchange(G, initial_cut=None, seed=None, weight=None):
    """Compute a partitioning of the graphs nodes and the corresponding cut value.

    Use a greedy one exchange strategy to find a locally maximal cut
    and its value, it works by finding the best node (one that gives
    the highest gain to the cut value) to add to the current cut
    and repeats this process until no improvement can be made.
    Parameters
    ----------
    G : networkx Graph
        Graph to find a maximum cut for.
    initial_cut : set
        Cut to use as a starting point. If not supplied the algorithm
        starts with an empty cut.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    weight : object
        Edge attribute key to use as weight. If not specified, edges
```

```
73: (8)                     have weight one.
74: (4)                 Returns
75: (4)                 -------
76: (4)                 cut_value : scalar
77: (8)                     Value of the maximum cut.
78: (4)                 partition : pair of node sets
79: (8)                     A partitioning of the nodes that defines a maximum cut.
80: (4)                 Examples
81: (4)                 --------
82: (4)                 >>> G = nx.complete_graph(5)
83: (4)                 >>> curr_cut_size, partition = nx.approximation.one_exchange(G, seed=1)
84: (4)                 >>> curr_cut_size
85: (4)                 6
86: (4)                 >>> partition
87: (4)                 ({0, 2}, {1, 3, 4})
88: (4)                 Raises
89: (4)                 ------
90: (4)                 NetworkXNotImplemented
91: (8)                     If the graph is directed or is a multigraph.
92: (4)                 """
93: (4)                 if initial_cut is None:
94: (8)                     initial_cut = set()
95: (4)                 cut = set(initial_cut)
96: (4)                 current_cut_size = nx.algorithms.cut_size(G, cut, weight=weight)
97: (4)                 while True:
98: (8)                     nodes = list(G.nodes())
99: (8)                     seed.shuffle(nodes)
100: (8)                    best_node_to_swap = max(
101: (12)                       nodes,
102: (12)                       key=lambda v: nx.algorithms.cut_size(
103: (16)                           G, _swap_node_partition(cut, v), weight=weight
104: (12)                       ),
105: (12)                       default=None,
106: (8)                    )
107: (8)                    potential_cut = _swap_node_partition(cut, best_node_to_swap)
108: (8)                    potential_cut_size = nx.algorithms.cut_size(G, potential_cut,
weight=weight)
109: (8)                    if potential_cut_size > current_cut_size:
110: (12)                       cut = potential_cut
111: (12)                       current_cut_size = potential_cut_size
112: (8)                    else:
113: (12)                       break
114: (4)                partition = (cut, G.nodes - cut)
115: (4)                return current_cut_size, partition
```

File 73 - ramsey.py:

```
1: (0)              """
2: (0)              Ramsey numbers.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              from ...utils import arbitrary_element
7: (0)              __all__ = ["ramsey_R2"]
8: (0)              @not_implemented_for("directed")
9: (0)              @not_implemented_for("multigraph")
10: (0)             @nx._dispatchable
11: (0)             def ramsey_R2(G):
12: (4)                 r"""Compute the largest clique and largest independent set in `G`.
13: (4)                 This can be used to estimate bounds for the 2-color
14: (4)                 Ramsey number `R(2;s,t)` for `G`.
15: (4)                 This is a recursive implementation which could run into trouble
16: (4)                 for large recursions. Note that self-loop edges are ignored.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : NetworkX graph
20: (8)                     Undirected graph
```

```
21: (4)                 Returns
22: (4)                 -------
23: (4)                 max_pair : (set, set) tuple
24: (8)                     Maximum clique, Maximum independent set.
25: (4)                 Raises
26: (4)                 ------
27: (4)                 NetworkXNotImplemented
28: (8)                     If the graph is directed or is a multigraph.
29: (4)                 """
30: (4)                 if not G:
31: (8)                     return set(), set()
32: (4)                 node = arbitrary_element(G)
33: (4)                 nbrs = (nbr for nbr in nx.all_neighbors(G, node) if nbr != node)
34: (4)                 nnbrs = nx.non_neighbors(G, node)
35: (4)                 c_1, i_1 = ramsey_R2(G.subgraph(nbrs).copy())
36: (4)                 c_2, i_2 = ramsey_R2(G.subgraph(nnbrs).copy())
37: (4)                 c_1.add(node)
38: (4)                 i_2.add(node)
39: (4)                 return max(c_1, c_2, key=len), max(i_1, i_2, key=len)

----------------------------------------

File 74 - steinertree.py:

1: (0)              from itertools import chain
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for, pairwise
4: (0)              __all__ = ["metric_closure", "steiner_tree"]
5: (0)              @not_implemented_for("directed")
6: (0)              @nx._dispatchable(edge_attrs="weight", returns_graph=True)
7: (0)              def metric_closure(G, weight="weight"):
8: (4)                  """Return the metric closure of a graph.
9: (4)                  The metric closure of a graph *G* is the complete graph in which each edge
10: (4)                 is weighted by the shortest path distance between the nodes in *G* .
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 G : NetworkX graph
14: (4)                 Returns
15: (4)                 -------
16: (4)                 NetworkX graph
17: (8)                     Metric closure of the graph `G`.
18: (4)                 """
19: (4)                 M = nx.Graph()
20: (4)                 Gnodes = set(G)
21: (4)                 all_paths_iter = nx.all_pairs_dijkstra(G, weight=weight)
22: (4)                 u, (distance, path) = next(all_paths_iter)
23: (4)                 if Gnodes - set(distance):
24: (8)                     msg = "G is not a connected graph. metric_closure is not defined."
25: (8)                     raise nx.NetworkXError(msg)
26: (4)                 Gnodes.remove(u)
27: (4)                 for v in Gnodes:
28: (8)                     M.add_edge(u, v, distance=distance[v], path=path[v])
29: (4)                 for u, (distance, path) in all_paths_iter:
30: (8)                     Gnodes.remove(u)
31: (8)                     for v in Gnodes:
32: (12)                        M.add_edge(u, v, distance=distance[v], path=path[v])
33: (4)                 return M
34: (0)             def _mehlhorn_steiner_tree(G, terminal_nodes, weight):
35: (4)                 paths = nx.multi_source_dijkstra_path(G, terminal_nodes)
36: (4)                 d_1 = {}
37: (4)                 s = {}
38: (4)                 for v in G.nodes():
39: (8)                     s[v] = paths[v][0]
40: (8)                     d_1[(v, s[v])] = len(paths[v]) - 1
41: (4)                 G_1_prime = nx.Graph()
42: (4)                 for u, v, data in G.edges(data=True):
43: (8)                     su, sv = s[u], s[v]
44: (8)                     weight_here = d_1[(u, su)] + data.get(weight, 1) + d_1[(v, sv)]
45: (8)                     if not G_1_prime.has_edge(su, sv):
```

```
46: (12)                        G_1_prime.add_edge(su, sv, weight=weight_here)
47: (8)                     else:
48: (12)                        new_weight = min(weight_here, G_1_prime[su][sv]["weight"])
49: (12)                        G_1_prime.add_edge(su, sv, weight=new_weight)
50: (4)                 G_2 = nx.minimum_spanning_edges(G_1_prime, data=True)
51: (4)                 G_3 = nx.Graph()
52: (4)                 for u, v, d in G_2:
53: (8)                     path = nx.shortest_path(G, u, v, weight)
54: (8)                     for n1, n2 in pairwise(path):
55: (12)                        G_3.add_edge(n1, n2)
56: (4)                 G_3_mst = list(nx.minimum_spanning_edges(G_3, data=False))
57: (4)                 if G.is_multigraph():
58: (8)                     G_3_mst = (
59: (12)                        (u, v, min(G[u][v], key=lambda k: G[u][v][k][weight])) for u, v in
G_3_mst
60: (8)                     )
61: (4)                 G_4 = G.edge_subgraph(G_3_mst).copy()
62: (4)                 _remove_nonterminal_leaves(G_4, terminal_nodes)
63: (4)                 return G_4.edges()
64: (0)             def _kou_steiner_tree(G, terminal_nodes, weight):
65: (4)                 M = metric_closure(G, weight=weight)
66: (4)                 H = M.subgraph(terminal_nodes)
67: (4)                 mst_edges = nx.minimum_spanning_edges(H, weight="distance", data=True)
68: (4)                 mst_all_edges = chain.from_iterable(pairwise(d["path"]) for u, v, d in
mst_edges)
69: (4)                 if G.is_multigraph():
70: (8)                     mst_all_edges = (
71: (12)                        (u, v, min(G[u][v], key=lambda k: G[u][v][k][weight]))
72: (12)                        for u, v in mst_all_edges
73: (8)                     )
74: (4)                 G_S = G.edge_subgraph(mst_all_edges)
75: (4)                 T_S = nx.minimum_spanning_edges(G_S, weight="weight", data=False)
76: (4)                 T_H = G.edge_subgraph(T_S).copy()
77: (4)                 _remove_nonterminal_leaves(T_H, terminal_nodes)
78: (4)                 return T_H.edges()
79: (0)             def _remove_nonterminal_leaves(G, terminals):
80: (4)                 terminals_set = set(terminals)
81: (4)                 for n in list(G.nodes):
82: (8)                     if n not in terminals_set and G.degree(n) == 1:
83: (12)                        G.remove_node(n)
84: (0)             ALGORITHMS = {
85: (4)                 "kou": _kou_steiner_tree,
86: (4)                 "mehlhorn": _mehlhorn_steiner_tree,
87: (0)             }
88: (0)             @not_implemented_for("directed")
89: (0)             @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
90: (0)             def steiner_tree(G, terminal_nodes, weight="weight", method=None):
91: (4)                 r"""Return an approximation to the minimum Steiner tree of a graph.
92: (4)                 The minimum Steiner tree of `G` w.r.t a set of `terminal_nodes` (also *S*)
93: (4)                 is a tree within `G` that spans those nodes and has minimum size (sum of
94: (4)                 edge weights) among all such trees.
95: (4)                 The approximation algorithm is specified with the `method` keyword
96: (4)                 argument. All three available algorithms produce a tree whose weight is
97: (4)                 within a ``(2 - (2 / l))`` factor of the weight of the optimal Steiner
tree,
98: (4)                 where ``l`` is the minimum number of leaf nodes across all possible
Steiner
99: (4)                 trees.
100: (4)                * ``"kou"`` [2]_ (runtime $O(|S| |V|^2)$) computes the minimum spanning
tree of
101: (6)                  the subgraph of the metric closure of *G* induced by the terminal nodes,
102: (6)                  where the metric closure of *G* is the complete graph in which each edge
is
103: (6)                  weighted by the shortest path distance between the nodes in *G*.
104: (4)                * ``"mehlhorn"`` [3]_ (runtime $O(|E|+|V|\log|V|)$) modifies Kou et al.'s
105: (6)                  algorithm, beginning by finding the closest terminal node for each
106: (6)                  non-terminal. This data is used to create a complete graph containing
only
107: (6)                  the terminal nodes, in which edge is weighted with the shortest path
```

```
108: (6)                  distance between them. The algorithm then proceeds in the same way as
Kou
109: (6)                  et al..
110: (4)                Parameters
111: (4)                ----------
112: (4)                G : NetworkX graph
113: (4)                terminal_nodes : list
114: (9)                     A list of terminal nodes for which minimum steiner tree is
115: (9)                     to be found.
116: (4)                weight : string (default = 'weight')
117: (8)                    Use the edge attribute specified by this string as the edge weight.
118: (8)                    Any edge attribute not present defaults to 1.
119: (4)                method : string, optional (default = 'mehlhorn')
120: (8)                    The algorithm to use to approximate the Steiner tree.
121: (8)                    Supported options: 'kou', 'mehlhorn'.
122: (8)                    Other inputs produce a ValueError.
123: (4)                Returns
124: (4)                -------
125: (4)                NetworkX graph
126: (8)                    Approximation to the minimum steiner tree of `G` induced by
127: (8)                    `terminal_nodes` .
128: (4)                Raises
129: (4)                ------
130: (4)                NetworkXNotImplemented
131: (8)                    If `G` is directed.
132: (4)                ValueError
133: (8)                    If the specified `method` is not supported.
134: (4)                Notes
135: (4)                -----
136: (4)                For multigraphs, the edge between two nodes with minimum weight is the
137: (4)                edge put into the Steiner tree.
138: (4)                References
139: (4)                ----------
140: (4)                .. [1] Steiner_tree_problem on Wikipedia.
141: (11)                      https://en.wikipedia.org/wiki/Steiner_tree_problem
142: (4)                .. [2] Kou, L., G. Markowsky, and L. Berman. 1981.
143: (11)                      ‘A Fast Algorithm for Steiner Trees’.
144: (11)                      Acta Informatica 15 (2): 141–45.
145: (11)                      https://doi.org/10.1007/BF00288961.
146: (4)                .. [3] Mehlhorn, Kurt. 1988.
147: (11)                      ‘A Faster Approximation Algorithm for the Steiner Problem in
Graphs’.
148: (11)                      Information Processing Letters 27 (3): 125–28.
149: (11)                      https://doi.org/10.1016/0020-0190(88)90066-X.
150: (4)                """
151: (4)                if method is None:
152: (8)                    method = "mehlhorn"
153: (4)                try:
154: (8)                    algo = ALGORITHMS[method]
155: (4)                except KeyError as e:
156: (8)                    raise ValueError(f"{method} is not a valid choice for an algorithm.")
from e
157: (4)                edges = algo(G, terminal_nodes, weight)
158: (4)                if G.is_multigraph():
159: (8)                    edges = (
160: (12)                       (u, v, min(G[u][v], key=lambda k: G[u][v][k][weight])) for u, v in
edges
161: (8)                    )
162: (4)                T = G.edge_subgraph(edges)
163: (4)                return T

----------------------------------------

File 75 - traveling_salesman.py:

1: (0)              """
2: (0)              ================================
3: (0)              Travelling Salesman Problem (TSP)
4: (0)              ================================
```

```
5: (0)              Implementation of approximate algorithms
6: (0)              for solving and approximating the TSP problem.
7: (0)              Categories of algorithms which are implemented:
8: (0)              - Christofides (provides a 3/2-approximation of TSP)
9: (0)              - Greedy
10: (0)             - Simulated Annealing (SA)
11: (0)             - Threshold Accepting (TA)
12: (0)             - Asadpour Asymmetric Traveling Salesman Algorithm
13: (0)             The Travelling Salesman Problem tries to find, given the weight
14: (0)             (distance) between all points where a salesman has to visit, the
15: (0)             route so that:
16: (0)             - The total distance (cost) which the salesman travels is minimized.
17: (0)             - The salesman returns to the starting point.
18: (0)             - Note that for a complete graph, the salesman visits each point once.
19: (0)             The function `travelling_salesman_problem` allows for incomplete
20: (0)             graphs by finding all-pairs shortest paths, effectively converting
21: (0)             the problem to a complete graph problem. It calls one of the
22: (0)             approximate methods on that problem and then converts the result
23: (0)             back to the original graph using the previously found shortest paths.
24: (0)             TSP is an NP-hard problem in combinatorial optimization,
25: (0)             important in operations research and theoretical computer science.
26: (0)             http://en.wikipedia.org/wiki/Travelling_salesman_problem
27: (0)             """
28: (0)             import math
29: (0)             import networkx as nx
30: (0)             from networkx.algorithms.tree.mst import random_spanning_tree
31: (0)             from networkx.utils import not_implemented_for, pairwise, py_random_state
32: (0)             __all__ = [
33: (4)                 "traveling_salesman_problem",
34: (4)                 "christofides",
35: (4)                 "asadpour_atsp",
36: (4)                 "greedy_tsp",
37: (4)                 "simulated_annealing_tsp",
38: (4)                 "threshold_accepting_tsp",
39: (0)             ]
40: (0)             def swap_two_nodes(soln, seed):
41: (4)                 """Swap two nodes in `soln` to give a neighbor solution.
42: (4)                 Parameters
43: (4)                 ----------
44: (4)                 soln : list of nodes
45: (8)                     Current cycle of nodes
46: (4)                 seed : integer, random_state, or None (default)
47: (8)                     Indicator of random number generation state.
48: (8)                     See :ref:`Randomness<randomness>`.
49: (4)                 Returns
50: (4)                 -------
51: (4)                 list
52: (8)                     The solution after move is applied. (A neighbor solution.)
53: (4)                 Notes
54: (4)                 -----
55: (8)                     This function assumes that the incoming list `soln` is a cycle
56: (8)                     (that the first and last element are the same) and also that
57: (8)                     we don't want any move to change the first node in the list
58: (8)                     (and thus not the last node either).
59: (8)                     The input list is changed as well as returned. Make a copy if needed.
60: (4)                 See Also
61: (4)                 --------
62: (8)                     move_one_node
63: (4)                 """
64: (4)                 a, b = seed.sample(range(1, len(soln) - 1), k=2)
65: (4)                 soln[a], soln[b] = soln[b], soln[a]
66: (4)                 return soln
67: (0)             def move_one_node(soln, seed):
68: (4)                 """Move one node to another position to give a neighbor solution.
69: (4)                 The node to move and the position to move to are chosen randomly.
70: (4)                 The first and last nodes are left untouched as soln must be a cycle
71: (4)                 starting at that node.
72: (4)                 Parameters
73: (4)                 ----------
```

```
(4)                 soln : list of nodes
(8)                     Current cycle of nodes
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 list
(8)                     The solution after move is applied. (A neighbor solution.)
(4)                 Notes
(4)                 -----
(8)                     This function assumes that the incoming list `soln` is a cycle
(8)                     (that the first and last element are the same) and also that
(8)                     we don't want any move to change the first node in the list
(8)                     (and thus not the last node either).
(8)                     The input list is changed as well as returned. Make a copy if needed.
(4)                 See Also
(4)                 --------
(8)                     swap_two_nodes
(4)                 """
(4)                 a, b = seed.sample(range(1, len(soln) - 1), k=2)
(4)                 soln.insert(b, soln.pop(a))
(4)                 return soln
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def christofides(G, weight="weight", tree=None):
(4)                 """Approximate a solution of the traveling salesman problem
(4)                 Compute a 3/2-approximation of the traveling salesman problem
(4)                 in a complete undirected graph using Christofides [1]_ algorithm.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph
(8)                     `G` should be a complete weighted undirected graph.
(8)                     The distance between all pairs of nodes should be included.
(4)                 weight : string, optional (default="weight")
(8)                     Edge data key corresponding to the edge weight.
(8)                     If any edge does not have this attribute the weight is set to 1.
(4)                 tree : NetworkX graph or None (default: None)
(8)                     A minimum spanning tree of G. Or, if None, the minimum spanning
(8)                     tree is computed using :func:`networkx.minimum_spanning_tree`
(4)                 Returns
(4)                 -------
(4)                 list
(8)                     List of nodes in `G` along a cycle with a 3/2-approximation of
(8)                     the minimal Hamiltonian cycle.
(4)                 References
(4)                 ----------
(4)                 .. [1] Christofides, Nicos. "Worst-case analysis of a new heuristic for
(7)                    the travelling salesman problem." No. RR-388. Carnegie-Mellon Univ
(7)                    Pittsburgh Pa Management Sciences Research Group, 1976.
(4)                 """
(4)                 loop_nodes = nx.nodes_with_selfloops(G)
(4)                 try:
(8)                     node = next(loop_nodes)
(4)                 except StopIteration:
(8)                     pass
(4)                 else:
(8)                     G = G.copy()
(8)                     G.remove_edge(node, node)
(8)                     G.remove_edges_from((n, n) for n in loop_nodes)
(4)                 N = len(G) - 1
(4)                 if any(len(nbrdict) != N for n, nbrdict in G.adj.items()):
(8)                     raise nx.NetworkXError("G must be a complete graph.")
(4)                 if tree is None:
(8)                     tree = nx.minimum_spanning_tree(G, weight=weight)
(4)                 L = G.copy()
(4)                 L.remove_nodes_from([v for v, degree in tree.degree if not (degree % 2)])
(4)                 MG = nx.MultiGraph()
(4)                 MG.add_edges_from(tree.edges)
```

```
143: (4)                edges = nx.min_weight_matching(L, weight=weight)
144: (4)                MG.add_edges_from(edges)
145: (4)                return _shortcutting(nx.eulerian_circuit(MG))
146: (0)            def _shortcutting(circuit):
147: (4)                """Remove duplicate nodes in the path"""
148: (4)                nodes = []
149: (4)                for u, v in circuit:
150: (8)                    if v in nodes:
151: (12)                       continue
152: (8)                    if not nodes:
153: (12)                       nodes.append(u)
154: (8)                    nodes.append(v)
155: (4)                nodes.append(nodes[0])
156: (4)                return nodes
157: (0)            @nx._dispatchable(edge_attrs="weight")
158: (0)            def traveling_salesman_problem(
159: (4)                G, weight="weight", nodes=None, cycle=True, method=None, **kwargs
160: (0)            ):
161: (4)                """Find the shortest path in `G` connecting specified nodes
162: (4)                This function allows approximate solution to the traveling salesman
163: (4)                problem on networks that are not complete graphs and/or where the
164: (4)                salesman does not need to visit all nodes.
165: (4)                This function proceeds in two steps. First, it creates a complete
166: (4)                graph using the all-pairs shortest_paths between nodes in `nodes`.
167: (4)                Edge weights in the new graph are the lengths of the paths
168: (4)                between each pair of nodes in the original graph.
169: (4)                Second, an algorithm (default: `christofides` for undirected and
170: (4)                `asadpour_atsp` for directed) is used to approximate the minimal
Hamiltonian
171: (4)                cycle on this new graph. The available algorithms are:
172: (5)                 - christofides
173: (5)                 - greedy_tsp
174: (5)                 - simulated_annealing_tsp
175: (5)                 - threshold_accepting_tsp
176: (5)                 - asadpour_atsp
177: (4)                Once the Hamiltonian Cycle is found, this function post-processes to
178: (4)                accommodate the structure of the original graph. If `cycle` is ``False``,
179: (4)                the biggest weight edge is removed to make a Hamiltonian path.
180: (4)                Then each edge on the new complete graph used for that analysis is
181: (4)                replaced by the shortest_path between those nodes on the original graph.
182: (4)                If the input graph `G` includes edges with weights that do not adhere to
183: (4)                the triangle inequality, such as when `G` is not a complete graph (i.e
184: (4)                length of non-existent edges is infinity), then the returned path may
185: (4)                contain some repeating nodes (other than the starting node).
186: (4)                Parameters
187: (4)                ----------
188: (4)                G : NetworkX graph
189: (8)                    A possibly weighted graph
190: (4)                nodes : collection of nodes (default=G.nodes)
191: (8)                    collection (list, set, etc.) of nodes to visit
192: (4)                weight : string, optional (default="weight")
193: (8)                    Edge data key corresponding to the edge weight.
194: (8)                    If any edge does not have this attribute the weight is set to 1.
195: (4)                cycle : bool (default: True)
196: (8)                    Indicates whether a cycle should be returned, or a path.
197: (8)                    Note: the cycle is the approximate minimal cycle.
198: (8)                    The path simply removes the biggest edge in that cycle.
199: (4)                method : function (default: None)
200: (8)                    A function that returns a cycle on all nodes and approximates
201: (8)                    the solution to the traveling salesman problem on a complete
202: (8)                    graph. The returned cycle is then used to find a corresponding
203: (8)                    solution on `G`. `method` should be callable; take inputs
204: (8)                    `G`, and `weight`; and return a list of nodes along the cycle.
205: (8)                    Provided options include :func:`christofides`, :func:`greedy_tsp`,
206: (8)                    :func:`simulated_annealing_tsp` and :func:`threshold_accepting_tsp`.
207: (8)                    If `method is None`: use :func:`christofides` for undirected `G` and
208: (8)                    :func:`asadpour_atsp` for directed `G`.
209: (4)                **kwargs : dict
210: (8)                    Other keyword arguments to be passed to the `method` function passed
```

```
in.
    Returns
    -------
    list
        List of nodes in `G` along a path with an approximation of the minimal
        path through `nodes`.
    Raises
    ------
    NetworkXError
        If `G` is a directed graph it has to be strongly connected or the
        complete version cannot be generated.
    Examples
    --------
    >>> tsp = nx.approximation.traveling_salesman_problem
    >>> G = nx.cycle_graph(9)
    >>> G[4][5]["weight"] = 5  # all other weights are 1
    >>> tsp(G, nodes=[3, 6])
    [3, 2, 1, 0, 8, 7, 6, 7, 8, 0, 1, 2, 3]
    >>> path = tsp(G, cycle=False)
    >>> path in ([4, 3, 2, 1, 0, 8, 7, 6, 5], [5, 6, 7, 8, 0, 1, 2, 3, 4])
    True
    While no longer required, you can still build (curry) your own function
    to provide parameter values to the methods.
    >>> SA_tsp = nx.approximation.simulated_annealing_tsp
    >>> method = lambda G, weight: SA_tsp(G, "greedy", weight=weight,
temp=500)
    >>> path = tsp(G, cycle=False, method=method)
    >>> path in ([4, 3, 2, 1, 0, 8, 7, 6, 5], [5, 6, 7, 8, 0, 1, 2, 3, 4])
    True
    Otherwise, pass other keyword arguments directly into the tsp function.
    >>> path = tsp(
    ...     G,
    ...     cycle=False,
    ...     method=nx.approximation.simulated_annealing_tsp,
    ...     init_cycle="greedy",
    ...     temp=500,
    ... )
    >>> path in ([4, 3, 2, 1, 0, 8, 7, 6, 5], [5, 6, 7, 8, 0, 1, 2, 3, 4])
    True
    """
    if method is None:
        if G.is_directed():
            method = asadpour_atsp
        else:
            method = christofides
    if nodes is None:
        nodes = list(G.nodes)
    dist = {}
    path = {}
    for n, (d, p) in nx.all_pairs_dijkstra(G, weight=weight):
        dist[n] = d
        path[n] = p
    if G.is_directed():
        if not nx.is_strongly_connected(G):
            raise nx.NetworkXError("G is not strongly connected")
        GG = nx.DiGraph()
    else:
        GG = nx.Graph()
    for u in nodes:
        for v in nodes:
            if u == v:
                continue
            GG.add_edge(u, v, weight=dist[u][v])
    best_GG = method(GG, weight=weight, **kwargs)
    if not cycle:
        (u, v) = max(pairwise(best_GG), key=lambda x: dist[x[0]][x[1]])
        pos = best_GG.index(u) + 1
        while best_GG[pos] != v:
            pos = best_GG[pos:].index(u) + 1
```

```
278: (8)                    best_GG = best_GG[pos:-1] + best_GG[:pos]
279: (4)                best_path = []
280: (4)                for u, v in pairwise(best_GG):
281: (8)                    best_path.extend(path[u][v][:-1])
282: (4)                best_path.append(v)
283: (4)                return best_path
284: (0)            @not_implemented_for("undirected")
285: (0)            @py_random_state(2)
286: (0)            @nx._dispatchable(edge_attrs="weight", mutates_input=True)
287: (0)            def asadpour_atsp(G, weight="weight", seed=None, source=None):
288: (4)                """
289: (4)                Returns an approximate solution to the traveling salesman problem.
290: (4)                This approximate solution is one of the best known approximations for the
291: (4)                asymmetric traveling salesman problem developed by Asadpour et al,
292: (4)                [1]_. The algorithm first solves the Held-Karp relaxation to find a lower
293: (4)                bound for the weight of the cycle. Next, it constructs an exponential
294: (4)                distribution of undirected spanning trees where the probability of an
295: (4)                edge being in the tree corresponds to the weight of that edge using a
296: (4)                maximum entropy rounding scheme. Next we sample that distribution
297: (4)                $2 \\lceil \\ln n \\rceil$ times and save the minimum sampled tree once
the
298: (4)                direction of the arcs is added back to the edges. Finally, we augment
299: (4)                then short circuit that graph to find the approximate tour for the
300: (4)                salesman.
301: (4)                Parameters
302: (4)                ----------
303: (4)                G : nx.DiGraph
304: (8)                    The graph should be a complete weighted directed graph. The
305: (8)                    distance between all paris of nodes should be included and the
triangle
306: (8)                    inequality should hold. That is, the direct edge between any two nodes
307: (8)                    should be the path of least cost.
308: (4)                weight : string, optional (default="weight")
309: (8)                    Edge data key corresponding to the edge weight.
310: (8)                    If any edge does not have this attribute the weight is set to 1.
311: (4)                seed : integer, random_state, or None (default)
312: (8)                    Indicator of random number generation state.
313: (8)                    See :ref:`Randomness<randomness>`.
314: (4)                source : node label (default=`None`)
315: (8)                    If given, return the cycle starting and ending at the given node.
316: (4)                Returns
317: (4)                -------
318: (4)                cycle : list of nodes
319: (8)                    Returns the cycle (list of nodes) that a salesman can follow to
minimize
320: (8)                    the total weight of the trip.
321: (4)                Raises
322: (4)                ------
323: (4)                NetworkXError
324: (8)                    If `G` is not complete or has less than two nodes, the algorithm
raises
325: (8)                    an exception.
326: (4)                NetworkXError
327: (8)                    If `source` is not `None` and is not a node in `G`, the algorithm
raises
328: (8)                    an exception.
329: (4)                NetworkXNotImplemented
330: (8)                    If `G` is an undirected graph.
331: (4)                References
332: (4)                ----------
333: (4)                .. [1] A. Asadpour, M. X. Goemans, A. Madry, S. O. Gharan, and A. Saberi,
334: (7)                   An o(log n/log log n)-approximation algorithm for the asymmetric
335: (7)                   traveling salesman problem, Operations research, 65 (2017),
336: (7)                   pp. 1043–1061
337: (4)                Examples
338: (4)                --------
339: (4)                >>> import networkx as nx
340: (4)                >>> import networkx.algorithms.approximation as approx
341: (4)                >>> G = nx.complete_graph(3, create_using=nx.DiGraph)
```

```
(4)                >>> nx.set_edge_attributes(
(4)                ...     G, {(0, 1): 2, (1, 2): 2, (2, 0): 2, (0, 2): 1, (2, 1): 1, (1, 0): 1}, "weight"
(4)                ... )
(4)                >>> tour = approx.asadpour_atsp(G, source=0)
(4)                >>> tour
(4)                [0, 2, 1, 0]
(4)                """
(4)                from math import ceil, exp
(4)                from math import log as ln
(4)                N = len(G) - 1
(4)                if N < 2:
(8)                    raise nx.NetworkXError("G must have at least two nodes")
(4)                if any(len(nbrdict) - (n in nbrdict) != N for n, nbrdict in G.adj.items()):
(8)                    raise nx.NetworkXError("G is not a complete DiGraph")
(4)                if source is not None and source not in G.nodes:
(8)                    raise nx.NetworkXError("Given source node not in G.")
(4)                opt_hk, z_star = held_karp_ascent(G, weight)
(4)                if not isinstance(z_star, dict):
(8)                    return _shortcutting(nx.eulerian_circuit(z_star, source=source))
(4)                z_support = nx.MultiGraph()
(4)                for u, v in z_star:
(8)                    if (u, v) not in z_support.edges:
(12)                       edge_weight = min(G[u][v][weight], G[v][u][weight])
(12)                       z_support.add_edge(u, v, **{weight: edge_weight})
(4)                gamma = spanning_tree_distribution(z_support, z_star)
(4)                z_support = nx.Graph(z_support)
(4)                lambda_dict = {(u, v): exp(gamma[(u, v)]) for u, v in z_support.edges()}
(4)                nx.set_edge_attributes(z_support, lambda_dict, "weight")
(4)                del gamma, lambda_dict
(4)                minimum_sampled_tree = None
(4)                minimum_sampled_tree_weight = math.inf
(4)                for _ in range(2 * ceil(ln(G.number_of_nodes()))):
(8)                    sampled_tree = random_spanning_tree(z_support, "weight", seed=seed)
(8)                    sampled_tree_weight = sampled_tree.size(weight)
(8)                    if sampled_tree_weight < minimum_sampled_tree_weight:
(12)                       minimum_sampled_tree = sampled_tree.copy()
(12)                       minimum_sampled_tree_weight = sampled_tree_weight
(4)                t_star = nx.MultiDiGraph()
(4)                for u, v, d in minimum_sampled_tree.edges(data=weight):
(8)                    if d == G[u][v][weight]:
(12)                       t_star.add_edge(u, v, **{weight: d})
(8)                    else:
(12)                       t_star.add_edge(v, u, **{weight: d})
(4)                node_demands = {n: t_star.out_degree(n) - t_star.in_degree(n) for n in t_star}
(4)                nx.set_node_attributes(G, node_demands, "demand")
(4)                flow_dict = nx.min_cost_flow(G, "demand")
(4)                for source, values in flow_dict.items():
(8)                    for target in values:
(12)                       if (source, target) not in t_star.edges and values[target] > 0:
(16)                           for _ in range(values[target]):
(20)                               t_star.add_edge(source, target)
(4)                circuit = nx.eulerian_circuit(t_star, source=source)
(4)                return _shortcutting(circuit)
(0)            @nx._dispatchable(edge_attrs="weight", mutates_input=True, returns_graph=True)
(0)            def held_karp_ascent(G, weight="weight"):
(4)                """
(4)                Minimizes the Held-Karp relaxation of the TSP for `G`
(4)                Solves the Held-Karp relaxation of the input complete digraph and scales
(4)                the output solution for use in the Asadpour [1]_ ASTP algorithm.
(4)                The Held-Karp relaxation defines the lower bound for solutions to the
(4)                ATSP, although it does return a fractional solution. This is used in the
(4)                Asadpour algorithm as an initial solution which is later rounded to a
(4)                integral tree within the spanning tree polytopes. This function solves
(4)                the relaxation with the branch and bound method in [2]_.
(4)                Parameters
(4)                ----------
```

```
    G : nx.DiGraph
        The graph should be a complete weighted directed graph.
        The distance between all paris of nodes should be included.
    weight : string, optional (default="weight")
        Edge data key corresponding to the edge weight.
        If any edge does not have this attribute the weight is set to 1.
    Returns
    -------
    OPT : float
        The cost for the optimal solution to the Held-Karp relaxation
    z : dict or nx.Graph
        A symmetrized and scaled version of the optimal solution to the
        Held-Karp relaxation for use in the Asadpour algorithm.
        If an integral solution is found, then that is an optimal solution for
        the ATSP problem and that is returned instead.
    References
    ----------
    .. [1] A. Asadpour, M. X. Goemans, A. Madry, S. O. Gharan, and A. Saberi,
       An o(log n/log log n)-approximation algorithm for the asymmetric
       traveling salesman problem, Operations research, 65 (2017),
       pp. 1043–1061
    .. [2] M. Held, R. M. Karp, The traveling-salesman problem and minimum
           spanning trees, Operations Research, 1970-11-01, Vol. 18 (6),
           pp.1138-1162
    """
    import numpy as np
    from scipy import optimize
    def k_pi():
        """
        Find the set of minimum 1-Arborescences for G at point pi.
        Returns
        -------
        Set
            The set of minimum 1-Arborescences
        """
        G_1 = G.copy()
        minimum_1_arborescences = set()
        minimum_1_arborescence_weight = math.inf
        n = next(G.__iter__())
        G_1.remove_node(n)
        min_root = {"node": None, weight: math.inf}
        max_root = {"node": None, weight: -math.inf}
        for u, v, d in G.edges(n, data=True):
            if d[weight] < min_root[weight]:
                min_root = {"node": v, weight: d[weight]}
            if d[weight] > max_root[weight]:
                max_root = {"node": v, weight: d[weight]}
        min_in_edge = min(G.in_edges(n, data=True), key=lambda x: x[2][weight])

        min_root[weight] = min_root[weight] + min_in_edge[2][weight]
        max_root[weight] = max_root[weight] + min_in_edge[2][weight]
        min_arb_weight = math.inf
        for arb in nx.ArborescenceIterator(G_1):
            arb_weight = arb.size(weight)
            if min_arb_weight == math.inf:
                min_arb_weight = arb_weight
            elif arb_weight > min_arb_weight + max_root[weight] - min_root[weight]:
                break
            for N, deg in arb.in_degree:
                if deg == 0:
                    arb.add_edge(n, N, **{weight: G[n][N][weight]})
                    arb_weight += G[n][N][weight]
                    break
            edge_data = G[N][n]
            G.remove_edge(N, n)
            min_weight = min(G.in_edges(n, data=weight), key=lambda x: x[2])[2]

            min_edges = [
```

```
474: (16)                            (u, v, d) for u, v, d in G.in_edges(n, data=weight) if d ==
min_weight
475: (12)                        ]
476: (12)                        for u, v, d in min_edges:
477: (16)                            new_arb = arb.copy()
478: (16)                            new_arb.add_edge(u, v, **{weight: d})
479: (16)                            new_arb_weight = arb_weight + d
480: (16)                            if new_arb_weight < minimum_1_arborescence_weight:
481: (20)                                minimum_1_arborescences.clear()
482: (20)                                minimum_1_arborescence_weight = new_arb_weight
483: (16)                            if new_arb_weight == minimum_1_arborescence_weight:
484: (20)                                minimum_1_arborescences.add(new_arb)
485: (12)                        G.add_edge(N, n, **edge_data)
486: (8)                    return minimum_1_arborescences
487: (4)                def direction_of_ascent():
488: (8)                    """
489: (8)                    Find the direction of ascent at point pi.
490: (8)                    See [1]_ for more information.
491: (8)                    Returns
492: (8)                    -------
493: (8)                    dict
494: (12)                       A mapping from the nodes of the graph which represents the
direction
495: (12)                       of ascent.
496: (8)                    References
497: (8)                    ----------
498: (8)                    .. [1] M. Held, R. M. Karp, The traveling-salesman problem and minimum
499: (11)                      spanning trees, Operations Research, 1970-11-01, Vol. 18 (6),
500: (11)                      pp.1138-1162
501: (8)                    """
502: (8)                    d = {}
503: (8)                    for n in G:
504: (12)                       d[n] = 0
505: (8)                    del n
506: (8)                    minimum_1_arborescences = k_pi()
507: (8)                    while True:
508: (12)                       min_k_d_weight = math.inf
509: (12)                       min_k_d = None
510: (12)                       for arborescence in minimum_1_arborescences:
511: (16)                           weighted_cost = 0
512: (16)                           for n, deg in arborescence.degree:
513: (20)                               weighted_cost += d[n] * (deg - 2)
514: (16)                           if weighted_cost < min_k_d_weight:
515: (20)                               min_k_d_weight = weighted_cost
516: (20)                               min_k_d = arborescence
517: (12)                       if min_k_d_weight > 0:
518: (16)                           return d, min_k_d
519: (12)                       for n, deg in min_k_d.degree:
520: (16)                           d[n] += deg - 2
521: (12)                       c = np.full(len(minimum_1_arborescences), -1, dtype=int)
522: (12)                       a_eq = np.empty((len(G) + 1, len(minimum_1_arborescences)),
dtype=int)
523: (12)                       b_eq = np.zeros(len(G) + 1, dtype=int)
524: (12)                       b_eq[len(G)] = 1
525: (12)                       for arb_count, arborescence in enumerate(minimum_1_arborescences):
526: (16)                           n_count = len(G) - 1
527: (16)                           for n, deg in arborescence.degree:
528: (20)                               a_eq[n_count][arb_count] = deg - 2
529: (20)                               n_count -= 1
530: (16)                           a_eq[len(G)][arb_count] = 1
531: (12)                       program_result = optimize.linprog(
532: (16)                           c, A_eq=a_eq, b_eq=b_eq, method="highs-ipm"
533: (12)                       )
534: (12)                       if program_result.success:
535: (16)                           return None, minimum_1_arborescences
536: (4)                def find_epsilon(k, d):
537: (8)                    """
538: (8)                    Given the direction of ascent at pi, find the maximum distance we can
go
```

```
539: (8)                    in that direction.
540: (8)                    Parameters
541: (8)                    ----------
542: (8)                    k_xy : set
543: (12)                       The set of 1-arborescences which have the minimum rate of increase
544: (12)                       in the direction of ascent
545: (8)                    d : dict
546: (12)                       The direction of ascent
547: (8)                    Returns
548: (8)                    -------
549: (8)                    float
550: (12)                       The distance we can travel in direction `d`
551: (8)                    """
552: (8)                    min_epsilon = math.inf
553: (8)                    for e_u, e_v, e_w in G.edges(data=weight):
554: (12)                       if (e_u, e_v) in k.edges:
555: (16)                           continue
556: (12)                       if len(k.in_edges(e_v, data=weight)) > 1:
557: (16)                           raise Exception
558: (12)                       sub_u, sub_v, sub_w = next(k.in_edges(e_v,
data=weight).__iter__())
559: (12)                       k.add_edge(e_u, e_v, **{weight: e_w})
560: (12)                       k.remove_edge(sub_u, sub_v)
561: (12)                       if (
562: (16)                           max(d for n, d in k.in_degree()) <= 1
563: (16)                           and len(G) == k.number_of_edges()
564: (16)                           and nx.is_weakly_connected(k)
565: (12)                       ):
566: (16)                           if d[sub_u] == d[e_u] or sub_w == e_w:
567: (20)                               k.remove_edge(e_u, e_v)
568: (20)                               k.add_edge(sub_u, sub_v, **{weight: sub_w})
569: (20)                               continue
570: (16)                           epsilon = (sub_w - e_w) / (d[e_u] - d[sub_u])
571: (16)                           if 0 < epsilon < min_epsilon:
572: (20)                               min_epsilon = epsilon
573: (12)                       k.remove_edge(e_u, e_v)
574: (12)                       k.add_edge(sub_u, sub_v, **{weight: sub_w})
575: (8)                    return min_epsilon
576: (4)                pi_dict = {}
577: (4)                for n in G:
578: (8)                    pi_dict[n] = 0
579: (4)                del n
580: (4)                original_edge_weights = {}
581: (4)                for u, v, d in G.edges(data=True):
582: (8)                    original_edge_weights[(u, v)] = d[weight]
583: (4)                dir_ascent, k_d = direction_of_ascent()
584: (4)                while dir_ascent is not None:
585: (8)                    max_distance = find_epsilon(k_d, dir_ascent)
586: (8)                    for n, v in dir_ascent.items():
587: (12)                       pi_dict[n] += max_distance * v
588: (8)                    for u, v, d in G.edges(data=True):
589: (12)                       d[weight] = original_edge_weights[(u, v)] + pi_dict[u]
590: (8)                    dir_ascent, k_d = direction_of_ascent()
591: (4)                nx._clear_cache(G)
592: (4)                k_max = k_d
593: (4)                for k in k_max:
594: (8)                    if len([n for n in k if k.degree(n) == 2]) == G.order():
595: (12)                       return k.size(weight), k
596: (4)                x_star = {}
597: (4)                size_k_max = len(k_max)
598: (4)                for u, v, d in G.edges(data=True):
599: (8)                    edge_count = 0
600: (8)                    d[weight] = original_edge_weights[(u, v)]
601: (8)                    for k in k_max:
602: (12)                       if (u, v) in k.edges():
603: (16)                           edge_count += 1
604: (16)                           k[u][v][weight] = original_edge_weights[(u, v)]
605: (8)                    x_star[(u, v)] = edge_count / size_k_max
606: (4)                z_star = {}
```

```
    scale_factor = (G.order() - 1) / G.order()
    for u, v in x_star:
        frequency = x_star[(u, v)] + x_star[(v, u)]
        if frequency > 0:
            z_star[(u, v)] = scale_factor * frequency
    del x_star
    return next(k_max.__iter__()).size(weight), z_star
@nx._dispatchable
def spanning_tree_distribution(G, z):
    """
    Find the asadpour exponential distribution of spanning trees.
    Solves the Maximum Entropy Convex Program in the Asadpour algorithm [1]_
    using the approach in section 7 to build an exponential distribution of
    undirected spanning trees.
    This algorithm ensures that the probability of any edge in a spanning
    tree is proportional to the sum of the probabilities of the tress
    containing that edge over the sum of the probabilities of all spanning
    trees of the graph.
    Parameters
    ----------
    G : nx.MultiGraph
        The undirected support graph for the Held Karp relaxation
    z : dict
        The output of `held_karp_ascent()`, a scaled version of the Held-Karp
        solution.
    Returns
    -------
    gamma : dict
        The probability distribution which approximately preserves the marginal
        probabilities of `z`.
    """
    from math import exp
    from math import log as ln
    def q(e):
        """
        The value of q(e) is described in the Asadpour paper is "the
        probability that edge e will be included in a spanning tree T that is
        chosen with probability proportional to exp(gamma(T))" which
        basically means that it is the total probability of the edge appearing
        across the whole distribution.
        Parameters
        ----------
        e : tuple
            The `(u, v)` tuple describing the edge we are interested in
        Returns
        -------
        float
            The probability that a spanning tree chosen according to the
            current values of gamma will include edge `e`.
        """
        for u, v, d in G.edges(data=True):
            d[lambda_key] = exp(gamma[(u, v)])
        G_Kirchhoff = nx.total_spanning_tree_weight(G, lambda_key)
        G_e = nx.contracted_edge(G, e, self_loops=False)
        G_e_Kirchhoff = nx.total_spanning_tree_weight(G_e, lambda_key)
        return exp(gamma[(e[0], e[1])]) * G_e_Kirchhoff / G_Kirchhoff
    gamma = {}
    for u, v, _ in G.edges:
        gamma[(u, v)] = 0
    EPSILON = 0.2
    lambda_key = "spanning_tree_distribution's secret attribute name for lambda"
    while True:
        in_range_count = 0
        for u, v in gamma:
            e = (u, v)
            q_e = q(e)
            z_e = z[e]
```

```
(12)                        if q_e > (1 + EPSILON) * z_e:
(16)                            delta = ln(
(20)                                (q_e * (1 - (1 + EPSILON / 2) * z_e))
(20)                                / ((1 - q_e) * (1 + EPSILON / 2) * z_e)
(16)                            )
(16)                            gamma[e] -= delta
(16)                            new_q_e = q(e)
(16)                            desired_q_e = (1 + EPSILON / 2) * z_e
(16)                            if round(new_q_e, 8) != round(desired_q_e, 8):
(20)                                raise nx.NetworkXError(
(24)                                    f"Unable to modify probability for edge ({u}, {v})"
(20)                                )
(12)                        else:
(16)                            in_range_count += 1
(8)                     if in_range_count == len(gamma):
(12)                        break
(4)                 for _, _, d in G.edges(data=True):
(8)                     if lambda_key in d:
(12)                        del d[lambda_key]
(4)                 return gamma
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def greedy_tsp(G, weight="weight", source=None):
(4)                 """Return a low cost cycle starting at `source` and its cost.
(4)                 This approximates a solution to the traveling salesman problem.
(4)                 It finds a cycle of all the nodes that a salesman can visit in order
(4)                 to visit many nodes while minimizing total distance.
(4)                 It uses a simple greedy algorithm.
(4)                 In essence, this function returns a large cycle given a source point
(4)                 for which the total cost of the cycle is minimized.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph
(8)                     The Graph should be a complete weighted undirected graph.
(8)                     The distance between all pairs of nodes should be included.
(4)                 weight : string, optional (default="weight")
(8)                     Edge data key corresponding to the edge weight.
(8)                     If any edge does not have this attribute the weight is set to 1.
(4)                 source : node, optional (default: first node in list(G))
(8)                     Starting node.  If None, defaults to ``next(iter(G))``
(4)                 Returns
(4)                 -------
(4)                 cycle : list of nodes
(8)                     Returns the cycle (list of nodes) that a salesman
(8)                     can follow to minimize total weight of the trip.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If `G` is not complete, the algorithm raises an exception.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import approximation as approx
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_weighted_edges_from(
(4)                 ...     {
(4)                 ...         ("A", "B", 3),
(4)                 ...         ("A", "C", 17),
(4)                 ...         ("A", "D", 14),
(4)                 ...         ("B", "A", 3),
(4)                 ...         ("B", "C", 12),
(4)                 ...         ("B", "D", 16),
(4)                 ...         ("C", "A", 13),
(4)                 ...         ("C", "B", 12),
(4)                 ...         ("C", "D", 4),
(4)                 ...         ("D", "A", 14),
(4)                 ...         ("D", "B", 15),
(4)                 ...         ("D", "C", 2),
(4)                 ...     }
(4)                 ... )
(4)                 >>> cycle = approx.greedy_tsp(G, source="D")
```

```
743: (4)                >>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
744: (4)                >>> cycle
745: (4)                ['D', 'C', 'B', 'A', 'D']
746: (4)                >>> cost
747: (4)                31
748: (4)                Notes
749: (4)                -----
750: (4)                This implementation of a greedy algorithm is based on the following:
751: (4)                - The algorithm adds a node to the solution at every iteration.
752: (4)                - The algorithm selects a node not already in the cycle whose connection
753: (6)                  to the previous node adds the least cost to the cycle.
754: (4)                A greedy algorithm does not always give the best solution.
755: (4)                However, it can construct a first feasible solution which can
756: (4)                be passed as a parameter to an iterative improvement algorithm such
757: (4)                as Simulated Annealing, or Threshold Accepting.
758: (4)                Time complexity: It has a running time $O(|V|^2)$
759: (4)                """
760: (4)                N = len(G) - 1
761: (4)                if any(len(nbrdict) - (n in nbrdict) != N for n, nbrdict in
G.adj.items()):
762: (8)                    raise nx.NetworkXError("G must be a complete graph.")
763: (4)                if source is None:
764: (8)                    source = nx.utils.arbitrary_element(G)
765: (4)                if G.number_of_nodes() == 2:
766: (8)                    neighbor = next(G.neighbors(source))
767: (8)                    return [source, neighbor, source]
768: (4)                nodeset = set(G)
769: (4)                nodeset.remove(source)
770: (4)                cycle = [source]
771: (4)                next_node = source
772: (4)                while nodeset:
773: (8)                    nbrdict = G[next_node]
774: (8)                    next_node = min(nodeset, key=lambda n: nbrdict[n].get(weight, 1))
775: (8)                    cycle.append(next_node)
776: (8)                    nodeset.remove(next_node)
777: (4)                cycle.append(cycle[0])
778: (4)                return cycle
779: (0)            @py_random_state(9)
780: (0)            @nx._dispatchable(edge_attrs="weight")
781: (0)            def simulated_annealing_tsp(
782: (4)                G,
783: (4)                init_cycle,
784: (4)                weight="weight",
785: (4)                source=None,
786: (4)                temp=100,
787: (4)                move="1-1",
788: (4)                max_iterations=10,
789: (4)                N_inner=100,
790: (4)                alpha=0.01,
791: (4)                seed=None,
792: (0)            ):
793: (4)                """Returns an approximate solution to the traveling salesman problem.
794: (4)                This function uses simulated annealing to approximate the minimal cost
795: (4)                cycle through the nodes. Starting from a suboptimal solution, simulated
796: (4)                annealing perturbs that solution, occasionally accepting changes that make
797: (4)                the solution worse to escape from a locally optimal solution. The chance
798: (4)                of accepting such changes decreases over the iterations to encourage
799: (4)                an optimal result.  In summary, the function returns a cycle starting
800: (4)                at `source` for which the total cost is minimized. It also returns the
cost.
801: (4)                The chance of accepting a proposed change is related to a parameter called
802: (4)                the temperature (annealing has a physical analogue of steel hardening
803: (4)                as it cools). As the temperature is reduced, the chance of moves that
804: (4)                increase cost goes down.
805: (4)                Parameters
806: (4)                ----------
807: (4)                G : Graph
808: (8)                    `G` should be a complete weighted graph.
809: (8)                    The distance between all pairs of nodes should be included.
```

```
(4)                 init_cycle : list of all nodes or "greedy"
(8)                     The initial solution (a cycle through all nodes returning to the start).
(8)                     This argument has no default to make you think about it.
(8)                     If "greedy", use `greedy_tsp(G, weight)`.
(8)                     Other common starting cycles are `list(G) + [next(iter(G))]` or the final
(8)                     result of `simulated_annealing_tsp` when doing `threshold_accepting_tsp`.
(4)                 weight : string, optional (default="weight")
(8)                     Edge data key corresponding to the edge weight.
(8)                     If any edge does not have this attribute the weight is set to 1.
(4)                 source : node, optional (default: first node in list(G))
(8)                     Starting node.  If None, defaults to ``next(iter(G))``
(4)                 temp : int, optional (default=100)
(8)                     The algorithm's temperature parameter. It represents the initial
(8)                     value of temperature
(4)                 move : "1-1" or "1-0" or function, optional (default="1-1")
(8)                     Indicator of what move to use when finding new trial solutions.
(8)                     Strings indicate two special built-in moves:
(8)                     - "1-1": 1-1 exchange which transposes the position
(10)                      of two elements of the current solution.
(10)                      The function called is :func:`swap_two_nodes`.
(10)                      For example if we apply 1-1 exchange in the solution
(10)                      ``A = [3, 2, 1, 4, 3]``
(10)                      we can get the following by the transposition of 1 and 4 elements:
(10)                      ``A' = [3, 2, 4, 1, 3]``
(8)                     - "1-0": 1-0 exchange which moves an node in the solution
(10)                      to a new position.
(10)                      The function called is :func:`move_one_node`.
(10)                      For example if we apply 1-0 exchange in the solution
(10)                      ``A = [3, 2, 1, 4, 3]``
(10)                      we can transfer the fourth element to the second position:
(10)                      ``A' = [3, 4, 2, 1, 3]``
(8)                     You may provide your own functions to enact a move from
(8)                     one solution to a neighbor solution. The function must take
(8)                     the solution as input along with a `seed` input to control
(8)                     random number generation (see the `seed` input here).
(8)                     Your function should maintain the solution as a cycle with
(8)                     equal first and last node and all others appearing once.
(8)                     Your function should return the new solution.
(4)                 max_iterations : int, optional (default=10)
(8)                     Declared done when this number of consecutive iterations of
(8)                     the outer loop occurs without any change in the best cost solution.
(4)                 N_inner : int, optional (default=100)
(8)                     The number of iterations of the inner loop.
(4)                 alpha : float between (0, 1), optional (default=0.01)
(8)                     Percentage of temperature decrease in each iteration
(8)                     of outer loop
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 cycle : list of nodes
(8)                     Returns the cycle (list of nodes) that a salesman
(8)                     can follow to minimize total weight of the trip.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If `G` is not complete the algorithm raises an exception.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import approximation as approx
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_weighted_edges_from(
(4)                 ...     {
(4)                 ...         ("A", "B", 3),
(4)                 ...         ("A", "C", 17),
```

```
(4)                 ...         ("A", "D", 14),
(4)                 ...         ("B", "A", 3),
(4)                 ...         ("B", "C", 12),
(4)                 ...         ("B", "D", 16),
(4)                 ...         ("C", "A", 13),
(4)                 ...         ("C", "B", 12),
(4)                 ...         ("C", "D", 4),
(4)                 ...         ("D", "A", 14),
(4)                 ...         ("D", "B", 15),
(4)                 ...         ("D", "C", 2),
(4)                 ...     }
(4)                 ... )
(4)                 >>> cycle = approx.simulated_annealing_tsp(G, "greedy", source="D")
(4)                 >>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
(4)                 >>> cycle
(4)                 ['D', 'C', 'B', 'A', 'D']
(4)                 >>> cost
(4)                 31
(4)                 >>> incycle = ["D", "B", "A", "C", "D"]
(4)                 >>> cycle = approx.simulated_annealing_tsp(G, incycle, source="D")
(4)                 >>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
(4)                 >>> cycle
(4)                 ['D', 'C', 'B', 'A', 'D']
(4)                 >>> cost
(4)                 31
(4)                 Notes
(4)                 -----
(4)                 Simulated Annealing is a metaheuristic local search algorithm.
(4)                 The main characteristic of this algorithm is that it accepts
(4)                 even solutions which lead to the increase of the cost in order
(4)                 to escape from low quality local optimal solutions.
(4)                 This algorithm needs an initial solution. If not provided, it is
(4)                 constructed by a simple greedy algorithm. At every iteration, the
(4)                 algorithm selects thoughtfully a neighbor solution.
(4)                 Consider $c(x)$ cost of current solution and $c(x')$ cost of a
(4)                 neighbor solution.
(4)                 If $c(x') - c(x) <= 0$ then the neighbor solution becomes the current
(4)                 solution for the next iteration. Otherwise, the algorithm accepts
(4)                 the neighbor solution with probability $p = exp - ([c(x') - c(x)] /
temp)$.
(4)                 Otherwise the current solution is retained.
(4)                 `temp` is a parameter of the algorithm and represents temperature.
(4)                 Time complexity:
(4)                 For $N_i$ iterations of the inner loop and $N_o$ iterations of the
(4)                 outer loop, this algorithm has running time $O(N_i * N_o * |V|)$.
(4)                 For more information and how the algorithm is inspired see:
(4)                 http://en.wikipedia.org/wiki/Simulated_annealing
(4)                 """
(4)                 if move == "1-1":
(8)                     move = swap_two_nodes
(4)                 elif move == "1-0":
(8)                     move = move_one_node
(4)                 if init_cycle == "greedy":
(8)                     cycle = greedy_tsp(G, weight=weight, source=source)
(8)                     if G.number_of_nodes() == 2:
(12)                        return cycle
(4)                 else:
(8)                     cycle = list(init_cycle)
(8)                     if source is None:
(12)                        source = cycle[0]
(8)                     elif source != cycle[0]:
(12)                        raise nx.NetworkXError("source must be first node in init_cycle")
(8)                     if cycle[0] != cycle[-1]:
(12)                        raise nx.NetworkXError("init_cycle must be a cycle. (return to
start)")
(8)                     if len(cycle) - 1 != len(G) or len(set(G.nbunch_iter(cycle))) !=
len(G):
(12)                        raise nx.NetworkXError("init_cycle should be a cycle over all
nodes in G.")
```

```
(8)                     N = len(G) - 1
(8)                     if any(len(nbrdict) - (n in nbrdict) != N for n, nbrdict in
G.adj.items()):
(12)                        raise nx.NetworkXError("G must be a complete graph.")
(8)                     if G.number_of_nodes() == 2:
(12)                        neighbor = next(G.neighbors(source))
(12)                        return [source, neighbor, source]
(4)                 cost = sum(G[u][v].get(weight, 1) for u, v in pairwise(cycle))
(4)                 count = 0
(4)                 best_cycle = cycle.copy()
(4)                 best_cost = cost
(4)                 while count <= max_iterations and temp > 0:
(8)                     count += 1
(8)                     for i in range(N_inner):
(12)                        adj_sol = move(cycle, seed)
(12)                        adj_cost = sum(G[u][v].get(weight, 1) for u, v in
pairwise(adj_sol))
(12)                        delta = adj_cost - cost
(12)                        if delta <= 0:
(16)                            cycle = adj_sol
(16)                            cost = adj_cost
(16)                            if cost < best_cost:
(20)                                count = 0
(20)                                best_cycle = cycle.copy()
(20)                                best_cost = cost
(12)                        else:
(16)                            p = math.exp(-delta / temp)
(16)                            if p >= seed.random():
(20)                                cycle = adj_sol
(20)                                cost = adj_cost
(8)                     temp -= temp * alpha
(4)                 return best_cycle
(0)             @py_random_state(9)
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def threshold_accepting_tsp(
(4)                 G,
(4)                 init_cycle,
(4)                 weight="weight",
(4)                 source=None,
(4)                 threshold=1,
(4)                 move="1-1",
(4)                 max_iterations=10,
(4)                 N_inner=100,
(4)                 alpha=0.1,
(4)                 seed=None,
(0)             ):
(4)                 """Returns an approximate solution to the traveling salesman problem.
(4)                 This function uses threshold accepting methods to approximate the minimal
cost
(4)                 cycle through the nodes. Starting from a suboptimal solution, threshold
(4)                 accepting methods perturb that solution, accepting any changes that make
(4)                 the solution no worse than increasing by a threshold amount. Improvements
(4)                 in cost are accepted, but so are changes leading to small increases in
cost.
(4)                 This allows the solution to leave suboptimal local minima in solution
space.
(4)                 The threshold is decreased slowly as iterations proceed helping to ensure
(4)                 an optimum. In summary, the function returns a cycle starting at `source`
(4)                 for which the total cost is minimized.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph
(8)                     `G` should be a complete weighted graph.
(8)                     The distance between all pairs of nodes should be included.
(4)                 init_cycle : list or "greedy"
(8)                     The initial solution (a cycle through all nodes returning to the
start).
(8)                     This argument has no default to make you think about it.
(8)                     If "greedy", use `greedy_tsp(G, weight)`.
```

```
(8)                 Other common starting cycles are `list(G) + [next(iter(G))]` or the final
(8)                 result of `simulated_annealing_tsp` when doing `threshold_accepting_tsp`.
(4)             weight : string, optional (default="weight")
(8)                 Edge data key corresponding to the edge weight.
(8)                 If any edge does not have this attribute the weight is set to 1.
(4)             source : node, optional (default: first node in list(G))
(8)                 Starting node.  If None, defaults to ``next(iter(G))``
(4)             threshold : int, optional (default=1)
(8)                 The algorithm's threshold parameter. It represents the initial
(8)                 threshold's value
(4)             move : "1-1" or "1-0" or function, optional (default="1-1")
(8)                 Indicator of what move to use when finding new trial solutions.
(8)                 Strings indicate two special built-in moves:
(8)                 - "1-1": 1-1 exchange which transposes the position
(10)                  of two elements of the current solution.
(10)                  The function called is :func:`swap_two_nodes`.
(10)                  For example if we apply 1-1 exchange in the solution
(10)                  ``A = [3, 2, 1, 4, 3]``
(10)                  we can get the following by the transposition of 1 and 4 elements:
(10)                  ``A' = [3, 2, 4, 1, 3]``
(8)                 - "1-0": 1-0 exchange which moves an node in the solution
(10)                  to a new position.
(10)                  The function called is :func:`move_one_node`.
(10)                  For example if we apply 1-0 exchange in the solution
(10)                  ``A = [3, 2, 1, 4, 3]``
(10)                  we can transfer the fourth element to the second position:
(10)                  ``A' = [3, 4, 2, 1, 3]``
(8)                 You may provide your own functions to enact a move from
(8)                 one solution to a neighbor solution. The function must take
(8)                 the solution as input along with a `seed` input to control
(8)                 random number generation (see the `seed` input here).
(8)                 Your function should maintain the solution as a cycle with
(8)                 equal first and last node and all others appearing once.
(8)                 Your function should return the new solution.
(4)             max_iterations : int, optional (default=10)
(8)                 Declared done when this number of consecutive iterations of
(8)                 the outer loop occurs without any change in the best cost solution.
(4)             N_inner : int, optional (default=100)
(8)                 The number of iterations of the inner loop.
(4)             alpha : float between (0, 1), optional (default=0.1)
(8)                 Percentage of threshold decrease when there is at
(8)                 least one acceptance of a neighbor solution.
(8)                 If no inner loop moves are accepted the threshold remains unchanged.
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             cycle : list of nodes
(8)                 Returns the cycle (list of nodes) that a salesman
(8)                 can follow to minimize total weight of the trip.
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If `G` is not complete the algorithm raises an exception.
(4)             Examples
(4)             --------
(4)             >>> from networkx.algorithms import approximation as approx
(4)             >>> G = nx.DiGraph()
(4)             >>> G.add_weighted_edges_from(
(4)             ...     {
(4)             ...         ("A", "B", 3),
(4)             ...         ("A", "C", 17),
(4)             ...         ("A", "D", 14),
(4)             ...         ("B", "A", 3),
(4)             ...         ("B", "C", 12),
(4)             ...         ("B", "D", 16),
```

```
(4)                 ...         ("C", "A", 13),
(4)                 ...         ("C", "B", 12),
(4)                 ...         ("C", "D", 4),
(4)                 ...         ("D", "A", 14),
(4)                 ...         ("D", "B", 15),
(4)                 ...         ("D", "C", 2),
(4)                 ...     }
(4)                 ... )
(4)                 >>> cycle = approx.threshold_accepting_tsp(G, "greedy", source="D")
(4)                 >>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
(4)                 >>> cycle
(4)                 ['D', 'C', 'B', 'A', 'D']
(4)                 >>> cost
(4)                 31
(4)                 >>> incycle = ["D", "B", "A", "C", "D"]
(4)                 >>> cycle = approx.threshold_accepting_tsp(G, incycle, source="D")
(4)                 >>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
(4)                 >>> cycle
(4)                 ['D', 'C', 'B', 'A', 'D']
(4)                 >>> cost
(4)                 31
(4)                 Notes
(4)                 -----
(4)                 Threshold Accepting is a metaheuristic local search algorithm.
(4)                 The main characteristic of this algorithm is that it accepts
(4)                 even solutions which lead to the increase of the cost in order
(4)                 to escape from low quality local optimal solutions.
(4)                 This algorithm needs an initial solution. This solution can be
(4)                 constructed by a simple greedy algorithm. At every iteration, it
(4)                 selects thoughtfully a neighbor solution.
(4)                 Consider $c(x)$ cost of current solution and $c(x')$ cost of
(4)                 neighbor solution.
(4)                 If $c(x') - c(x) <= threshold$ then the neighbor solution becomes the
current
(4)                 solution for the next iteration, where the threshold is named threshold.
(4)                 In comparison to the Simulated Annealing algorithm, the Threshold
(4)                 Accepting algorithm does not accept very low quality solutions
(4)                 (due to the presence of the threshold value). In the case of
(4)                 Simulated Annealing, even a very low quality solution can
(4)                 be accepted with probability $p$.
(4)                 Time complexity:
(4)                 It has a running time $O(m * n * |V|)$ where $m$ and $n$ are the number
(4)                 of times the outer and inner loop run respectively.
(4)                 For more information and how algorithm is inspired see:
(4)                 https://doi.org/10.1016/0021-9991(90)90201-B
(4)                 See Also
(4)                 --------
(4)                 simulated_annealing_tsp
(4)                 """
(4)                 if move == "1-1":
(8)                     move = swap_two_nodes
(4)                 elif move == "1-0":
(8)                     move = move_one_node
(4)                 if init_cycle == "greedy":
(8)                     cycle = greedy_tsp(G, weight=weight, source=source)
(8)                     if G.number_of_nodes() == 2:
(12)                        return cycle
(4)                 else:
(8)                     cycle = list(init_cycle)
(8)                     if source is None:
(12)                        source = cycle[0]
(8)                     elif source != cycle[0]:
(12)                        raise nx.NetworkXError("source must be first node in init_cycle")
(8)                     if cycle[0] != cycle[-1]:
(12)                        raise nx.NetworkXError("init_cycle must be a cycle. (return to
start)")
(8)                     if len(cycle) - 1 != len(G) or len(set(G.nbunch_iter(cycle))) !=
len(G):
(12)                        raise nx.NetworkXError("init_cycle is not all and only nodes.")
```

```
1137: (8)                   N = len(G) - 1
1138: (8)                   if any(len(nbrdict) - (n in nbrdict) != N for n, nbrdict in
G.adj.items()):
1139: (12)                      raise nx.NetworkXError("G must be a complete graph.")
1140: (8)                   if G.number_of_nodes() == 2:
1141: (12)                      neighbor = list(G.neighbors(source))[0]
1142: (12)                      return [source, neighbor, source]
1143: (4)               cost = sum(G[u][v].get(weight, 1) for u, v in pairwise(cycle))
1144: (4)               count = 0
1145: (4)               best_cycle = cycle.copy()
1146: (4)               best_cost = cost
1147: (4)               while count <= max_iterations:
1148: (8)                   count += 1
1149: (8)                   accepted = False
1150: (8)                   for i in range(N_inner):
1151: (12)                      adj_sol = move(cycle, seed)
1152: (12)                      adj_cost = sum(G[u][v].get(weight, 1) for u, v in
pairwise(adj_sol))
1153: (12)                      delta = adj_cost - cost
1154: (12)                      if delta <= threshold:
1155: (16)                          accepted = True
1156: (16)                          cycle = adj_sol
1157: (16)                          cost = adj_cost
1158: (16)                          if cost < best_cost:
1159: (20)                              count = 0
1160: (20)                              best_cycle = cycle.copy()
1161: (20)                              best_cost = cost
1162: (8)                   if accepted:
1163: (12)                      threshold -= threshold * alpha
1164: (4)               return best_cycle

----------------------------------------

File 76 - treewidth.py:

1: (0)              """Functions for computing treewidth decomposition.
2: (0)              Treewidth of an undirected graph is a number associated with the graph.
3: (0)              It can be defined as the size of the largest vertex set (bag) in a tree
4: (0)              decomposition of the graph minus one.
5: (0)              `Wikipedia: Treewidth <https://en.wikipedia.org/wiki/Treewidth>`_
6: (0)              The notions of treewidth and tree decomposition have gained their
7: (0)              attractiveness partly because many graph and network problems that are
8: (0)              intractable (e.g., NP-hard) on arbitrary graphs become efficiently
9: (0)              solvable (e.g., with a linear time algorithm) when the treewidth of the
10: (0)             input graphs is bounded by a constant [1]_ [2]_.
11: (0)             There are two different functions for computing a tree decomposition:
12: (0)             :func:`treewidth_min_degree` and :func:`treewidth_min_fill_in`.
13: (0)             .. [1] Hans L. Bodlaender and Arie M. C. A. Koster. 2010. "Treewidth
14: (6)                   computations I.Upper bounds". Inf. Comput. 208, 3 (March 2010),259-275.
15: (6)                   http://dx.doi.org/10.1016/j.ic.2009.03.008
16: (0)             .. [2] Hans L. Bodlaender. "Discovering Treewidth". Institute of Information
17: (6)                   and Computing Sciences, Utrecht University.
18: (6)                   Technical Report UU-CS-2005-018.
19: (6)                   http://www.cs.uu.nl
20: (0)             .. [3] K. Wang, Z. Lu, and J. Hicks *Treewidth*.
21: (6)
https://web.archive.org/web/20210507025929/http://web.eecs.utk.edu/~cphill25/cs594_spring2015_proj
ects/treewidth.pdf
22: (0)             """
23: (0)             import itertools
24: (0)             import sys
25: (0)             from heapq import heapify, heappop, heappush
26: (0)             import networkx as nx
27: (0)             from networkx.utils import not_implemented_for
28: (0)             __all__ = ["treewidth_min_degree", "treewidth_min_fill_in"]
29: (0)             @not_implemented_for("directed")
30: (0)             @not_implemented_for("multigraph")
31: (0)             @nx._dispatchable(returns_graph=True)
32: (0)             def treewidth_min_degree(G):
```

```
(4)                 """Returns a treewidth decomposition using the Minimum Degree heuristic.
(4)                 The heuristic chooses the nodes according to their degree, i.e., first
(4)                 the node with the lowest degree is chosen, then the graph is updated
(4)                 and the corresponding node is removed. Next, a new node with the lowest
(4)                 degree is chosen, and so on.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 Returns
(4)                 -------
(4)                 Treewidth decomposition : (int, Graph) tuple
(10)                      2-tuple with treewidth and the corresponding decomposed tree.
(4)                 """
(4)                 deg_heuristic = MinDegreeHeuristic(G)
(4)                 return treewidth_decomp(G, lambda graph: deg_heuristic.best_node(graph))
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(returns_graph=True)
(0)             def treewidth_min_fill_in(G):
(4)                 """Returns a treewidth decomposition using the Minimum Fill-in heuristic.
(4)                 The heuristic chooses a node from the graph, where the number of edges
(4)                 added turning the neighborhood of the chosen node into clique is as
(4)                 small as possible.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 Returns
(4)                 -------
(4)                 Treewidth decomposition : (int, Graph) tuple
(8)                     2-tuple with treewidth and the corresponding decomposed tree.
(4)                 """
(4)                 return treewidth_decomp(G, min_fill_in_heuristic)
(0)             class MinDegreeHeuristic:
(4)                 """Implements the Minimum Degree heuristic.
(4)                 The heuristic chooses the nodes according to their degree
(4)                 (number of neighbors), i.e., first the node with the lowest degree is
(4)                 chosen, then the graph is updated and the corresponding node is
(4)                 removed. Next, a new node with the lowest degree is chosen, and so on.
(4)                 """
(4)                 def __init__(self, graph):
(8)                     self._graph = graph
(8)                     self._update_nodes = []
(8)                     self._degreeq = []  # a heapq with 3-tuples (degree,unique_id,node)
(8)                     self.count = itertools.count()
(8)                     for n in graph:
(12)                        self._degreeq.append((len(graph[n]), next(self.count), n))
(8)                     heapify(self._degreeq)
(4)                 def best_node(self, graph):
(8)                     for n in self._update_nodes:
(12)                        heappush(self._degreeq, (len(graph[n]), next(self.count), n))
(8)                     while self._degreeq:
(12)                        (min_degree, _, elim_node) = heappop(self._degreeq)
(12)                        if elim_node not in graph or len(graph[elim_node]) != min_degree:
(16)                            continue
(12)                        elif min_degree == len(graph) - 1:
(16)                            return None
(12)                        self._update_nodes = graph[elim_node]
(12)                        return elim_node
(8)                     return None
(0)             def min_fill_in_heuristic(graph):
(4)                 """Implements the Minimum Degree heuristic.
(4)                 Returns the node from the graph, where the number of edges added when
(4)                 turning the neighborhood of the chosen node into clique is as small as
(4)                 possible. This algorithm chooses the nodes using the Minimum Fill-In
(4)                 heuristic. The running time of the algorithm is :math:`O(V^3)` and it uses
(4)                 additional constant memory."""
(4)                 if len(graph) == 0:
(8)                     return None
(4)                 min_fill_in_node = None
```

```
102: (4)                min_fill_in = sys.maxsize
103: (4)                nodes_by_degree = sorted(graph, key=lambda x: len(graph[x]))
104: (4)                min_degree = len(graph[nodes_by_degree[0]])
105: (4)                if min_degree == len(graph) - 1:
106: (8)                    return None
107: (4)                for node in nodes_by_degree:
108: (8)                    num_fill_in = 0
109: (8)                    nbrs = graph[node]
110: (8)                    for nbr in nbrs:
111: (12)                       num_fill_in += len(nbrs - graph[nbr]) - 1
112: (12)                       if num_fill_in >= 2 * min_fill_in:
113: (16)                           break
114: (8)                    num_fill_in /= 2  # divide by 2 because of double counting
115: (8)                    if num_fill_in < min_fill_in:  # update min-fill-in node
116: (12)                       if num_fill_in == 0:
117: (16)                           return node
118: (12)                       min_fill_in = num_fill_in
119: (12)                       min_fill_in_node = node
120: (4)                return min_fill_in_node
121: (0)            @nx._dispatchable(returns_graph=True)
122: (0)            def treewidth_decomp(G, heuristic=min_fill_in_heuristic):
123: (4)                """Returns a treewidth decomposition using the passed heuristic.
124: (4)                Parameters
125: (4)                ----------
126: (4)                G : NetworkX graph
127: (4)                heuristic : heuristic function
128: (4)                Returns
129: (4)                -------
130: (4)                Treewidth decomposition : (int, Graph) tuple
131: (8)                    2-tuple with treewidth and the corresponding decomposed tree.
132: (4)                """
133: (4)                graph = {n: set(G[n]) - {n} for n in G}
134: (4)                node_stack = []
135: (4)                elim_node = heuristic(graph)
136: (4)                while elim_node is not None:
137: (8)                    nbrs = graph[elim_node]
138: (8)                    for u, v in itertools.permutations(nbrs, 2):
139: (12)                       if v not in graph[u]:
140: (16)                           graph[u].add(v)
141: (8)                    node_stack.append((elim_node, nbrs))
142: (8)                    for u in graph[elim_node]:
143: (12)                       graph[u].remove(elim_node)
144: (8)                    del graph[elim_node]
145: (8)                    elim_node = heuristic(graph)
146: (4)                decomp = nx.Graph()
147: (4)                first_bag = frozenset(graph.keys())
148: (4)                decomp.add_node(first_bag)
149: (4)                treewidth = len(first_bag) - 1
150: (4)                while node_stack:
151: (8)                    (curr_node, nbrs) = node_stack.pop()
152: (8)                    old_bag = None
153: (8)                    for bag in decomp.nodes:
154: (12)                       if nbrs <= bag:
155: (16)                           old_bag = bag
156: (16)                           break
157: (8)                    if old_bag is None:
158: (12)                       old_bag = first_bag
159: (8)                    nbrs.add(curr_node)
160: (8)                    new_bag = frozenset(nbrs)
161: (8)                    treewidth = max(treewidth, len(new_bag) - 1)
162: (8)                    decomp.add_edge(old_bag, new_bag)
163: (4)                return treewidth, decomp
```

File 77 - vertex_cover.py:

```
1: (0)              """Functions for computing an approximate minimum weight vertex cover.
2: (0)              A |vertex cover|_ is a subset of nodes such that each edge in the graph
```

```
3: (0)              is incident to at least one node in the subset.
4: (0)              .. _vertex cover: https://en.wikipedia.org/wiki/Vertex_cover
5: (0)              .. |vertex cover| replace:: *vertex cover*
6: (0)              """
7: (0)              import networkx as nx
8: (0)              __all__ = ["min_weighted_vertex_cover"]
9: (0)              @nx._dispatchable(node_attrs="weight")
10: (0)             def min_weighted_vertex_cover(G, weight=None):
11: (4)                 r"""Returns an approximate minimum weighted vertex cover.
12: (4)                 The set of nodes returned by this function is guaranteed to be a
13: (4)                 vertex cover, and the total weight of the set is guaranteed to be at
14: (4)                 most twice the total weight of the minimum weight vertex cover. In
15: (4)                 other words,
16: (4)                 .. math::
17: (7)                    w(S) \leq 2 * w(S^*),
18: (4)                 where $S$ is the vertex cover returned by this function,
19: (4)                 $S^*$ is the vertex cover of minimum weight out of all vertex
20: (4)                 covers of the graph, and $w$ is the function that computes the
21: (4)                 sum of the weights of each node in that given set.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 G : NetworkX graph
25: (4)                 weight : string, optional (default = None)
26: (8)                     If None, every node has weight 1. If a string, use this node
27: (8)                     attribute as the node weight. A node without this attribute is
28: (8)                     assumed to have weight 1.
29: (4)                 Returns
30: (4)                 -------
31: (4)                 min_weighted_cover : set
32: (8)                     Returns a set of nodes whose weight sum is no more than twice
33: (8)                     the weight sum of the minimum weight vertex cover.
34: (4)                 Notes
35: (4)                 -----
36: (4)                 For a directed graph, a vertex cover has the same definition: a set
37: (4)                 of nodes such that each edge in the graph is incident to at least
38: (4)                 one node in the set. Whether the node is the head or tail of the
39: (4)                 directed edge is ignored.
40: (4)                 This is the local-ratio algorithm for computing an approximate
41: (4)                 vertex cover. The algorithm greedily reduces the costs over edges,
42: (4)                 iteratively building a cover. The worst-case runtime of this
43: (4)                 implementation is $O(m \log n)$, where $n$ is the number
44: (4)                 of nodes and $m$ the number of edges in the graph.
45: (4)                 References
46: (4)                 ----------
47: (4)                 .. [1] Bar-Yehuda, R., and Even, S. (1985). "A local-ratio theorem for
48: (7)                    approximating the weighted vertex cover problem."
49: (7)                    *Annals of Discrete Mathematics*, 25, 27–46
50: (7)                    <http://www.cs.technion.ac.il/~reuven/PDF/vc_lr.pdf>
51: (4)                 """
52: (4)                 cost = dict(G.nodes(data=weight, default=1))
53: (4)                 cover = set()
54: (4)                 for u, v in G.edges():
55: (8)                     if u in cover or v in cover:
56: (12)                        continue
57: (8)                     if cost[u] <= cost[v]:
58: (12)                        cover.add(u)
59: (12)                        cost[v] -= cost[u]
60: (8)                     else:
61: (12)                        cover.add(v)
62: (12)                        cost[u] -= cost[v]
63: (4)                 return cover
```

File 78 - __init__.py:

```
1: (0)
```

File 79 - test_approx_clust_coeff.py:

```
1: (0)              import networkx as nx
2: (0)              from networkx.algorithms.approximation import average_clustering
3: (0)              def test_petersen():
4: (4)                  G = nx.petersen_graph()
5: (4)                  assert average_clustering(G, trials=len(G) // 2) ==
nx.average_clustering(G)
6: (0)              def test_petersen_seed():
7: (4)                  G = nx.petersen_graph()
8: (4)                  assert average_clustering(G, trials=len(G) // 2, seed=1) ==
nx.average_clustering(G)
9: (0)              def test_tetrahedral():
10: (4)                 G = nx.tetrahedral_graph()
11: (4)                 assert average_clustering(G, trials=len(G) // 2) ==
nx.average_clustering(G)
12: (0)             def test_dodecahedral():
13: (4)                 G = nx.dodecahedral_graph()
14: (4)                 assert average_clustering(G, trials=len(G) // 2) ==
nx.average_clustering(G)
15: (0)             def test_empty():
16: (4)                 G = nx.empty_graph(5)
17: (4)                 assert average_clustering(G, trials=len(G) // 2) == 0
18: (0)             def test_complete():
19: (4)                 G = nx.complete_graph(5)
20: (4)                 assert average_clustering(G, trials=len(G) // 2) == 1
21: (4)                 G = nx.complete_graph(7)
22: (4)                 assert average_clustering(G, trials=len(G) // 2) == 1
```

File 80 - test_clique.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.approximation.clique`
module."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.approximation import (
4: (4)                  clique_removal,
5: (4)                  large_clique_size,
6: (4)                  max_clique,
7: (4)                  maximum_independent_set,
8: (0)              )
9: (0)              def is_independent_set(G, nodes):
10: (4)                 """Returns True if and only if `nodes` is a clique in `G`.
11: (4)                 `G` is a NetworkX graph. `nodes` is an iterable of nodes in
12: (4)                 `G`.
13: (4)                 """
14: (4)                 return G.subgraph(nodes).number_of_edges() == 0
15: (0)             def is_clique(G, nodes):
16: (4)                 """Returns True if and only if `nodes` is an independent set
17: (4)                 in `G`.
18: (4)                 `G` is an undirected simple graph. `nodes` is an iterable of
19: (4)                 nodes in `G`.
20: (4)                 """
21: (4)                 H = G.subgraph(nodes)
22: (4)                 n = len(H)
23: (4)                 return H.number_of_edges() == n * (n - 1) // 2
24: (0)             class TestCliqueRemoval:
25: (4)                 """Unit tests for the
26: (4)                 :func:`~networkx.algorithms.approximation.clique_removal` function.
27: (4)                 """
28: (4)                 def test_trivial_graph(self):
29: (8)                     G = nx.trivial_graph()
30: (8)                     independent_set, cliques = clique_removal(G)
31: (8)                     assert is_independent_set(G, independent_set)
32: (8)                     assert all(is_clique(G, clique) for clique in cliques)
33: (8)                     assert all(len(clique) == 1 for clique in cliques)
34: (4)                 def test_complete_graph(self):
```

```
35: (8)                     G = nx.complete_graph(10)
36: (8)                     independent_set, cliques = clique_removal(G)
37: (8)                     assert is_independent_set(G, independent_set)
38: (8)                     assert all(is_clique(G, clique) for clique in cliques)
39: (4)                 def test_barbell_graph(self):
40: (8)                     G = nx.barbell_graph(10, 5)
41: (8)                     independent_set, cliques = clique_removal(G)
42: (8)                     assert is_independent_set(G, independent_set)
43: (8)                     assert all(is_clique(G, clique) for clique in cliques)
44: (0)             class TestMaxClique:
45: (4)                 """Unit tests for the :func:`networkx.algorithms.approximation.max_clique`
46: (4)                 function.
47: (4)                 """
48: (4)                 def test_null_graph(self):
49: (8)                     G = nx.null_graph()
50: (8)                     assert len(max_clique(G)) == 0
51: (4)                 def test_complete_graph(self):
52: (8)                     graph = nx.complete_graph(30)
53: (8)                     mc = max_clique(graph)
54: (8)                     assert 30 == len(mc)
55: (4)                 def test_maximal_by_cardinality(self):
56: (8)                     """Tests that the maximal clique is computed according to maximum
57: (8)                     cardinality of the sets.
58: (8)                     For more information, see pull request #1531.
59: (8)                     """
60: (8)                     G = nx.complete_graph(5)
61: (8)                     G.add_edge(4, 5)
62: (8)                     clique = max_clique(G)
63: (8)                     assert len(clique) > 1
64: (8)                     G = nx.lollipop_graph(30, 2)
65: (8)                     clique = max_clique(G)
66: (8)                     assert len(clique) > 2
67: (0)             def test_large_clique_size():
68: (4)                 G = nx.complete_graph(9)
69: (4)                 nx.add_cycle(G, [9, 10, 11])
70: (4)                 G.add_edge(8, 9)
71: (4)                 G.add_edge(1, 12)
72: (4)                 G.add_node(13)
73: (4)                 assert large_clique_size(G) == 9
74: (4)                 G.remove_node(5)
75: (4)                 assert large_clique_size(G) == 8
76: (4)                 G.remove_edge(2, 3)
77: (4)                 assert large_clique_size(G) == 7
78: (0)             def test_independent_set():
79: (4)                 G = nx.Graph()
80: (4)                 assert len(maximum_independent_set(G)) == 0
```

File 81 - test_connectivity.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import approximation as approx
4: (0)              def test_global_node_connectivity():
5: (4)                  G = nx.Graph()
6: (4)                  G.add_edges_from(
7: (8)                      [
8: (12)                         (1, 2),
9: (12)                         (1, 3),
10: (12)                        (1, 4),
11: (12)                        (1, 5),
12: (12)                        (2, 3),
13: (12)                        (2, 6),
14: (12)                        (3, 4),
15: (12)                        (3, 6),
16: (12)                        (4, 6),
17: (12)                        (4, 7),
18: (12)                        (5, 7),
```

```
(12)                        (6, 8),
(12)                        (6, 9),
(12)                        (7, 8),
(12)                        (7, 10),
(12)                        (8, 11),
(12)                        (9, 10),
(12)                        (9, 11),
(12)                        (10, 11),
(8)                     ]
(4)                 )
(4)                 assert 2 == approx.local_node_connectivity(G, 1, 11)
(4)                 assert 2 == approx.node_connectivity(G)
(4)                 assert 2 == approx.node_connectivity(G, 1, 11)
(0)             def test_white_harary1():
(4)                 G = nx.disjoint_union(nx.complete_graph(4), nx.complete_graph(4))
(4)                 G.remove_node(7)
(4)                 for i in range(4, 7):
(8)                     G.add_edge(0, i)
(4)                 G = nx.disjoint_union(G, nx.complete_graph(4))
(4)                 G.remove_node(G.order() - 1)
(4)                 for i in range(7, 10):
(8)                     G.add_edge(0, i)
(4)                 assert 1 == approx.node_connectivity(G)
(0)             def test_complete_graphs():
(4)                 for n in range(5, 25, 5):
(8)                     G = nx.complete_graph(n)
(8)                     assert n - 1 == approx.node_connectivity(G)
(8)                     assert n - 1 == approx.node_connectivity(G, 0, 3)
(0)             def test_empty_graphs():
(4)                 for k in range(5, 25, 5):
(8)                     G = nx.empty_graph(k)
(8)                     assert 0 == approx.node_connectivity(G)
(8)                     assert 0 == approx.node_connectivity(G, 0, 3)
(0)             def test_petersen():
(4)                 G = nx.petersen_graph()
(4)                 assert 3 == approx.node_connectivity(G)
(4)                 assert 3 == approx.node_connectivity(G, 0, 5)
(0)             def test_dodecahedral():
(4)                 G = nx.dodecahedral_graph()
(4)                 assert 3 == approx.node_connectivity(G)
(4)                 assert 3 == approx.node_connectivity(G, 0, 5)
(0)             def test_octahedral():
(4)                 G = nx.octahedral_graph()
(4)                 assert 4 == approx.node_connectivity(G)
(4)                 assert 4 == approx.node_connectivity(G, 0, 5)
(0)             def test_only_source():
(4)                 G = nx.complete_graph(5)
(4)                 pytest.raises(nx.NetworkXError, approx.node_connectivity, G, s=0)
(0)             def test_only_target():
(4)                 G = nx.complete_graph(5)
(4)                 pytest.raises(nx.NetworkXError, approx.node_connectivity, G, t=0)
(0)             def test_missing_source():
(4)                 G = nx.path_graph(4)
(4)                 pytest.raises(nx.NetworkXError, approx.node_connectivity, G, 10, 1)
(0)             def test_missing_target():
(4)                 G = nx.path_graph(4)
(4)                 pytest.raises(nx.NetworkXError, approx.node_connectivity, G, 1, 10)
(0)             def test_source_equals_target():
(4)                 G = nx.complete_graph(5)
(4)                 pytest.raises(nx.NetworkXError, approx.local_node_connectivity, G, 0, 0)
(0)             def test_directed_node_connectivity():
(4)                 G = nx.cycle_graph(10, create_using=nx.DiGraph())  # only one direction
(4)                 D = nx.cycle_graph(10).to_directed()  # 2 reciprocal edges
(4)                 assert 1 == approx.node_connectivity(G)
(4)                 assert 1 == approx.node_connectivity(G, 1, 4)
(4)                 assert 2 == approx.node_connectivity(D)
(4)                 assert 2 == approx.node_connectivity(D, 1, 4)
(0)             class TestAllPairsNodeConnectivityApprox:
(4)                 @classmethod
```

```
88: (4)                 def setup_class(cls):
89: (8)                     cls.path = nx.path_graph(7)
90: (8)                     cls.directed_path = nx.path_graph(7, create_using=nx.DiGraph())
91: (8)                     cls.cycle = nx.cycle_graph(7)
92: (8)                     cls.directed_cycle = nx.cycle_graph(7, create_using=nx.DiGraph())
93: (8)                     cls.gnp = nx.gnp_random_graph(30, 0.1)
94: (8)                     cls.directed_gnp = nx.gnp_random_graph(30, 0.1, directed=True)
95: (8)                     cls.K20 = nx.complete_graph(20)
96: (8)                     cls.K10 = nx.complete_graph(10)
97: (8)                     cls.K5 = nx.complete_graph(5)
98: (8)                     cls.G_list = [
99: (12)                        cls.path,
100: (12)                       cls.directed_path,
101: (12)                       cls.cycle,
102: (12)                       cls.directed_cycle,
103: (12)                       cls.gnp,
104: (12)                       cls.directed_gnp,
105: (12)                       cls.K10,
106: (12)                       cls.K5,
107: (12)                       cls.K20,
108: (8)                    ]
109: (4)                def test_cycles(self):
110: (8)                    K_undir = approx.all_pairs_node_connectivity(self.cycle)
111: (8)                    for source in K_undir:
112: (12)                       for target, k in K_undir[source].items():
113: (16)                           assert k == 2
114: (8)                    K_dir = approx.all_pairs_node_connectivity(self.directed_cycle)
115: (8)                    for source in K_dir:
116: (12)                       for target, k in K_dir[source].items():
117: (16)                           assert k == 1
118: (4)                def test_complete(self):
119: (8)                    for G in [self.K10, self.K5, self.K20]:
120: (12)                       K = approx.all_pairs_node_connectivity(G)
121: (12)                       for source in K:
122: (16)                           for target, k in K[source].items():
123: (20)                               assert k == len(G) - 1
124: (4)                def test_paths(self):
125: (8)                    K_undir = approx.all_pairs_node_connectivity(self.path)
126: (8)                    for source in K_undir:
127: (12)                       for target, k in K_undir[source].items():
128: (16)                           assert k == 1
129: (8)                    K_dir = approx.all_pairs_node_connectivity(self.directed_path)
130: (8)                    for source in K_dir:
131: (12)                       for target, k in K_dir[source].items():
132: (16)                           if source < target:
133: (20)                               assert k == 1
134: (16)                           else:
135: (20)                               assert k == 0
136: (4)                def test_cutoff(self):
137: (8)                    for G in [self.K10, self.K5, self.K20]:
138: (12)                       for mp in [2, 3, 4]:
139: (16)                           paths = approx.all_pairs_node_connectivity(G, cutoff=mp)
140: (16)                           for source in paths:
141: (20)                               for target, K in paths[source].items():
142: (24)                                   assert K == mp
143: (4)                def test_all_pairs_connectivity_nbunch(self):
144: (8)                    G = nx.complete_graph(5)
145: (8)                    nbunch = [0, 2, 3]
146: (8)                    C = approx.all_pairs_node_connectivity(G, nbunch=nbunch)
147: (8)                    assert len(C) == len(nbunch)
```

File 82 - test_distance_measures.py:

```
1: (0)              """Unit tests for the
:mod:`networkx.algorithms.approximation.distance_measures` module.
2: (0)              """
3: (0)              import pytest
```

```
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.approximation import diameter
6: (0)              class TestDiameter:
7: (4)                  """Unit tests for the approximate diameter function
8: (4)                  :func:`~networkx.algorithms.approximation.distance_measures.diameter`.
9: (4)                  """
10: (4)                 def test_null_graph(self):
11: (8)                     """Test empty graph."""
12: (8)                     G = nx.null_graph()
13: (8)                     with pytest.raises(
14: (12)                        nx.NetworkXError, match="Expected non-empty NetworkX graph!"
15: (8)                     ):
16: (12)                        diameter(G)
17: (4)                 def test_undirected_non_connected(self):
18: (8)                     """Test an undirected disconnected graph."""
19: (8)                     graph = nx.path_graph(10)
20: (8)                     graph.remove_edge(3, 4)
21: (8)                     with pytest.raises(nx.NetworkXError, match="Graph not connected."):
22: (12)                        diameter(graph)
23: (4)                 def test_directed_non_strongly_connected(self):
24: (8)                     """Test a directed non strongly connected graph."""
25: (8)                     graph = nx.path_graph(10, create_using=nx.DiGraph())
26: (8)                     with pytest.raises(nx.NetworkXError, match="DiGraph not strongly
connected."):
27: (12)                        diameter(graph)
28: (4)                 def test_complete_undirected_graph(self):
29: (8)                     """Test a complete undirected graph."""
30: (8)                     graph = nx.complete_graph(10)
31: (8)                     assert diameter(graph) == 1
32: (4)                 def test_complete_directed_graph(self):
33: (8)                     """Test a complete directed graph."""
34: (8)                     graph = nx.complete_graph(10, create_using=nx.DiGraph())
35: (8)                     assert diameter(graph) == 1
36: (4)                 def test_undirected_path_graph(self):
37: (8)                     """Test an undirected path graph with 10 nodes."""
38: (8)                     graph = nx.path_graph(10)
39: (8)                     assert diameter(graph) == 9
40: (4)                 def test_directed_path_graph(self):
41: (8)                     """Test a directed path graph with 10 nodes."""
42: (8)                     graph = nx.path_graph(10).to_directed()
43: (8)                     assert diameter(graph) == 9
44: (4)                 def test_single_node(self):
45: (8)                     """Test a graph which contains just a node."""
46: (8)                     graph = nx.Graph()
47: (8)                     graph.add_node(1)
48: (8)                     assert diameter(graph) == 0

----------------------------------------

File 83 - test_dominating_set.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.approximation import (
4: (4)                  min_edge_dominating_set,
5: (4)                  min_weighted_dominating_set,
6: (0)              )
7: (0)              class TestMinWeightDominatingSet:
8: (4)                  def test_min_weighted_dominating_set(self):
9: (8)                      graph = nx.Graph()
10: (8)                     graph.add_edge(1, 2)
11: (8)                     graph.add_edge(1, 5)
12: (8)                     graph.add_edge(2, 3)
13: (8)                     graph.add_edge(2, 5)
14: (8)                     graph.add_edge(3, 4)
15: (8)                     graph.add_edge(3, 6)
16: (8)                     graph.add_edge(5, 6)
17: (8)                     vertices = {1, 2, 3, 4, 5, 6}
18: (8)                     dom_set = min_weighted_dominating_set(graph)
```

```
19: (8)                     for vertex in vertices - dom_set:
20: (12)                        neighbors = set(graph.neighbors(vertex))
21: (12)                        assert len(neighbors & dom_set) > 0, "Non dominating set found!"
22: (4)                 def test_star_graph(self):
23: (8)                     """Tests that an approximate dominating set for the star graph,
24: (8)                     even when the center node does not have the smallest integer
25: (8)                     label, gives just the center node.
26: (8)                     For more information, see #1527.
27: (8)                     """
28: (8)                     G = nx.star_graph(10)
29: (8)                     G = nx.relabel_nodes(G, {0: 9, 9: 0})
30: (8)                     assert min_weighted_dominating_set(G) == {9}
31: (4)                 def test_null_graph(self):
32: (8)                     """Tests that the unique dominating set for the null graph is an empty
set"""
33: (8)                     G = nx.Graph()
34: (8)                     assert min_weighted_dominating_set(G) == set()
35: (4)                 def test_min_edge_dominating_set(self):
36: (8)                     graph = nx.path_graph(5)
37: (8)                     dom_set = min_edge_dominating_set(graph)
38: (8)                     for edge in graph.edges():
39: (12)                        if edge in dom_set:
40: (16)                            continue
41: (12)                        else:
42: (16)                            u, v = edge
43: (16)                            found = False
44: (16)                            for dom_edge in dom_set:
45: (20)                                found |= u == dom_edge[0] or u == dom_edge[1]
46: (16)                            assert found, "Non adjacent edge found!"
47: (8)                     graph = nx.complete_graph(10)
48: (8)                     dom_set = min_edge_dominating_set(graph)
49: (8)                     for edge in graph.edges():
50: (12)                        if edge in dom_set:
51: (16)                            continue
52: (12)                        else:
53: (16)                            u, v = edge
54: (16)                            found = False
55: (16)                            for dom_edge in dom_set:
56: (20)                                found |= u == dom_edge[0] or u == dom_edge[1]
57: (16)                            assert found, "Non adjacent edge found!"
58: (8)                     graph = nx.Graph()  # empty Networkx graph
59: (8)                     with pytest.raises(ValueError, match="Expected non-empty NetworkX
graph!"):
60: (12)                        min_edge_dominating_set(graph)

----------------------------------------

File 84 - test_kcomponents.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.approximation import k_components
4: (0)              from networkx.algorithms.approximation.kcomponents import _AntiGraph, _same
5: (0)              def build_k_number_dict(k_components):
6: (4)                  k_num = {}
7: (4)                  for k, comps in sorted(k_components.items()):
8: (8)                      for comp in comps:
9: (12)                         for node in comp:
10: (16)                            k_num[node] = k
11: (4)                 return k_num
12: (0)             def graph_example_1():
13: (4)                 G = nx.convert_node_labels_to_integers(
14: (8)                     nx.grid_graph([5, 5]), label_attribute="labels"
15: (4)                 )
16: (4)                 rlabels = nx.get_node_attributes(G, "labels")
17: (4)                 labels = {v: k for k, v in rlabels.items()}
18: (4)                 for nodes in [
19: (8)                     (labels[(0, 0)], labels[(1, 0)]),
20: (8)                     (labels[(0, 4)], labels[(1, 4)]),
```

```
 21: (8)                    (labels[(3, 0)], labels[(4, 0)]),
 22: (8)                    (labels[(3, 4)], labels[(4, 4)]),
 23: (4)                ]:
 24: (8)                    new_node = G.order() + 1
 25: (8)                    P = nx.petersen_graph()
 26: (8)                    G = nx.disjoint_union(G, P)
 27: (8)                    G.add_edge(new_node + 1, nodes[0])
 28: (8)                    G.add_edge(new_node, nodes[1])
 29: (8)                    K = nx.complete_graph(5)
 30: (8)                    G = nx.disjoint_union(G, K)
 31: (8)                    G.add_edge(new_node + 2, new_node + 11)
 32: (8)                    G.add_edge(new_node + 3, new_node + 12)
 33: (8)                    G.add_edge(new_node + 4, new_node + 13)
 34: (8)                    G = nx.disjoint_union(G, K)
 35: (8)                    nbrs = G[new_node + 10]
 36: (8)                    G.remove_node(new_node + 10)
 37: (8)                    for nbr in nbrs:
 38: (12)                       G.add_edge(new_node + 17, nbr)
 39: (8)                    G.add_edge(new_node + 16, new_node + 5)
 40: (4)                return G
 41: (0)            def torrents_and_ferraro_graph():
 42: (4)                G = nx.convert_node_labels_to_integers(
 43: (8)                    nx.grid_graph([5, 5]), label_attribute="labels"
 44: (4)                )
 45: (4)                rlabels = nx.get_node_attributes(G, "labels")
 46: (4)                labels = {v: k for k, v in rlabels.items()}
 47: (4)                for nodes in [(labels[(0, 4)], labels[(1, 4)]), (labels[(3, 4)],
labels[(4, 4)])]:
 48: (8)                    new_node = G.order() + 1
 49: (8)                    P = nx.petersen_graph()
 50: (8)                    G = nx.disjoint_union(G, P)
 51: (8)                    G.add_edge(new_node + 1, nodes[0])
 52: (8)                    G.add_edge(new_node, nodes[1])
 53: (8)                    K = nx.complete_graph(5)
 54: (8)                    G = nx.disjoint_union(G, K)
 55: (8)                    G.add_edge(new_node + 2, new_node + 11)
 56: (8)                    G.add_edge(new_node + 3, new_node + 12)
 57: (8)                    G.add_edge(new_node + 4, new_node + 13)
 58: (8)                    G = nx.disjoint_union(G, K)
 59: (8)                    nbrs = G[new_node + 10]
 60: (8)                    G.remove_node(new_node + 10)
 61: (8)                    for nbr in nbrs:
 62: (12)                       G.add_edge(new_node + 17, nbr)
 63: (8)                    G.add_edge(new_node + 16, new_node + 8)
 64: (4)                for nodes in [(labels[(0, 0)], labels[(1, 0)]), (labels[(3, 0)],
labels[(4, 0)])]:
 65: (8)                    new_node = G.order() + 1
 66: (8)                    P = nx.petersen_graph()
 67: (8)                    G = nx.disjoint_union(G, P)
 68: (8)                    G.add_edge(new_node + 1, nodes[0])
 69: (8)                    G.add_edge(new_node, nodes[1])
 70: (8)                    K = nx.complete_graph(5)
 71: (8)                    G = nx.disjoint_union(G, K)
 72: (8)                    G.add_edge(new_node + 2, new_node + 11)
 73: (8)                    G.add_edge(new_node + 3, new_node + 12)
 74: (8)                    G.add_edge(new_node + 4, new_node + 13)
 75: (8)                    G = nx.disjoint_union(G, K)
 76: (8)                    nbrs = G[new_node + 10]
 77: (8)                    G.remove_node(new_node + 10)
 78: (8)                    for nbr in nbrs:
 79: (12)                       G.add_edge(new_node + 17, nbr)
 80: (8)                    nbrs2 = G[new_node + 9]
 81: (8)                    G.remove_node(new_node + 9)
 82: (8)                    for nbr in nbrs2:
 83: (12)                       G.add_edge(new_node + 18, nbr)
 84: (4)                return G
 85: (0)            def _check_connectivity(G):
 86: (4)                result = k_components(G)
 87: (4)                for k, components in result.items():
```

```
88: (8)                     if k < 3:
89: (12)                        continue
90: (8)                     for component in components:
91: (12)                        C = G.subgraph(component)
92: (12)                        K = nx.node_connectivity(C)
93: (12)                        assert K >= k
94: (0)             def test_torrents_and_ferraro_graph():
95: (4)                 G = torrents_and_ferraro_graph()
96: (4)                 _check_connectivity(G)
97: (0)             def test_example_1():
98: (4)                 G = graph_example_1()
99: (4)                 _check_connectivity(G)
100: (0)            def test_karate_0():
101: (4)                G = nx.karate_club_graph()
102: (4)                _check_connectivity(G)
103: (0)            def test_karate_1():
104: (4)                karate_k_num = {
105: (8)                    0: 4,
106: (8)                    1: 4,
107: (8)                    2: 4,
108: (8)                    3: 4,
109: (8)                    4: 3,
110: (8)                    5: 3,
111: (8)                    6: 3,
112: (8)                    7: 4,
113: (8)                    8: 4,
114: (8)                    9: 2,
115: (8)                    10: 3,
116: (8)                    11: 1,
117: (8)                    12: 2,
118: (8)                    13: 4,
119: (8)                    14: 2,
120: (8)                    15: 2,
121: (8)                    16: 2,
122: (8)                    17: 2,
123: (8)                    18: 2,
124: (8)                    19: 3,
125: (8)                    20: 2,
126: (8)                    21: 2,
127: (8)                    22: 2,
128: (8)                    23: 3,
129: (8)                    24: 3,
130: (8)                    25: 3,
131: (8)                    26: 2,
132: (8)                    27: 3,
133: (8)                    28: 3,
134: (8)                    29: 3,
135: (8)                    30: 4,
136: (8)                    31: 3,
137: (8)                    32: 4,
138: (8)                    33: 4,
139: (4)                }
140: (4)                approx_karate_k_num = karate_k_num.copy()
141: (4)                approx_karate_k_num[24] = 2
142: (4)                approx_karate_k_num[25] = 2
143: (4)                G = nx.karate_club_graph()
144: (4)                k_comps = k_components(G)
145: (4)                k_num = build_k_number_dict(k_comps)
146: (4)                assert k_num in (karate_k_num, approx_karate_k_num)
147: (0)            def test_example_1_detail_3_and_4():
148: (4)                G = graph_example_1()
149: (4)                result = k_components(G)
150: (4)                assert len(result[3]) == 8
151: (4)                assert len([c for c in result[3] if len(c) == 15]) == 4
152: (4)                assert len([c for c in result[3] if len(c) == 5]) == 4
153: (4)                assert len(result[4]) == 8
154: (4)                assert all(len(c) == 5 for c in result[4])
155: (4)                for k, components in result.items():
156: (8)                    if k < 3:
```

```
(12)                        continue
(8)                     for component in components:
(12)                        K = nx.node_connectivity(G.subgraph(component))
(12)                        assert K >= k
(0)             def test_directed():
(4)                 with pytest.raises(nx.NetworkXNotImplemented):
(8)                     G = nx.gnp_random_graph(10, 0.4, directed=True)
(8)                     kc = k_components(G)
(0)             def test_same():
(4)                 equal = {"A": 2, "B": 2, "C": 2}
(4)                 slightly_different = {"A": 2, "B": 1, "C": 2}
(4)                 different = {"A": 2, "B": 8, "C": 18}
(4)                 assert _same(equal)
(4)                 assert not _same(slightly_different)
(4)                 assert _same(slightly_different, tol=1)
(4)                 assert not _same(different)
(4)                 assert not _same(different, tol=4)
(0)             class TestAntiGraph:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.Gnp = nx.gnp_random_graph(20, 0.8, seed=42)
(8)                     cls.Anp = _AntiGraph(nx.complement(cls.Gnp))
(8)                     cls.Gd = nx.davis_southern_women_graph()
(8)                     cls.Ad = _AntiGraph(nx.complement(cls.Gd))
(8)                     cls.Gk = nx.karate_club_graph()
(8)                     cls.Ak = _AntiGraph(nx.complement(cls.Gk))
(8)                     cls.GA = [(cls.Gnp, cls.Anp), (cls.Gd, cls.Ad), (cls.Gk, cls.Ak)]
(4)                 def test_size(self):
(8)                     for G, A in self.GA:
(12)                        n = G.order()
(12)                        s = len(list(G.edges())) + len(list(A.edges()))
(12)                        assert s == (n * (n - 1)) / 2
(4)                 def test_degree(self):
(8)                     for G, A in self.GA:
(12)                        assert sorted(G.degree()) == sorted(A.degree())
(4)                 def test_core_number(self):
(8)                     for G, A in self.GA:
(12)                        assert nx.core_number(G) == nx.core_number(A)
(4)                 def test_connected_components(self):
(8)                     for G, A in self.GA:
(12)                        gc = [set(c) for c in nx.connected_components(G)]
(12)                        ac = [set(c) for c in nx.connected_components(A)]
(12)                        for comp in ac:
(16)                            assert comp in gc
(4)                 def test_adj(self):
(8)                     for G, A in self.GA:
(12)                        for n, nbrs in G.adj.items():
(16)                            a_adj = sorted((n, sorted(ad)) for n, ad in A.adj.items())
(16)                            g_adj = sorted((n, sorted(ad)) for n, ad in G.adj.items())
(16)                            assert a_adj == g_adj
(4)                 def test_adjacency(self):
(8)                     for G, A in self.GA:
(12)                        a_adj = list(A.adjacency())
(12)                        for n, nbrs in G.adjacency():
(16)                            assert (n, set(nbrs)) in a_adj
(4)                 def test_neighbors(self):
(8)                     for G, A in self.GA:
(12)                        node = list(G.nodes())[0]
(12)                        assert set(G.neighbors(node)) == set(A.neighbors(node))
(4)                 def test_node_not_in_graph(self):
(8)                     for G, A in self.GA:
(12)                        node = "non_existent_node"
(12)                        pytest.raises(nx.NetworkXError, A.neighbors, node)
(12)                        pytest.raises(nx.NetworkXError, G.neighbors, node)
(4)                 def test_degree_thingraph(self):
(8)                     for G, A in self.GA:
(12)                        node = list(G.nodes())[0]
(12)                        nodes = list(G.nodes())[1:4]
(12)                        assert G.degree(node) == A.degree(node)
```

```
226: (12)                        assert sum(d for n, d in G.degree()) == sum(d for n, d in
A.degree())
227: (12)                        assert sum(d for n, d in A.degree()) == sum(
228: (16)                            d for n, d in A.degree(weight="weight")
229: (12)                        )
230: (12)                        assert sum(d for n, d in G.degree(nodes)) == sum(
231: (16)                            d for n, d in A.degree(nodes)
232: (12)                        )
```

File 85 - test_matching.py:

```
1: (0)              import networkx as nx
2: (0)              import networkx.algorithms.approximation as a
3: (0)              def test_min_maximal_matching():
4: (4)                  G = nx.Graph()
5: (4)                  assert len(a.min_maximal_matching(G)) == 0
```

File 86 - test_maxcut.py:

```
1: (0)              import random
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.approximation import maxcut
5: (0)              @pytest.mark.parametrize(
6: (4)                  "f", (nx.approximation.randomized_partitioning,
nx.approximation.one_exchange)
7: (0)              )
8: (0)              @pytest.mark.parametrize("graph_constructor", (nx.DiGraph, nx.MultiGraph))
9: (0)              def test_raises_on_directed_and_multigraphs(f, graph_constructor):
10: (4)                 G = graph_constructor([(0, 1), (1, 2)])
11: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
12: (8)                     f(G)
13: (0)             def _is_valid_cut(G, set1, set2):
14: (4)                 union = set1.union(set2)
15: (4)                 assert union == set(G.nodes)
16: (4)                 assert len(set1) + len(set2) == G.number_of_nodes()
17: (0)             def _cut_is_locally_optimal(G, cut_size, set1):
18: (4)                 for i, node in enumerate(set1):
19: (8)                     cut_size_without_node = nx.algorithms.cut_size(
20: (12)                        G, set1 - {node}, weight="weight"
21: (8)                     )
22: (8)                     assert cut_size_without_node <= cut_size
23: (0)             def test_random_partitioning():
24: (4)                 G = nx.complete_graph(5)
25: (4)                 _, (set1, set2) = maxcut.randomized_partitioning(G, seed=5)
26: (4)                 _is_valid_cut(G, set1, set2)
27: (0)             def test_random_partitioning_all_to_one():
28: (4)                 G = nx.complete_graph(5)
29: (4)                 _, (set1, set2) = maxcut.randomized_partitioning(G, p=1)
30: (4)                 _is_valid_cut(G, set1, set2)
31: (4)                 assert len(set1) == G.number_of_nodes()
32: (4)                 assert len(set2) == 0
33: (0)             def test_one_exchange_basic():
34: (4)                 G = nx.complete_graph(5)
35: (4)                 random.seed(5)
36: (4)                 for u, v, w in G.edges(data=True):
37: (8)                     w["weight"] = random.randrange(-100, 100, 1) / 10
38: (4)                 initial_cut = set(random.sample(sorted(G.nodes()), k=5))
39: (4)                 cut_size, (set1, set2) = maxcut.one_exchange(
40: (8)                     G, initial_cut, weight="weight", seed=5
41: (4)                 )
42: (4)                 _is_valid_cut(G, set1, set2)
43: (4)                 _cut_is_locally_optimal(G, cut_size, set1)
44: (0)             def test_one_exchange_optimal():
45: (4)                 G = nx.Graph()
```

```
46: (4)                 G.add_edge(1, 2, weight=3)
47: (4)                 G.add_edge(1, 3, weight=3)
48: (4)                 G.add_edge(1, 4, weight=3)
49: (4)                 G.add_edge(1, 5, weight=3)
50: (4)                 G.add_edge(2, 3, weight=5)
51: (4)                 cut_size, (set1, set2) = maxcut.one_exchange(G, weight="weight", seed=5)
52: (4)                 _is_valid_cut(G, set1, set2)
53: (4)                 _cut_is_locally_optimal(G, cut_size, set1)
54: (4)                 assert cut_size == 14
55: (0)             def test_negative_weights():
56: (4)                 G = nx.complete_graph(5)
57: (4)                 random.seed(5)
58: (4)                 for u, v, w in G.edges(data=True):
59: (8)                     w["weight"] = -1 * random.random()
60: (4)                 initial_cut = set(random.sample(sorted(G.nodes()), k=5))
61: (4)                 cut_size, (set1, set2) = maxcut.one_exchange(G, initial_cut,
weight="weight")
62: (4)                 _is_valid_cut(G, set1, set2)
63: (4)                 _cut_is_locally_optimal(G, cut_size, set1)
64: (4)                 assert len(set1) == len(G.nodes) or len(set2) == len(G.nodes)

----------------------------------------

File 87 - test_ramsey.py:

1: (0)              import networkx as nx
2: (0)              import networkx.algorithms.approximation as apxa
3: (0)              def test_ramsey():
4: (4)                  graph = nx.complete_graph(10)
5: (4)                  c, i = apxa.ramsey_R2(graph)
6: (4)                  cdens = nx.density(graph.subgraph(c))
7: (4)                  assert cdens == 1.0, "clique not correctly found by ramsey!"
8: (4)                  idens = nx.density(graph.subgraph(i))
9: (4)                  assert idens == 0.0, "i-set not correctly found by ramsey!"
10: (4)                 graph = nx.trivial_graph()
11: (4)                 c, i = apxa.ramsey_R2(graph)
12: (4)                 assert c == {0}, "clique not correctly found by ramsey!"
13: (4)                 assert i == {0}, "i-set not correctly found by ramsey!"
14: (4)                 graph = nx.barbell_graph(10, 5, nx.Graph())
15: (4)                 c, i = apxa.ramsey_R2(graph)
16: (4)                 cdens = nx.density(graph.subgraph(c))
17: (4)                 assert cdens == 1.0, "clique not correctly found by ramsey!"
18: (4)                 idens = nx.density(graph.subgraph(i))
19: (4)                 assert idens == 0.0, "i-set not correctly found by ramsey!"
20: (4)                 graph.add_edges_from([(n, n) for n in range(0, len(graph), 2)])
21: (4)                 cc, ii = apxa.ramsey_R2(graph)
22: (4)                 assert cc == c
23: (4)                 assert ii == i

----------------------------------------

File 88 - test_steinertree.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.approximation.steinertree import metric_closure,
steiner_tree
4: (0)              from networkx.utils import edges_equal
5: (0)              class TestSteinerTree:
6: (4)                  @classmethod
7: (4)                  def setup_class(cls):
8: (8)                      G1 = nx.Graph()
9: (8)                      G1.add_edge(1, 2, weight=10)
10: (8)                     G1.add_edge(2, 3, weight=10)
11: (8)                     G1.add_edge(3, 4, weight=10)
12: (8)                     G1.add_edge(4, 5, weight=10)
13: (8)                     G1.add_edge(5, 6, weight=10)
14: (8)                     G1.add_edge(2, 7, weight=1)
15: (8)                     G1.add_edge(7, 5, weight=1)
```

```
(8)                     G2 = nx.Graph()
(8)                     G2.add_edge(0, 5, weight=6)
(8)                     G2.add_edge(1, 2, weight=2)
(8)                     G2.add_edge(1, 5, weight=3)
(8)                     G2.add_edge(2, 4, weight=4)
(8)                     G2.add_edge(3, 5, weight=5)
(8)                     G2.add_edge(4, 5, weight=1)
(8)                     G3 = nx.Graph()
(8)                     G3.add_edge(1, 2, weight=8)
(8)                     G3.add_edge(1, 9, weight=3)
(8)                     G3.add_edge(1, 8, weight=6)
(8)                     G3.add_edge(1, 10, weight=2)
(8)                     G3.add_edge(1, 14, weight=3)
(8)                     G3.add_edge(2, 3, weight=6)
(8)                     G3.add_edge(3, 4, weight=3)
(8)                     G3.add_edge(3, 10, weight=2)
(8)                     G3.add_edge(3, 11, weight=1)
(8)                     G3.add_edge(4, 5, weight=1)
(8)                     G3.add_edge(4, 11, weight=1)
(8)                     G3.add_edge(5, 6, weight=4)
(8)                     G3.add_edge(5, 11, weight=2)
(8)                     G3.add_edge(5, 12, weight=1)
(8)                     G3.add_edge(5, 13, weight=3)
(8)                     G3.add_edge(6, 7, weight=2)
(8)                     G3.add_edge(6, 12, weight=3)
(8)                     G3.add_edge(6, 13, weight=1)
(8)                     G3.add_edge(7, 8, weight=3)
(8)                     G3.add_edge(7, 9, weight=3)
(8)                     G3.add_edge(7, 11, weight=5)
(8)                     G3.add_edge(7, 13, weight=2)
(8)                     G3.add_edge(7, 14, weight=4)
(8)                     G3.add_edge(8, 9, weight=2)
(8)                     G3.add_edge(9, 14, weight=1)
(8)                     G3.add_edge(10, 11, weight=2)
(8)                     G3.add_edge(10, 14, weight=1)
(8)                     G3.add_edge(11, 12, weight=1)
(8)                     G3.add_edge(11, 14, weight=7)
(8)                     G3.add_edge(12, 14, weight=3)
(8)                     G3.add_edge(12, 15, weight=1)
(8)                     G3.add_edge(13, 14, weight=4)
(8)                     G3.add_edge(13, 15, weight=1)
(8)                     G3.add_edge(14, 15, weight=2)
(8)                     cls.G1 = G1
(8)                     cls.G2 = G2
(8)                     cls.G3 = G3
(8)                     cls.G1_term_nodes = [1, 2, 3, 4, 5]
(8)                     cls.G2_term_nodes = [0, 2, 3]
(8)                     cls.G3_term_nodes = [1, 3, 5, 6, 8, 10, 11, 12, 13]
(8)                     cls.methods = ["kou", "mehlhorn"]
(4)                 def test_connected_metric_closure(self):
(8)                     G = self.G1.copy()
(8)                     G.add_node(100)
(8)                     pytest.raises(nx.NetworkXError, metric_closure, G)
(4)                 def test_metric_closure(self):
(8)                     M = metric_closure(self.G1)
(8)                     mc = [
(12)                        (1, 2, {"distance": 10, "path": [1, 2]}),
(12)                        (1, 3, {"distance": 20, "path": [1, 2, 3]}),
(12)                        (1, 4, {"distance": 22, "path": [1, 2, 7, 5, 4]}),
(12)                        (1, 5, {"distance": 12, "path": [1, 2, 7, 5]}),
(12)                        (1, 6, {"distance": 22, "path": [1, 2, 7, 5, 6]}),
(12)                        (1, 7, {"distance": 11, "path": [1, 2, 7]}),
(12)                        (2, 3, {"distance": 10, "path": [2, 3]}),
(12)                        (2, 4, {"distance": 12, "path": [2, 7, 5, 4]}),
(12)                        (2, 5, {"distance": 2, "path": [2, 7, 5]}),
(12)                        (2, 6, {"distance": 12, "path": [2, 7, 5, 6]}),
(12)                        (2, 7, {"distance": 1, "path": [2, 7]}),
(12)                        (3, 4, {"distance": 10, "path": [3, 4]}),
(12)                        (3, 5, {"distance": 12, "path": [3, 2, 7, 5]}),
```

```
(12)                        (3, 6, {"distance": 22, "path": [3, 2, 7, 5, 6]}),
(12)                        (3, 7, {"distance": 11, "path": [3, 2, 7]}),
(12)                        (4, 5, {"distance": 10, "path": [4, 5]}),
(12)                        (4, 6, {"distance": 20, "path": [4, 5, 6]}),
(12)                        (4, 7, {"distance": 11, "path": [4, 5, 7]}),
(12)                        (5, 6, {"distance": 10, "path": [5, 6]}),
(12)                        (5, 7, {"distance": 1, "path": [5, 7]}),
(12)                        (6, 7, {"distance": 11, "path": [6, 5, 7]}),
(8)                     ]
(8)                     assert edges_equal(list(M.edges(data=True)), mc)
(4)                 def test_steiner_tree(self):
(8)                     valid_steiner_trees = [
(12)                        [
(16)                            [
(20)                                (1, 2, {"weight": 10}),
(20)                                (2, 3, {"weight": 10}),
(20)                                (2, 7, {"weight": 1}),
(20)                                (3, 4, {"weight": 10}),
(20)                                (5, 7, {"weight": 1}),
(16)                            ],
(16)                            [
(20)                                (1, 2, {"weight": 10}),
(20)                                (2, 7, {"weight": 1}),
(20)                                (3, 4, {"weight": 10}),
(20)                                (4, 5, {"weight": 10}),
(20)                                (5, 7, {"weight": 1}),
(16)                            ],
(16)                            [
(20)                                (1, 2, {"weight": 10}),
(20)                                (2, 3, {"weight": 10}),
(20)                                (2, 7, {"weight": 1}),
(20)                                (4, 5, {"weight": 10}),
(20)                                (5, 7, {"weight": 1}),
(16)                            ],
(12)                        ],
(12)                        [
(16)                            [
(20)                                (0, 5, {"weight": 6}),
(20)                                (1, 2, {"weight": 2}),
(20)                                (1, 5, {"weight": 3}),
(20)                                (3, 5, {"weight": 5}),
(16)                            ],
(16)                            [
(20)                                (0, 5, {"weight": 6}),
(20)                                (4, 2, {"weight": 4}),
(20)                                (4, 5, {"weight": 1}),
(20)                                (3, 5, {"weight": 5}),
(16)                            ],
(12)                        ],
(12)                        [
(16)                            [
(20)                                (1, 10, {"weight": 2}),
(20)                                (3, 10, {"weight": 2}),
(20)                                (3, 11, {"weight": 1}),
(20)                                (5, 12, {"weight": 1}),
(20)                                (6, 13, {"weight": 1}),
(20)                                (8, 9, {"weight": 2}),
(20)                                (9, 14, {"weight": 1}),
(20)                                (10, 14, {"weight": 1}),
(20)                                (11, 12, {"weight": 1}),
(20)                                (12, 15, {"weight": 1}),
(20)                                (13, 15, {"weight": 1}),
(16)                            ]
(12)                        ],
(8)                     ]
(8)                     for method in self.methods:
(12)                        for G, term_nodes, valid_trees in zip(
(16)                            [self.G1, self.G2, self.G3],
(16)                            [self.G1_term_nodes, self.G2_term_nodes, self.G3_term_nodes],
```

```
154: (16)                           valid_steiner_trees,
155: (12)                       ):
156: (16)                           S = steiner_tree(G, term_nodes, method=method)
157: (16)                           assert any(
158: (20)                               edges_equal(list(S.edges(data=True)), valid_tree)
159: (20)                               for valid_tree in valid_trees
160: (16)                           )
161: (4)                def test_multigraph_steiner_tree(self):
162: (8)                    G = nx.MultiGraph()
163: (8)                    G.add_edges_from(
164: (12)                       [
165: (16)                           (1, 2, 0, {"weight": 1}),
166: (16)                           (2, 3, 0, {"weight": 999}),
167: (16)                           (2, 3, 1, {"weight": 1}),
168: (16)                           (3, 4, 0, {"weight": 1}),
169: (16)                           (3, 5, 0, {"weight": 1}),
170: (12)                       ]
171: (8)                    )
172: (8)                    terminal_nodes = [2, 4, 5]
173: (8)                    expected_edges = [
174: (12)                       (2, 3, 1, {"weight": 1}),  # edge with key 1 has lower weight
175: (12)                       (3, 4, 0, {"weight": 1}),
176: (12)                       (3, 5, 0, {"weight": 1}),
177: (8)                    ]
178: (8)                    for method in self.methods:
179: (12)                       S = steiner_tree(G, terminal_nodes, method=method)
180: (12)                       assert edges_equal(S.edges(data=True, keys=True), expected_edges)
181: (0)            @pytest.mark.parametrize("method", ("kou", "mehlhorn"))
182: (0)            def test_steiner_tree_weight_attribute(method):
183: (4)                G = nx.star_graph(4)
184: (4)                nx.set_edge_attributes(G, {e: 10 for e in G.edges}, name="distance")
185: (4)                H = nx.approximation.steiner_tree(G, [1, 3], method=method,
weight="distance")
186: (4)                assert nx.utils.edges_equal(H.edges, [(0, 1), (0, 3)])
187: (0)            @pytest.mark.parametrize("method", ("kou", "mehlhorn"))
188: (0)            def test_steiner_tree_multigraph_weight_attribute(method):
189: (4)                G = nx.cycle_graph(3, create_using=nx.MultiGraph)
190: (4)                nx.set_edge_attributes(G, {e: 10 for e in G.edges}, name="distance")
191: (4)                G.add_edge(2, 0, distance=5)
192: (4)                H = nx.approximation.steiner_tree(G, list(G), method=method,
weight="distance")
193: (4)                assert len(H.edges) == 2 and H.has_edge(2, 0, key=1)
194: (4)                assert sum(dist for *_, dist in H.edges(data="distance")) == 15
195: (0)            @pytest.mark.parametrize("method", (None, "mehlhorn", "kou"))
196: (0)            def test_steiner_tree_methods(method):
197: (4)                G = nx.star_graph(4)
198: (4)                expected = nx.Graph([(0, 1), (0, 3)])
199: (4)                st = nx.approximation.steiner_tree(G, [1, 3], method=method)
200: (4)                assert nx.utils.edges_equal(st.edges, expected.edges)
201: (0)            def test_steiner_tree_method_invalid():
202: (4)                G = nx.star_graph(4)
203: (4)                with pytest.raises(
204: (8)                    ValueError, match="invalid_method is not a valid choice for an
algorithm."
205: (4)                ):
206: (8)                    nx.approximation.steiner_tree(G, terminal_nodes=[1, 3],
method="invalid_method")

----------------------------------------

File 89 - test_traveling_salesman.py:

1: (0)              """Unit tests for the traveling_salesman module."""
2: (0)              import random
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              import networkx.algorithms.approximation as nx_app
6: (0)              pairwise = nx.utils.pairwise
7: (0)              def test_christofides_hamiltonian():
```

```
 8: (4)                 random.seed(42)
 9: (4)                 G = nx.complete_graph(20)
10: (4)                 for u, v in G.edges():
11: (8)                     G[u][v]["weight"] = random.randint(0, 10)
12: (4)                 H = nx.Graph()
13: (4)                 H.add_edges_from(pairwise(nx_app.christofides(G)))
14: (4)                 H.remove_edges_from(nx.find_cycle(H))
15: (4)                 assert len(H.edges) == 0
16: (4)                 tree = nx.minimum_spanning_tree(G, weight="weight")
17: (4)                 H = nx.Graph()
18: (4)                 H.add_edges_from(pairwise(nx_app.christofides(G, tree)))
19: (4)                 H.remove_edges_from(nx.find_cycle(H))
20: (4)                 assert len(H.edges) == 0
21: (0)             def test_christofides_incomplete_graph():
22: (4)                 G = nx.complete_graph(10)
23: (4)                 G.remove_edge(0, 1)
24: (4)                 pytest.raises(nx.NetworkXError, nx_app.christofides, G)
25: (0)             def test_christofides_ignore_selfloops():
26: (4)                 G = nx.complete_graph(5)
27: (4)                 G.add_edge(3, 3)
28: (4)                 cycle = nx_app.christofides(G)
29: (4)                 assert len(cycle) - 1 == len(G) == len(set(cycle))
30: (0)             class TestBase:
31: (4)                 @classmethod
32: (4)                 def setup_class(cls):
33: (8)                     cls.DG = nx.DiGraph()
34: (8)                     cls.DG.add_weighted_edges_from(
35: (12)                        {
36: (16)                            ("A", "B", 3),
37: (16)                            ("A", "C", 17),
38: (16)                            ("A", "D", 14),
39: (16)                            ("B", "A", 3),
40: (16)                            ("B", "C", 12),
41: (16)                            ("B", "D", 16),
42: (16)                            ("C", "A", 13),
43: (16)                            ("C", "B", 12),
44: (16)                            ("C", "D", 4),
45: (16)                            ("D", "A", 14),
46: (16)                            ("D", "B", 15),
47: (16)                            ("D", "C", 2),
48: (12)                        }
49: (8)                     )
50: (8)                     cls.DG_cycle = ["D", "C", "B", "A", "D"]
51: (8)                     cls.DG_cost = 31.0
52: (8)                     cls.DG2 = nx.DiGraph()
53: (8)                     cls.DG2.add_weighted_edges_from(
54: (12)                        {
55: (16)                            ("A", "B", 3),
56: (16)                            ("A", "C", 17),
57: (16)                            ("A", "D", 14),
58: (16)                            ("B", "A", 30),
59: (16)                            ("B", "C", 2),
60: (16)                            ("B", "D", 16),
61: (16)                            ("C", "A", 33),
62: (16)                            ("C", "B", 32),
63: (16)                            ("C", "D", 34),
64: (16)                            ("D", "A", 14),
65: (16)                            ("D", "B", 15),
66: (16)                            ("D", "C", 2),
67: (12)                        }
68: (8)                     )
69: (8)                     cls.DG2_cycle = ["D", "A", "B", "C", "D"]
70: (8)                     cls.DG2_cost = 53.0
71: (8)                     cls.unweightedUG = nx.complete_graph(5, nx.Graph())
72: (8)                     cls.unweightedDG = nx.complete_graph(5, nx.DiGraph())
73: (8)                     cls.incompleteUG = nx.Graph()
74: (8)                     cls.incompleteUG.add_weighted_edges_from({(0, 1, 1), (1, 2, 3)})
75: (8)                     cls.incompleteDG = nx.DiGraph()
76: (8)                     cls.incompleteDG.add_weighted_edges_from({(0, 1, 1), (1, 2, 3)})
```

```
77: (8)                     cls.UG = nx.Graph()
78: (8)                     cls.UG.add_weighted_edges_from(
79: (12)                        {
80: (16)                            ("A", "B", 3),
81: (16)                            ("A", "C", 17),
82: (16)                            ("A", "D", 14),
83: (16)                            ("B", "C", 12),
84: (16)                            ("B", "D", 16),
85: (16)                            ("C", "D", 4),
86: (12)                        }
87: (8)                     )
88: (8)                     cls.UG_cycle = ["D", "C", "B", "A", "D"]
89: (8)                     cls.UG_cost = 33.0
90: (8)                     cls.UG2 = nx.Graph()
91: (8)                     cls.UG2.add_weighted_edges_from(
92: (12)                        {
93: (16)                            ("A", "B", 1),
94: (16)                            ("A", "C", 15),
95: (16)                            ("A", "D", 5),
96: (16)                            ("B", "C", 16),
97: (16)                            ("B", "D", 8),
98: (16)                            ("C", "D", 3),
99: (12)                        }
100: (8)                    )
101: (8)                    cls.UG2_cycle = ["D", "C", "B", "A", "D"]
102: (8)                    cls.UG2_cost = 25.0
103: (0)            def validate_solution(soln, cost, exp_soln, exp_cost):
104: (4)                assert soln == exp_soln
105: (4)                assert cost == exp_cost
106: (0)            def validate_symmetric_solution(soln, cost, exp_soln, exp_cost):
107: (4)                assert soln == exp_soln or soln == exp_soln[::-1]
108: (4)                assert cost == exp_cost
109: (0)            class TestGreedyTSP(TestBase):
110: (4)                def test_greedy(self):
111: (8)                    cycle = nx_app.greedy_tsp(self.DG, source="D")
112: (8)                    cost = sum(self.DG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
113: (8)                    validate_solution(cycle, cost, ["D", "C", "B", "A", "D"], 31.0)
114: (8)                    cycle = nx_app.greedy_tsp(self.DG2, source="D")
115: (8)                    cost = sum(self.DG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
116: (8)                    validate_solution(cycle, cost, ["D", "C", "B", "A", "D"], 78.0)
117: (8)                    cycle = nx_app.greedy_tsp(self.UG, source="D")
118: (8)                    cost = sum(self.UG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
119: (8)                    validate_solution(cycle, cost, ["D", "C", "B", "A", "D"], 33.0)
120: (8)                    cycle = nx_app.greedy_tsp(self.UG2, source="D")
121: (8)                    cost = sum(self.UG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
122: (8)                    validate_solution(cycle, cost, ["D", "C", "A", "B", "D"], 27.0)
123: (4)                def test_not_complete_graph(self):
124: (8)                    pytest.raises(nx.NetworkXError, nx_app.greedy_tsp, self.incompleteUG)
125: (8)                    pytest.raises(nx.NetworkXError, nx_app.greedy_tsp, self.incompleteDG)
126: (4)                def test_not_weighted_graph(self):
127: (8)                    nx_app.greedy_tsp(self.unweightedUG)
128: (8)                    nx_app.greedy_tsp(self.unweightedDG)
129: (4)                def test_two_nodes(self):
130: (8)                    G = nx.Graph()
131: (8)                    G.add_weighted_edges_from({(1, 2, 1)})
132: (8)                    cycle = nx_app.greedy_tsp(G)
133: (8)                    cost = sum(G[n][nbr]["weight"] for n, nbr in pairwise(cycle))
134: (8)                    validate_solution(cycle, cost, [1, 2, 1], 2)
135: (4)                def test_ignore_selfloops(self):
136: (8)                    G = nx.complete_graph(5)
137: (8)                    G.add_edge(3, 3)
138: (8)                    cycle = nx_app.greedy_tsp(G)
139: (8)                    assert len(cycle) - 1 == len(G) == len(set(cycle))
140: (0)            class TestSimulatedAnnealingTSP(TestBase):
141: (4)                tsp = staticmethod(nx_app.simulated_annealing_tsp)
142: (4)                def test_simulated_annealing_directed(self):
143: (8)                    cycle = self.tsp(self.DG, "greedy", source="D", seed=42)
144: (8)                    cost = sum(self.DG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
145: (8)                    validate_solution(cycle, cost, self.DG_cycle, self.DG_cost)
```

```
146: (8)                    initial_sol = ["D", "B", "A", "C", "D"]
147: (8)                    cycle = self.tsp(self.DG, initial_sol, source="D", seed=42)
148: (8)                    cost = sum(self.DG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
149: (8)                    validate_solution(cycle, cost, self.DG_cycle, self.DG_cost)
150: (8)                    initial_sol = ["D", "A", "C", "B", "D"]
151: (8)                    cycle = self.tsp(self.DG, initial_sol, move="1-0", source="D",
seed=42)
152: (8)                    cost = sum(self.DG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
153: (8)                    validate_solution(cycle, cost, self.DG_cycle, self.DG_cost)
154: (8)                    cycle = self.tsp(self.DG2, "greedy", source="D", seed=42)
155: (8)                    cost = sum(self.DG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
156: (8)                    validate_solution(cycle, cost, self.DG2_cycle, self.DG2_cost)
157: (8)                    cycle = self.tsp(self.DG2, "greedy", move="1-0", source="D", seed=42)
158: (8)                    cost = sum(self.DG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
159: (8)                    validate_solution(cycle, cost, self.DG2_cycle, self.DG2_cost)
160: (4)                def test_simulated_annealing_undirected(self):
161: (8)                    cycle = self.tsp(self.UG, "greedy", source="D", seed=42)
162: (8)                    cost = sum(self.UG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
163: (8)                    validate_solution(cycle, cost, self.UG_cycle, self.UG_cost)
164: (8)                    cycle = self.tsp(self.UG2, "greedy", source="D", seed=42)
165: (8)                    cost = sum(self.UG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
166: (8)                    validate_symmetric_solution(cycle, cost, self.UG2_cycle,
self.UG2_cost)
167: (8)                    cycle = self.tsp(self.UG2, "greedy", move="1-0", source="D", seed=42)
168: (8)                    cost = sum(self.UG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
169: (8)                    validate_symmetric_solution(cycle, cost, self.UG2_cycle,
self.UG2_cost)
170: (4)                def test_error_on_input_order_mistake(self):
171: (8)                    pytest.raises(TypeError, self.tsp, self.UG, weight="weight")
172: (8)                    pytest.raises(nx.NetworkXError, self.tsp, self.UG, "weight")
173: (4)                def test_not_complete_graph(self):
174: (8)                    pytest.raises(nx.NetworkXError, self.tsp, self.incompleteUG, "greedy",
source=0)
175: (8)                    pytest.raises(nx.NetworkXError, self.tsp, self.incompleteDG, "greedy",
source=0)
176: (4)                def test_ignore_selfloops(self):
177: (8)                    G = nx.complete_graph(5)
178: (8)                    G.add_edge(3, 3)
179: (8)                    cycle = self.tsp(G, "greedy")
180: (8)                    assert len(cycle) - 1 == len(G) == len(set(cycle))
181: (4)                def test_not_weighted_graph(self):
182: (8)                    self.tsp(self.unweightedUG, "greedy")
183: (8)                    self.tsp(self.unweightedDG, "greedy")
184: (4)                def test_two_nodes(self):
185: (8)                    G = nx.Graph()
186: (8)                    G.add_weighted_edges_from({(1, 2, 1)})
187: (8)                    cycle = self.tsp(G, "greedy", source=1, seed=42)
188: (8)                    cost = sum(G[n][nbr]["weight"] for n, nbr in pairwise(cycle))
189: (8)                    validate_solution(cycle, cost, [1, 2, 1], 2)
190: (8)                    cycle = self.tsp(G, [1, 2, 1], source=1, seed=42)
191: (8)                    cost = sum(G[n][nbr]["weight"] for n, nbr in pairwise(cycle))
192: (8)                    validate_solution(cycle, cost, [1, 2, 1], 2)
193: (4)                def test_failure_of_costs_too_high_when_iterations_low(self):
194: (8)                    cycle = self.tsp(
195: (12)                       self.DG2, "greedy", source="D", move="1-0", alpha=1, N_inner=1,
seed=42
196: (8)                    )
197: (8)                    cost = sum(self.DG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
198: (8)                    print(cycle, cost)
199: (8)                    assert cost > self.DG2_cost
200: (8)                    initial_sol = ["D", "A", "B", "C", "D"]
201: (8)                    cycle = self.tsp(
202: (12)                       self.DG,
203: (12)                       initial_sol,
204: (12)                       source="D",
205: (12)                       move="1-0",
206: (12)                       alpha=0.1,
207: (12)                       N_inner=1,
208: (12)                       max_iterations=1,
```

```
209: (12)                        seed=42,
210: (8)                    )
211: (8)                    cost = sum(self.DG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
212: (8)                    print(cycle, cost)
213: (8)                    assert cost > self.DG_cost
214: (0)            class TestThresholdAcceptingTSP(TestSimulatedAnnealingTSP):
215: (4)                tsp = staticmethod(nx_app.threshold_accepting_tsp)
216: (4)                def test_failure_of_costs_too_high_when_iterations_low(self):
217: (8)                    cycle = self.tsp(
218: (12)                       self.DG2,
219: (12)                       "greedy",
220: (12)                       source="D",
221: (12)                       move="1-0",
222: (12)                       N_inner=1,
223: (12)                       max_iterations=1,
224: (12)                       seed=4,
225: (8)                    )
226: (8)                    cost = sum(self.DG2[n][nbr]["weight"] for n, nbr in pairwise(cycle))
227: (8)                    assert cost > self.DG2_cost
228: (8)                    initial_sol = ["D", "A", "B", "C", "D"]
229: (8)                    cycle = self.tsp(
230: (12)                       self.DG, initial_sol, source="D", move="1-0", threshold=-3,
seed=42
231: (8)                    )
232: (8)                    cost = sum(self.DG[n][nbr]["weight"] for n, nbr in pairwise(cycle))
233: (8)                    assert cost > self.DG_cost
234: (0)            def test_TSP_method():
235: (4)                G = nx.cycle_graph(9)
236: (4)                G[4][5]["weight"] = 10
237: (4)                sa_tsp = lambda G, weight: nx_app.simulated_annealing_tsp(
238: (8)                    G, "greedy", weight, source=4, seed=1
239: (4)                )
240: (4)                path = nx_app.traveling_salesman_problem(
241: (8)                    G,
242: (8)                    method=sa_tsp,
243: (8)                    cycle=False,
244: (4)                )
245: (4)                print(path)
246: (4)                assert path == [4, 3, 2, 1, 0, 8, 7, 6, 5]
247: (0)            def test_TSP_unweighted():
248: (4)                G = nx.cycle_graph(9)
249: (4)                path = nx_app.traveling_salesman_problem(G, nodes=[3, 6], cycle=False)
250: (4)                assert path in ([3, 4, 5, 6], [6, 5, 4, 3])
251: (4)                cycle = nx_app.traveling_salesman_problem(G, nodes=[3, 6])
252: (4)                assert cycle in ([3, 4, 5, 6, 5, 4, 3], [6, 5, 4, 3, 4, 5, 6])
253: (0)            def test_TSP_weighted():
254: (4)                G = nx.cycle_graph(9)
255: (4)                G[0][1]["weight"] = 2
256: (4)                G[1][2]["weight"] = 2
257: (4)                G[2][3]["weight"] = 2
258: (4)                G[3][4]["weight"] = 4
259: (4)                G[4][5]["weight"] = 5
260: (4)                G[5][6]["weight"] = 4
261: (4)                G[6][7]["weight"] = 2
262: (4)                G[7][8]["weight"] = 2
263: (4)                G[8][0]["weight"] = 2
264: (4)                tsp = nx_app.traveling_salesman_problem
265: (4)                expected_paths = ([3, 2, 1, 0, 8, 7, 6], [6, 7, 8, 0, 1, 2, 3])
266: (4)                expected_cycles = (
267: (8)                    [3, 2, 1, 0, 8, 7, 6, 7, 8, 0, 1, 2, 3],
268: (8)                    [6, 7, 8, 0, 1, 2, 3, 2, 1, 0, 8, 7, 6],
269: (4)                )
270: (4)                expected_tourpaths = ([5, 6, 7, 8, 0, 1, 2, 3, 4], [4, 3, 2, 1, 0, 8, 7,
6, 5])
271: (4)                cycle = tsp(G, nodes=[3, 6], weight="weight")
272: (4)                assert cycle in expected_cycles
273: (4)                path = tsp(G, nodes=[3, 6], weight="weight", cycle=False)
274: (4)                assert path in expected_paths
275: (4)                tourpath = tsp(G, weight="weight", cycle=False)
```

```
(4)                 assert tourpath in expected_tourpaths
(4)                 methods = [
(8)                     (nx_app.christofides, {}),
(8)                     (nx_app.greedy_tsp, {}),
(8)                     (
(12)                        nx_app.simulated_annealing_tsp,
(12)                        {"init_cycle": "greedy"},
(8)                     ),
(8)                     (
(12)                        nx_app.threshold_accepting_tsp,
(12)                        {"init_cycle": "greedy"},
(8)                     ),
(4)                 ]
(4)                 for method, kwargs in methods:
(8)                     cycle = tsp(G, nodes=[3, 6], weight="weight", method=method, **kwargs)
(8)                     assert cycle in expected_cycles
(8)                     path = tsp(
(12)                        G, nodes=[3, 6], weight="weight", method=method, cycle=False, **kwargs
(8)                     )
(8)                     assert path in expected_paths
(8)                     tourpath = tsp(G, weight="weight", method=method, cycle=False, **kwargs)
(8)                     assert tourpath in expected_tourpaths
(0)             def test_TSP_incomplete_graph_short_path():
(4)                 G = nx.cycle_graph(9)
(4)                 G.add_edges_from([(4, 9), (9, 10), (10, 11), (11, 0)])
(4)                 G[4][5]["weight"] = 5
(4)                 cycle = nx_app.traveling_salesman_problem(G)
(4)                 print(cycle)
(4)                 assert len(cycle) == 17 and len(set(cycle)) == 12
(4)                 path = nx_app.traveling_salesman_problem(G, cycle=False)
(4)                 print(path)
(4)                 assert len(path) == 13 and len(set(path)) == 12
(0)             def test_held_karp_ascent():
(4)                 """
(4)                 Test the Held-Karp relaxation with the ascent method
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
(8)                     [
(12)                        [0, 97, 60, 73, 17, 52],
(12)                        [97, 0, 41, 52, 90, 30],
(12)                        [60, 41, 0, 21, 35, 41],
(12)                        [73, 52, 21, 0, 95, 46],
(12)                        [17, 90, 35, 95, 0, 81],
(12)                        [52, 30, 41, 46, 81, 0],
(8)                     ]
(4)                 )
(4)                 solution_edges = [(1, 3), (2, 4), (3, 2), (4, 0), (5, 1), (0, 5)]
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 opt_hk, z_star = tsp.held_karp_ascent(G)
(4)                 assert round(opt_hk, 2) == 207.00
(4)                 solution = nx.DiGraph()
(4)                 solution.add_edges_from(solution_edges)
(4)                 assert nx.utils.edges_equal(z_star.edges, solution.edges)
(0)             def test_ascent_fractional_solution():
(4)                 """
(4)                 Test the ascent method using a modified version of Figure 2 on page 1140
(4)                 in 'The Traveling Salesman Problem and Minimum Spanning Trees' by Held and
(4)                 Karp
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
(8)                     [
```

```
(12)                        [0, 100, 100, 100000, 100000, 1],
(12)                        [100, 0, 100, 100000, 1, 100000],
(12)                        [100, 100, 0, 1, 100000, 100000],
(12)                        [100000, 100000, 1, 0, 100, 100],
(12)                        [100000, 1, 100000, 100, 0, 100],
(12)                        [1, 100000, 100000, 100, 100, 0],
(8)                     ]
(4)                 )
(4)                 solution_z_star = {
(8)                     (0, 1): 5 / 12,
(8)                     (0, 2): 5 / 12,
(8)                     (0, 5): 5 / 6,
(8)                     (1, 0): 5 / 12,
(8)                     (1, 2): 1 / 3,
(8)                     (1, 4): 5 / 6,
(8)                     (2, 0): 5 / 12,
(8)                     (2, 1): 1 / 3,
(8)                     (2, 3): 5 / 6,
(8)                     (3, 2): 5 / 6,
(8)                     (3, 4): 1 / 3,
(8)                     (3, 5): 1 / 2,
(8)                     (4, 1): 5 / 6,
(8)                     (4, 3): 1 / 3,
(8)                     (4, 5): 1 / 2,
(8)                     (5, 0): 5 / 6,
(8)                     (5, 3): 1 / 2,
(8)                     (5, 4): 1 / 2,
(4)                 }
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 opt_hk, z_star = tsp.held_karp_ascent(G)
(4)                 assert round(opt_hk, 2) == 303.00
(4)                 assert {key: round(z_star[key], 4) for key in z_star} == {
(8)                     key: round(solution_z_star[key], 4) for key in solution_z_star
(4)                 }
(0)             def test_ascent_method_asymmetric():
(4)                 """
(4)                 Tests the ascent method using a truly asymmetric graph for which the
(4)                 solution has been brute forced
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
(8)                     [
(12)                        [0, 26, 63, 59, 69, 31, 41],
(12)                        [62, 0, 91, 53, 75, 87, 47],
(12)                        [47, 82, 0, 90, 15, 9, 18],
(12)                        [68, 19, 5, 0, 58, 34, 93],
(12)                        [11, 58, 53, 55, 0, 61, 79],
(12)                        [88, 75, 13, 76, 98, 0, 40],
(12)                        [41, 61, 55, 88, 46, 45, 0],
(8)                     ]
(4)                 )
(4)                 solution_edges = [(0, 1), (1, 3), (3, 2), (2, 5), (5, 6), (4, 0), (6, 4)]
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 opt_hk, z_star = tsp.held_karp_ascent(G)
(4)                 assert round(opt_hk, 2) == 190.00
(4)                 solution = nx.DiGraph()
(4)                 solution.add_edges_from(solution_edges)
(4)                 assert nx.utils.edges_equal(z_star.edges, solution.edges)
(0)             def test_ascent_method_asymmetric_2():
(4)                 """
(4)                 Tests the ascent method using a truly asymmetric graph for which the
(4)                 solution has been brute forced
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
```

```
(8)                     [
(12)                        [0, 45, 39, 92, 29, 31],
(12)                        [72, 0, 4, 12, 21, 60],
(12)                        [81, 6, 0, 98, 70, 53],
(12)                        [49, 71, 59, 0, 98, 94],
(12)                        [74, 95, 24, 43, 0, 47],
(12)                        [56, 43, 3, 65, 22, 0],
(8)                     ]
(4)                 )
(4)                 solution_edges = [(0, 5), (5, 4), (1, 3), (3, 0), (2, 1), (4, 2)]
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 opt_hk, z_star = tsp.held_karp_ascent(G)
(4)                 assert round(opt_hk, 2) == 144.00
(4)                 solution = nx.DiGraph()
(4)                 solution.add_edges_from(solution_edges)
(4)                 assert nx.utils.edges_equal(z_star.edges, solution.edges)
(0)             def test_held_karp_ascent_asymmetric_3():
(4)                 """
(4)                 Tests the ascent method using a truly asymmetric graph with a fractional
(4)                 solution for which the solution has been brute forced.
(4)                 In this graph their are two different optimal, integral solutions (which
(4)                 are also the overall atsp solutions) to the Held Karp relaxation. However,
(4)                 this particular graph has two different tours of optimal value and the
(4)                 possible solutions in the held_karp_ascent function are not stored in an
(4)                 ordered data structure.
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
(8)                     [
(12)                        [0, 1, 5, 2, 7, 4],
(12)                        [7, 0, 7, 7, 1, 4],
(12)                        [4, 7, 0, 9, 2, 1],
(12)                        [7, 2, 7, 0, 4, 4],
(12)                        [5, 5, 4, 4, 0, 3],
(12)                        [3, 9, 1, 3, 4, 0],
(8)                     ]
(4)                 )
(4)                 solution1_edges = [(0, 3), (1, 4), (2, 5), (3, 1), (4, 2), (5, 0)]
(4)                 solution2_edges = [(0, 3), (3, 1), (1, 4), (4, 5), (2, 0), (5, 2)]
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 opt_hk, z_star = tsp.held_karp_ascent(G)
(4)                 assert round(opt_hk, 2) == 13.00
(4)                 solution1 = nx.DiGraph()
(4)                 solution1.add_edges_from(solution1_edges)
(4)                 solution2 = nx.DiGraph()
(4)                 solution2.add_edges_from(solution2_edges)
(4)                 assert nx.utils.edges_equal(z_star.edges, solution1.edges) or
nx.utils.edges_equal(
(8)                     z_star.edges, solution2.edges
(4)                 )
(0)             def test_held_karp_ascent_fractional_asymmetric():
(4)                 """
(4)                 Tests the ascent method using a truly asymmetric graph with a fractional
(4)                 solution for which the solution has been brute forced
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
(8)                     [
(12)                        [0, 100, 150, 100000, 100000, 1],
(12)                        [150, 0, 100, 100000, 1, 100000],
(12)                        [100, 150, 0, 1, 100000, 100000],
(12)                        [100000, 100000, 1, 0, 150, 100],
(12)                        [100000, 2, 100000, 100, 0, 150],
(12)                        [2, 100000, 100000, 150, 100, 0],
(8)                     ]
```

```
(4)                 )
(4)                 solution_z_star = {
(8)                     (0, 1): 5 / 12,
(8)                     (0, 2): 5 / 12,
(8)                     (0, 5): 5 / 6,
(8)                     (1, 0): 5 / 12,
(8)                     (1, 2): 5 / 12,
(8)                     (1, 4): 5 / 6,
(8)                     (2, 0): 5 / 12,
(8)                     (2, 1): 5 / 12,
(8)                     (2, 3): 5 / 6,
(8)                     (3, 2): 5 / 6,
(8)                     (3, 4): 5 / 12,
(8)                     (3, 5): 5 / 12,
(8)                     (4, 1): 5 / 6,
(8)                     (4, 3): 5 / 12,
(8)                     (4, 5): 5 / 12,
(8)                     (5, 0): 5 / 6,
(8)                     (5, 3): 5 / 12,
(8)                     (5, 4): 5 / 12,
(4)                 }
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 opt_hk, z_star = tsp.held_karp_ascent(G)
(4)                 assert round(opt_hk, 2) == 304.00
(4)                 assert {key: round(z_star[key], 4) for key in z_star} == {
(8)                     key: round(solution_z_star[key], 4) for key in solution_z_star
(4)                 }
(0)             def test_spanning_tree_distribution():
(4)                 """
(4)                 Test that we can create an exponential distribution of spanning trees such
(4)                 that the probability of each tree is proportional to the product of edge
(4)                 weights.
(4)                 Results of this test have been confirmed with hypothesis testing from the
(4)                 created distribution.
(4)                 This test uses the symmetric, fractional Held Karp solution.
(4)                 """
(4)                 import networkx.algorithms.approximation.traveling_salesman as tsp
(4)                 pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 z_star = {
(8)                     (0, 1): 5 / 12,
(8)                     (0, 2): 5 / 12,
(8)                     (0, 5): 5 / 6,
(8)                     (1, 0): 5 / 12,
(8)                     (1, 2): 1 / 3,
(8)                     (1, 4): 5 / 6,
(8)                     (2, 0): 5 / 12,
(8)                     (2, 1): 1 / 3,
(8)                     (2, 3): 5 / 6,
(8)                     (3, 2): 5 / 6,
(8)                     (3, 4): 1 / 3,
(8)                     (3, 5): 1 / 2,
(8)                     (4, 1): 5 / 6,
(8)                     (4, 3): 1 / 3,
(8)                     (4, 5): 1 / 2,
(8)                     (5, 0): 5 / 6,
(8)                     (5, 3): 1 / 2,
(8)                     (5, 4): 1 / 2,
(4)                 }
(4)                 solution_gamma = {
(8)                     (0, 1): -0.6383,
(8)                     (0, 2): -0.6827,
(8)                     (0, 5): 0,
(8)                     (1, 2): -1.0781,
(8)                     (1, 4): 0,
(8)                     (2, 3): 0,
(8)                     (5, 3): -0.2820,
(8)                     (5, 4): -0.3327,
(8)                     (4, 3): -0.9927,
```

```
549: (4)                }
550: (4)                G = nx.MultiGraph()
551: (4)                for u, v in z_star:
552: (8)                    if (u, v) in G.edges or (v, u) in G.edges:
553: (12)                       continue
554: (8)                    G.add_edge(u, v)
555: (4)                gamma = tsp.spanning_tree_distribution(G, z_star)
556: (4)                assert {key: round(gamma[key], 4) for key in gamma} == solution_gamma
557: (0)            def test_asadpour_tsp():
558: (4)                """
559: (4)                Test the complete asadpour tsp algorithm with the fractional, symmetric
560: (4)                Held Karp solution. This test also uses an incomplete graph as input.
561: (4)                """
562: (4)                pytest.importorskip("numpy")
563: (4)                pytest.importorskip("scipy")
564: (4)                edge_list = [
565: (8)                    (0, 1, 100),
566: (8)                    (0, 2, 100),
567: (8)                    (0, 5, 1),
568: (8)                    (1, 2, 100),
569: (8)                    (1, 4, 1),
570: (8)                    (2, 3, 1),
571: (8)                    (3, 4, 100),
572: (8)                    (3, 5, 100),
573: (8)                    (4, 5, 100),
574: (8)                    (1, 0, 100),
575: (8)                    (2, 0, 100),
576: (8)                    (5, 0, 1),
577: (8)                    (2, 1, 100),
578: (8)                    (4, 1, 1),
579: (8)                    (3, 2, 1),
580: (8)                    (4, 3, 100),
581: (8)                    (5, 3, 100),
582: (8)                    (5, 4, 100),
583: (4)                ]
584: (4)                G = nx.DiGraph()
585: (4)                G.add_weighted_edges_from(edge_list)
586: (4)                tour = nx_app.traveling_salesman_problem(
587: (8)                    G, weight="weight", method=nx_app.asadpour_atsp, seed=19
588: (4)                )
589: (4)                expected_tours = [
590: (8)                    [1, 4, 5, 0, 2, 3, 2, 1],
591: (8)                    [3, 2, 0, 1, 4, 5, 3],
592: (8)                    [3, 2, 1, 0, 5, 4, 3],
593: (4)                ]
594: (4)                assert tour in expected_tours
595: (0)            def test_asadpour_real_world():
596: (4)                """
597: (4)                This test uses airline prices between the six largest cities in the US.
598: (8)                    * New York City -> JFK
599: (8)                    * Los Angeles -> LAX
600: (8)                    * Chicago -> ORD
601: (8)                    * Houston -> IAH
602: (8)                    * Phoenix -> PHX
603: (8)                    * Philadelphia -> PHL
604: (4)                Flight prices from August 2021 using Delta or American airlines to get
605: (4)                nonstop flight. The brute force solution found the optimal tour to cost
$872
606: (4)                This test also uses the `source` keyword argument to ensure that the tour
607: (4)                always starts at city 0.
608: (4)                """
609: (4)                np = pytest.importorskip("numpy")
610: (4)                pytest.importorskip("scipy")
611: (4)                G_array = np.array(
612: (8)                    [
613: (12)                       [0, 243, 199, 208, 169, 183],  # JFK
614: (12)                       [277, 0, 217, 123, 127, 252],  # LAX
615: (12)                       [297, 197, 0, 197, 123, 177],  # ORD
616: (12)                       [303, 169, 197, 0, 117, 117],  # IAH
```

```
(12)                        [257, 127, 160, 117, 0, 319],  # PHX
(12)                        [183, 332, 217, 117, 319, 0],  # PHL
(8)                     ]
(4)                 )
(4)                 node_map = {0: "JFK", 1: "LAX", 2: "ORD", 3: "IAH", 4: "PHX", 5: "PHL"}
(4)                 expected_tours = [
(8)                     ["JFK", "LAX", "PHX", "ORD", "IAH", "PHL", "JFK"],
(8)                     ["JFK", "ORD", "PHX", "LAX", "IAH", "PHL", "JFK"],
(4)                 ]
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 nx.relabel_nodes(G, node_map, copy=False)
(4)                 tour = nx_app.traveling_salesman_problem(
(8)                     G, weight="weight", method=nx_app.asadpour_atsp, seed=37, source="JFK"
(4)                 )
(4)                 assert tour in expected_tours
(0)             def test_asadpour_real_world_path():
(4)                 """
(4)                 This test uses airline prices between the six largest cities in the US.
This
(4)                 time using a path, not a cycle.
(8)                     * New York City -> JFK
(8)                     * Los Angeles -> LAX
(8)                     * Chicago -> ORD
(8)                     * Houston -> IAH
(8)                     * Phoenix -> PHX
(8)                     * Philadelphia -> PHL
(4)                 Flight prices from August 2021 using Delta or American airlines to get
(4)                 nonstop flight. The brute force solution found the optimal tour to cost
$872
(4)                 """
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 G_array = np.array(
(8)                     [
(12)                        [0, 243, 199, 208, 169, 183],  # JFK
(12)                        [277, 0, 217, 123, 127, 252],  # LAX
(12)                        [297, 197, 0, 197, 123, 177],  # ORD
(12)                        [303, 169, 197, 0, 117, 117],  # IAH
(12)                        [257, 127, 160, 117, 0, 319],  # PHX
(12)                        [183, 332, 217, 117, 319, 0],  # PHL
(8)                     ]
(4)                 )
(4)                 node_map = {0: "JFK", 1: "LAX", 2: "ORD", 3: "IAH", 4: "PHX", 5: "PHL"}
(4)                 expected_paths = [
(8)                     ["ORD", "PHX", "LAX", "IAH", "PHL", "JFK"],
(8)                     ["JFK", "PHL", "IAH", "ORD", "PHX", "LAX"],
(4)                 ]
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 nx.relabel_nodes(G, node_map, copy=False)
(4)                 path = nx_app.traveling_salesman_problem(
(8)                     G, weight="weight", cycle=False, method=nx_app.asadpour_atsp, seed=56
(4)                 )
(4)                 assert path in expected_paths
(0)             def test_asadpour_disconnected_graph():
(4)                 """
(4)                 Test that the proper exception is raised when asadpour_atsp is given an
(4)                 disconnected graph.
(4)                 """
(4)                 G = nx.complete_graph(4, create_using=nx.DiGraph)
(4)                 nx.set_edge_attributes(G, 1, "weight")
(4)                 G.add_node(5)
(4)                 pytest.raises(nx.NetworkXError, nx_app.asadpour_atsp, G)
(0)             def test_asadpour_incomplete_graph():
(4)                 """
(4)                 Test that the proper exception is raised when asadpour_atsp is given an
(4)                 incomplete graph
(4)                 """
(4)                 G = nx.complete_graph(4, create_using=nx.DiGraph)
(4)                 nx.set_edge_attributes(G, 1, "weight")
```

```
(4)                 G.remove_edge(0, 1)
(4)                 pytest.raises(nx.NetworkXError, nx_app.asadpour_atsp, G)
(0)             def test_asadpour_empty_graph():
(4)                 """
(4)                 Test the asadpour_atsp function with an empty graph
(4)                 """
(4)                 G = nx.DiGraph()
(4)                 pytest.raises(nx.NetworkXError, nx_app.asadpour_atsp, G)
(0)             @pytest.mark.slow
(0)             def test_asadpour_integral_held_karp():
(4)                 """
(4)                 This test uses an integral held karp solution and the held karp function
(4)                 will return a graph rather than a dict, bypassing most of the asadpour
(4)                 algorithm.
(4)                 At first glance, this test probably doesn't look like it ensures that we
(4)                 skip the rest of the asadpour algorithm, but it does. We are not fixing a
(4)                 see for the random number generator, so if we sample any spanning trees
(4)                 the approximation would be different basically every time this test is
(4)                 executed but it is not since held karp is deterministic and we do not
(4)                 reach the portion of the code with the dependence on random numbers.
(4)                 """
(4)                 np = pytest.importorskip("numpy")
(4)                 G_array = np.array(
(8)                     [
(12)                        [0, 26, 63, 59, 69, 31, 41],
(12)                        [62, 0, 91, 53, 75, 87, 47],
(12)                        [47, 82, 0, 90, 15, 9, 18],
(12)                        [68, 19, 5, 0, 58, 34, 93],
(12)                        [11, 58, 53, 55, 0, 61, 79],
(12)                        [88, 75, 13, 76, 98, 0, 40],
(12)                        [41, 61, 55, 88, 46, 45, 0],
(8)                     ]
(4)                 )
(4)                 G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)                 for _ in range(2):
(8)                     tour = nx_app.traveling_salesman_problem(G,
method=nx_app.asadpour_atsp)
(8)                     assert [1, 3, 2, 5, 2, 6, 4, 0, 1] == tour
(0)             def test_directed_tsp_impossible():
(4)                 """
(4)                 Test the asadpour algorithm with a graph without a hamiltonian circuit
(4)                 """
(4)                 pytest.importorskip("numpy")
(4)                 edges = [
(8)                     (0, 1, 10),
(8)                     (0, 2, 11),
(8)                     (0, 3, 12),
(8)                     (1, 2, 4),
(8)                     (1, 3, 6),
(8)                     (2, 1, 3),
(8)                     (2, 3, 2),
(8)                     (3, 1, 5),
(8)                     (3, 2, 1),
(4)                 ]
(4)                 G = nx.DiGraph()
(4)                 G.add_weighted_edges_from(edges)
(4)                 pytest.raises(nx.NetworkXError, nx_app.traveling_salesman_problem, G)
```

File 90 - test_treewidth.py:

```
(0)             import itertools
(0)             import networkx as nx
(0)             from networkx.algorithms.approximation import (
(4)                 treewidth_min_degree,
(4)                 treewidth_min_fill_in,
(0)             )
(0)             from networkx.algorithms.approximation.treewidth import (
```

```
8: (4)                  MinDegreeHeuristic,
9: (4)                  min_fill_in_heuristic,
10: (0)             )
11: (0)             def is_tree_decomp(graph, decomp):
12: (4)                 """Check if the given tree decomposition is valid."""
13: (4)                 for x in graph.nodes():
14: (8)                     appear_once = False
15: (8)                     for bag in decomp.nodes():
16: (12)                        if x in bag:
17: (16)                            appear_once = True
18: (16)                            break
19: (8)                     assert appear_once
20: (4)                 for x, y in graph.edges():
21: (8)                     appear_together = False
22: (8)                     for bag in decomp.nodes():
23: (12)                        if x in bag and y in bag:
24: (16)                            appear_together = True
25: (16)                            break
26: (8)                     assert appear_together
27: (4)                 for v in graph.nodes():
28: (8)                     subset = []
29: (8)                     for bag in decomp.nodes():
30: (12)                        if v in bag:
31: (16)                            subset.append(bag)
32: (8)                     sub_graph = decomp.subgraph(subset)
33: (8)                     assert nx.is_connected(sub_graph)
34: (0)             class TestTreewidthMinDegree:
35: (4)                 """Unit tests for the min_degree function"""
36: (4)                 @classmethod
37: (4)                 def setup_class(cls):
38: (8)                     """Setup for different kinds of trees"""
39: (8)                     cls.complete = nx.Graph()
40: (8)                     cls.complete.add_edge(1, 2)
41: (8)                     cls.complete.add_edge(2, 3)
42: (8)                     cls.complete.add_edge(1, 3)
43: (8)                     cls.small_tree = nx.Graph()
44: (8)                     cls.small_tree.add_edge(1, 3)
45: (8)                     cls.small_tree.add_edge(4, 3)
46: (8)                     cls.small_tree.add_edge(2, 3)
47: (8)                     cls.small_tree.add_edge(3, 5)
48: (8)                     cls.small_tree.add_edge(5, 6)
49: (8)                     cls.small_tree.add_edge(5, 7)
50: (8)                     cls.small_tree.add_edge(6, 7)
51: (8)                     cls.deterministic_graph = nx.Graph()
52: (8)                     cls.deterministic_graph.add_edge(0, 1)  # deg(0) = 1
53: (8)                     cls.deterministic_graph.add_edge(1, 2)  # deg(1) = 2
54: (8)                     cls.deterministic_graph.add_edge(2, 3)
55: (8)                     cls.deterministic_graph.add_edge(2, 4)  # deg(2) = 3
56: (8)                     cls.deterministic_graph.add_edge(3, 4)
57: (8)                     cls.deterministic_graph.add_edge(3, 5)
58: (8)                     cls.deterministic_graph.add_edge(3, 6)  # deg(3) = 4
59: (8)                     cls.deterministic_graph.add_edge(4, 5)
60: (8)                     cls.deterministic_graph.add_edge(4, 6)
61: (8)                     cls.deterministic_graph.add_edge(4, 7)  # deg(4) = 5
62: (8)                     cls.deterministic_graph.add_edge(5, 6)
63: (8)                     cls.deterministic_graph.add_edge(5, 7)
64: (8)                     cls.deterministic_graph.add_edge(5, 8)
65: (8)                     cls.deterministic_graph.add_edge(5, 9)  # deg(5) = 6
66: (8)                     cls.deterministic_graph.add_edge(6, 7)
67: (8)                     cls.deterministic_graph.add_edge(6, 8)
68: (8)                     cls.deterministic_graph.add_edge(6, 9)  # deg(6) = 6
69: (8)                     cls.deterministic_graph.add_edge(7, 8)
70: (8)                     cls.deterministic_graph.add_edge(7, 9)  # deg(7) = 5
71: (8)                     cls.deterministic_graph.add_edge(8, 9)  # deg(8) = 4
72: (4)                 def test_petersen_graph(self):
73: (8)                     """Test Petersen graph tree decomposition result"""
74: (8)                     G = nx.petersen_graph()
75: (8)                     _, decomp = treewidth_min_degree(G)
76: (8)                     is_tree_decomp(G, decomp)
```

```
 77: (4)                 def test_small_tree_treewidth(self):
 78: (8)                     """Test small tree
 79: (8)                     Test if the computed treewidth of the known self.small_tree is 2.
 80: (8)                     As we know which value we can expect from our heuristic, values other
 81: (8)                     than two are regressions
 82: (8)                     """
 83: (8)                     G = self.small_tree
 84: (8)                     treewidth, _ = treewidth_min_fill_in(G)
 85: (8)                     assert treewidth == 2
 86: (4)                 def test_heuristic_abort(self):
 87: (8)                     """Test heuristic abort condition for fully connected graph"""
 88: (8)                     graph = {}
 89: (8)                     for u in self.complete:
 90: (12)                        graph[u] = set()
 91: (12)                        for v in self.complete[u]:
 92: (16)                            if u != v:  # ignore self-loop
 93: (20)                                graph[u].add(v)
 94: (8)                     deg_heuristic = MinDegreeHeuristic(graph)
 95: (8)                     node = deg_heuristic.best_node(graph)
 96: (8)                     if node is None:
 97: (12)                        pass
 98: (8)                     else:
 99: (12)                        assert False
 100: (4)                 def test_empty_graph(self):
 101: (8)                     """Test empty graph"""
 102: (8)                     G = nx.Graph()
 103: (8)                     _, _ = treewidth_min_degree(G)
 104: (4)                 def test_two_component_graph(self):
 105: (8)                     G = nx.Graph()
 106: (8)                     G.add_node(1)
 107: (8)                     G.add_node(2)
 108: (8)                     treewidth, _ = treewidth_min_degree(G)
 109: (8)                     assert treewidth == 0
 110: (4)                 def test_not_sortable_nodes(self):
 111: (8)                     G = nx.Graph([(0, "a")])
 112: (8)                     treewidth_min_degree(G)
 113: (4)                 def test_heuristic_first_steps(self):
 114: (8)                     """Test first steps of min_degree heuristic"""
 115: (8)                     graph = {
 116: (12)                        n: set(self.deterministic_graph[n]) - {n} for n in
self.deterministic_graph
 117: (8)                     }
 118: (8)                     deg_heuristic = MinDegreeHeuristic(graph)
 119: (8)                     elim_node = deg_heuristic.best_node(graph)
 120: (8)                     print(f"Graph {graph}:")
 121: (8)                     steps = []
 122: (8)                     while elim_node is not None:
 123: (12)                        print(f"Removing {elim_node}:")
 124: (12)                        steps.append(elim_node)
 125: (12)                        nbrs = graph[elim_node]
 126: (12)                        for u, v in itertools.permutations(nbrs, 2):
 127: (16)                            if v not in graph[u]:
 128: (20)                                graph[u].add(v)
 129: (12)                        for u in graph:
 130: (16)                            if elim_node in graph[u]:
 131: (20)                                graph[u].remove(elim_node)
 132: (12)                        del graph[elim_node]
 133: (12)                        print(f"Graph {graph}:")
 134: (12)                        elim_node = deg_heuristic.best_node(graph)
 135: (8)                     assert steps[:5] == [0, 1, 2, 3, 4]
 136: (0)             class TestTreewidthMinFillIn:
 137: (4)                 """Unit tests for the treewidth_min_fill_in function."""
 138: (4)                 @classmethod
 139: (4)                 def setup_class(cls):
 140: (8)                     """Setup for different kinds of trees"""
 141: (8)                     cls.complete = nx.Graph()
 142: (8)                     cls.complete.add_edge(1, 2)
 143: (8)                     cls.complete.add_edge(2, 3)
 144: (8)                     cls.complete.add_edge(1, 3)
```

```
(8)                 cls.small_tree = nx.Graph()
(8)                 cls.small_tree.add_edge(1, 2)
(8)                 cls.small_tree.add_edge(2, 3)
(8)                 cls.small_tree.add_edge(3, 4)
(8)                 cls.small_tree.add_edge(1, 4)
(8)                 cls.small_tree.add_edge(2, 4)
(8)                 cls.small_tree.add_edge(4, 5)
(8)                 cls.small_tree.add_edge(5, 6)
(8)                 cls.small_tree.add_edge(5, 7)
(8)                 cls.small_tree.add_edge(6, 7)
(8)                 cls.deterministic_graph = nx.Graph()
(8)                 cls.deterministic_graph.add_edge(1, 2)
(8)                 cls.deterministic_graph.add_edge(1, 3)
(8)                 cls.deterministic_graph.add_edge(3, 4)
(8)                 cls.deterministic_graph.add_edge(2, 4)
(8)                 cls.deterministic_graph.add_edge(3, 5)
(8)                 cls.deterministic_graph.add_edge(4, 5)
(8)                 cls.deterministic_graph.add_edge(3, 6)
(8)                 cls.deterministic_graph.add_edge(5, 6)
(4)             def test_petersen_graph(self):
(8)                 """Test Petersen graph tree decomposition result"""
(8)                 G = nx.petersen_graph()
(8)                 _, decomp = treewidth_min_fill_in(G)
(8)                 is_tree_decomp(G, decomp)
(4)             def test_small_tree_treewidth(self):
(8)                 """Test if the computed treewidth of the known self.small_tree is 2"""
(8)                 G = self.small_tree
(8)                 treewidth, _ = treewidth_min_fill_in(G)
(8)                 assert treewidth == 2
(4)             def test_heuristic_abort(self):
(8)                 """Test if min_fill_in returns None for fully connected graph"""
(8)                 graph = {}
(8)                 for u in self.complete:
(12)                    graph[u] = set()
(12)                    for v in self.complete[u]:
(16)                        if u != v:  # ignore self-loop
(20)                            graph[u].add(v)
(8)                 next_node = min_fill_in_heuristic(graph)
(8)                 if next_node is None:
(12)                    pass
(8)                 else:
(12)                    assert False
(4)             def test_empty_graph(self):
(8)                 """Test empty graph"""
(8)                 G = nx.Graph()
(8)                 _, _ = treewidth_min_fill_in(G)
(4)             def test_two_component_graph(self):
(8)                 G = nx.Graph()
(8)                 G.add_node(1)
(8)                 G.add_node(2)
(8)                 treewidth, _ = treewidth_min_fill_in(G)
(8)                 assert treewidth == 0
(4)             def test_not_sortable_nodes(self):
(8)                 G = nx.Graph([(0, "a")])
(8)                 treewidth_min_fill_in(G)
(4)             def test_heuristic_first_steps(self):
(8)                 """Test first steps of min_fill_in heuristic"""
(8)                 graph = {
(12)                    n: set(self.deterministic_graph[n]) - {n} for n in
self.deterministic_graph
(8)                 }
(8)                 print(f"Graph {graph}:")
(8)                 elim_node = min_fill_in_heuristic(graph)
(8)                 steps = []
(8)                 while elim_node is not None:
(12)                    print(f"Removing {elim_node}:")
(12)                    steps.append(elim_node)
(12)                    nbrs = graph[elim_node]
(12)                    for u, v in itertools.permutations(nbrs, 2):
```

```
213: (16)                            if v not in graph[u]:
214: (20)                                graph[u].add(v)
215: (12)                        for u in graph:
216: (16)                            if elim_node in graph[u]:
217: (20)                                graph[u].remove(elim_node)
218: (12)                        del graph[elim_node]
219: (12)                        print(f"Graph {graph}:")
220: (12)                        elim_node = min_fill_in_heuristic(graph)
221: (8)                     assert steps[:2] == [6, 5]

----------------------------------------

File 91 - test_vertex_cover.py:

1: (0)              import networkx as nx
2: (0)              from networkx.algorithms.approximation import min_weighted_vertex_cover
3: (0)              def is_cover(G, node_cover):
4: (4)                  return all({u, v} & node_cover for u, v in G.edges())
5: (0)              class TestMWVC:
6: (4)                  """Unit tests for the approximate minimum weighted vertex cover
7: (4)                  function,
8: (4)
:func:`~networkx.algorithms.approximation.vertex_cover.min_weighted_vertex_cover`.
9: (4)                  """
10: (4)                 def test_unweighted_directed(self):
11: (8)                     G = nx.DiGraph()
12: (8)                     G.add_edges_from((0, v) for v in range(1, 26))
13: (8)                     G.add_edges_from((v, 0) for v in range(26, 51))
14: (8)                     cover = min_weighted_vertex_cover(G)
15: (8)                     assert 1 == len(cover)
16: (8)                     assert is_cover(G, cover)
17: (4)                 def test_unweighted_undirected(self):
18: (8)                     size = 50
19: (8)                     sg = nx.star_graph(size)
20: (8)                     cover = min_weighted_vertex_cover(sg)
21: (8)                     assert 1 == len(cover)
22: (8)                     assert is_cover(sg, cover)
23: (4)                 def test_weighted(self):
24: (8)                     wg = nx.Graph()
25: (8)                     wg.add_node(0, weight=10)
26: (8)                     wg.add_node(1, weight=1)
27: (8)                     wg.add_node(2, weight=1)
28: (8)                     wg.add_node(3, weight=1)
29: (8)                     wg.add_node(4, weight=1)
30: (8)                     wg.add_edge(0, 1)
31: (8)                     wg.add_edge(0, 2)
32: (8)                     wg.add_edge(0, 3)
33: (8)                     wg.add_edge(0, 4)
34: (8)                     wg.add_edge(1, 2)
35: (8)                     wg.add_edge(2, 3)
36: (8)                     wg.add_edge(3, 4)
37: (8)                     wg.add_edge(4, 1)
38: (8)                     cover = min_weighted_vertex_cover(wg, weight="weight")
39: (8)                     csum = sum(wg.nodes[node]["weight"] for node in cover)
40: (8)                     assert 4 == csum
41: (8)                     assert is_cover(wg, cover)
42: (4)                 def test_unweighted_self_loop(self):
43: (8)                     slg = nx.Graph()
44: (8)                     slg.add_node(0)
45: (8)                     slg.add_node(1)
46: (8)                     slg.add_node(2)
47: (8)                     slg.add_edge(0, 1)
48: (8)                     slg.add_edge(2, 2)
49: (8)                     cover = min_weighted_vertex_cover(slg)
50: (8)                     assert 2 == len(cover)
51: (8)                     assert is_cover(slg, cover)

----------------------------------------
```

File 92 - __init__.py:

```
1: (0)              from networkx.algorithms.assortativity.connectivity import *
2: (0)              from networkx.algorithms.assortativity.correlation import *
3: (0)              from networkx.algorithms.assortativity.mixing import *
4: (0)              from networkx.algorithms.assortativity.neighbor_degree import *
5: (0)              from networkx.algorithms.assortativity.pairs import *
```

File 93 - connectivity.py:

```
1: (0)              from collections import defaultdict
2: (0)              import networkx as nx
3: (0)              __all__ = ["average_degree_connectivity"]
4: (0)              @nx._dispatchable(edge_attrs="weight")
5: (0)              def average_degree_connectivity(
6: (4)                  G, source="in+out", target="in+out", nodes=None, weight=None
7: (0)              ):
8: (4)                  r"""Compute the average degree connectivity of graph.
9: (4)                  The average degree connectivity is the average nearest neighbor degree of
10: (4)                 nodes with degree k. For weighted graphs, an analogous measure can
11: (4)                 be computed using the weighted average neighbors degree defined in
12: (4)                 [1]_, for a node `i`, as
13: (4)                 .. math::
14: (8)                     k_{nn,i}^{w} = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij} k_j
15: (4)                 where `s_i` is the weighted degree of node `i`,
16: (4)                 `w_{ij}` is the weight of the edge that links `i` and `j`,
17: (4)                 and `N(i)` are the neighbors of node `i`.
18: (4)                 Parameters
19: (4)                 ----------
20: (4)                 G : NetworkX graph
21: (4)                 source :  "in"|"out"|"in+out" (default:"in+out")
22: (7)                    Directed graphs only. Use "in"- or "out"-degree for source node.
23: (4)                 target : "in"|"out"|"in+out" (default:"in+out"
24: (7)                    Directed graphs only. Use "in"- or "out"-degree for target node.
25: (4)                 nodes : list or iterable (optional)
26: (8)                     Compute neighbor connectivity for these nodes. The default is all
27: (8)                     nodes.
28: (4)                 weight : string or None, optional (default=None)
29: (7)                    The edge attribute that holds the numerical value used as a weight.
30: (7)                    If None, then each edge has weight 1.
31: (4)                 Returns
32: (4)                 -------
33: (4)                 d : dict
34: (7)                    A dictionary keyed by degree k with the value of average connectivity.
35: (4)                 Raises
36: (4)                 ------
37: (4)                 NetworkXError
38: (8)                     If either `source` or `target` are not one of 'in',
39: (8)                     'out', or 'in+out'.
40: (8)                     If either `source` or `target` is passed for an undirected graph.
41: (4)                 Examples
42: (4)                 --------
43: (4)                 >>> G = nx.path_graph(4)
44: (4)                 >>> G.edges[1, 2]["weight"] = 3
45: (4)                 >>> nx.average_degree_connectivity(G)
46: (4)                 {1: 2.0, 2: 1.5}
47: (4)                 >>> nx.average_degree_connectivity(G, weight="weight")
48: (4)                 {1: 2.0, 2: 1.75}
49: (4)                 See Also
50: (4)                 --------
51: (4)                 average_neighbor_degree
52: (4)                 References
53: (4)                 ----------
54: (4)                 .. [1] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani,
55: (7)                    "The architecture of complex weighted networks".
56: (7)                    PNAS 101 (11): 3747–3752 (2004).
57: (4)                 """
```

```
58: (4)                 if G.is_directed():
59: (8)                     if source not in ("in", "out", "in+out"):
60: (12)                        raise nx.NetworkXError('source must be one of "in", "out", or
"in+out"')
61: (8)                     if target not in ("in", "out", "in+out"):
62: (12)                        raise nx.NetworkXError('target must be one of "in", "out", or
"in+out"')
63: (8)                     direction = {"out": G.out_degree, "in": G.in_degree, "in+out":
G.degree}
64: (8)                     neighbor_funcs = {
65: (12)                        "out": G.successors,
66: (12)                        "in": G.predecessors,
67: (12)                        "in+out": G.neighbors,
68: (8)                     }
69: (8)                     source_degree = direction[source]
70: (8)                     target_degree = direction[target]
71: (8)                     neighbors = neighbor_funcs[source]
72: (8)                     reverse = source == "in"
73: (4)                 else:
74: (8)                     if source != "in+out" or target != "in+out":
75: (12)                        raise nx.NetworkXError(
76: (16)                            f"source and target arguments are only supported for directed
graphs"
77: (12)                        )
78: (8)                     source_degree = G.degree
79: (8)                     target_degree = G.degree
80: (8)                     neighbors = G.neighbors
81: (8)                     reverse = False
82: (4)                 dsum = defaultdict(int)
83: (4)                 dnorm = defaultdict(int)
84: (4)                 source_nodes = source_degree(nodes)
85: (4)                 if nodes in G:
86: (8)                     source_nodes = [(nodes, source_degree(nodes))]
87: (4)                 for n, k in source_nodes:
88: (8)                     nbrdeg = target_degree(neighbors(n))
89: (8)                     if weight is None:
90: (12)                        s = sum(d for n, d in nbrdeg)
91: (8)                     else:  # weight nbr degree by weight of (n,nbr) edge
92: (12)                        if reverse:
93: (16)                            s = sum(G[nbr][n].get(weight, 1) * d for nbr, d in nbrdeg)
94: (12)                        else:
95: (16)                            s = sum(G[n][nbr].get(weight, 1) * d for nbr, d in nbrdeg)
96: (8)                     dnorm[k] += source_degree(n, weight=weight)
97: (8)                     dsum[k] += s
98: (4)                 return {k: avg if dnorm[k] == 0 else avg / dnorm[k] for k, avg in
dsum.items()}
```

File 94 - correlation.py:

```
1: (0)              """Node assortativity coefficients and correlation measures.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.assortativity.mixing import (
5: (4)                  attribute_mixing_matrix,
6: (4)                  degree_mixing_matrix,
7: (0)              )
8: (0)              from networkx.algorithms.assortativity.pairs import node_degree_xy
9: (0)              __all__ = [
10: (4)                 "degree_pearson_correlation_coefficient",
11: (4)                 "degree_assortativity_coefficient",
12: (4)                 "attribute_assortativity_coefficient",
13: (4)                 "numeric_assortativity_coefficient",
14: (0)             ]
15: (0)             @nx._dispatchable(edge_attrs="weight")
16: (0)             def degree_assortativity_coefficient(G, x="out", y="in", weight=None,
nodes=None):
17: (4)                 """Compute degree assortativity of graph.
```

```
(4)                 Assortativity measures the similarity of connections
(4)                 in the graph with respect to the node degree.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 x: string ('in','out')
(7)                    The degree type for source node (directed graphs only).
(4)                 y: string ('in','out')
(7)                    The degree type for target node (directed graphs only).
(4)                 weight: string or None, optional (default=None)
(7)                    The edge attribute that holds the numerical value used
(7)                    as a weight.  If None, then each edge has weight 1.
(7)                    The degree is the sum of the edge weights adjacent to the node.
(4)                 nodes: list or iterable (optional)
(8)                     Compute degree assortativity only for nodes in container.
(8)                     The default is all nodes.
(4)                 Returns
(4)                 -------
(4)                 r : float
(7)                    Assortativity of graph by degree.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> r = nx.degree_assortativity_coefficient(G)
(4)                 >>> print(f"{r:3.1f}")
(4)                 -0.5
(4)                 See Also
(4)                 --------
(4)                 attribute_assortativity_coefficient
(4)                 numeric_assortativity_coefficient
(4)                 degree_mixing_dict
(4)                 degree_mixing_matrix
(4)                 Notes
(4)                 -----
(4)                 This computes Eq. (21) in Ref. [1]_ , where e is the joint
(4)                 probability distribution (mixing matrix) of the degrees.  If G is
(4)                 directed than the matrix e is the joint probability of the
(4)                 user-specified degree type for the source and target.
(4)                 References
(4)                 ----------
(4)                 .. [1] M. E. J. Newman, Mixing patterns in networks,
(7)                    Physical Review E, 67 026126, 2003
(4)                 .. [2] Foster, J.G., Foster, D.V., Grassberger, P. & Paczuski, M.
(7)                    Edge direction and the structure of networks, PNAS 107, 10815-20
(2010).
(4)                 """
(4)                 if nodes is None:
(8)                     nodes = G.nodes
(4)                 degrees = None
(4)                 if G.is_directed():
(8)                     indeg = (
(12)                        {d for _, d in G.in_degree(nodes, weight=weight)}
(12)                        if "in" in (x, y)
(12)                        else set()
(8)                     )
(8)                     outdeg = (
(12)                        {d for _, d in G.out_degree(nodes, weight=weight)}
(12)                        if "out" in (x, y)
(12)                        else set()
(8)                     )
(8)                     degrees = set.union(indeg, outdeg)
(4)                 else:
(8)                     degrees = {d for _, d in G.degree(nodes, weight=weight)}
(4)                 mapping = {d: i for i, d in enumerate(degrees)}
(4)                 M = degree_mixing_matrix(G, x=x, y=y, nodes=nodes, weight=weight,
mapping=mapping)
(4)                 return _numeric_ac(M, mapping=mapping)
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def degree_pearson_correlation_coefficient(G, x="out", y="in", weight=None,
```

```
nodes=None):
    """Compute degree assortativity of graph.
    Assortativity measures the similarity of connections
    in the graph with respect to the node degree.
    This is the same as degree_assortativity_coefficient but uses the
    potentially faster scipy.stats.pearsonr function.
    Parameters
    ----------
    G : NetworkX graph
    x: string ('in','out')
       The degree type for source node (directed graphs only).
    y: string ('in','out')
       The degree type for target node (directed graphs only).
    weight: string or None, optional (default=None)
       The edge attribute that holds the numerical value used
       as a weight.  If None, then each edge has weight 1.
       The degree is the sum of the edge weights adjacent to the node.
    nodes: list or iterable (optional)
        Compute pearson correlation of degrees only for specified nodes.
        The default is all nodes.
    Returns
    -------
    r : float
       Assortativity of graph by degree.
    Examples
    --------
    >>> G = nx.path_graph(4)
    >>> r = nx.degree_pearson_correlation_coefficient(G)
    >>> print(f"{r:3.1f}")
    -0.5
    Notes
    -----
    This calls scipy.stats.pearsonr.
    References
    ----------
    .. [1] M. E. J. Newman, Mixing patterns in networks
           Physical Review E, 67 026126, 2003
    .. [2] Foster, J.G., Foster, D.V., Grassberger, P. & Paczuski, M.
       Edge direction and the structure of networks, PNAS 107, 10815-20
(2010).
    """
    import scipy as sp
    xy = node_degree_xy(G, x=x, y=y, nodes=nodes, weight=weight)
    x, y = zip(*xy)
    return float(sp.stats.pearsonr(x, y)[0])
@nx._dispatchable(node_attrs="attribute")
def attribute_assortativity_coefficient(G, attribute, nodes=None):
    """Compute assortativity for node attributes.
    Assortativity measures the similarity of connections
    in the graph with respect to the given attribute.
    Parameters
    ----------
    G : NetworkX graph
    attribute : string
        Node attribute key
    nodes: list or iterable (optional)
        Compute attribute assortativity for nodes in container.
        The default is all nodes.
    Returns
    -------
    r: float
       Assortativity of graph for given attribute
    Examples
    --------
    >>> G = nx.Graph()
    >>> G.add_nodes_from([0, 1], color="red")
    >>> G.add_nodes_from([2, 3], color="blue")
    >>> G.add_edges_from([(0, 1), (2, 3)])
    >>> print(nx.attribute_assortativity_coefficient(G, "color"))
```

```
(4)                 1.0
(4)                 Notes
(4)                 -----
(4)                 This computes Eq. (2) in Ref. [1]_ , (trace(M)-sum(M^2))/(1-sum(M^2)),
(4)                 where M is the joint probability distribution (mixing matrix)
(4)                 of the specified attribute.
(4)                 References
(4)                 ----------
(4)                 .. [1] M. E. J. Newman, Mixing patterns in networks,
(7)                    Physical Review E, 67 026126, 2003
(4)                 """
(4)                 M = attribute_mixing_matrix(G, attribute, nodes)
(4)                 return attribute_ac(M)
(0)             @nx._dispatchable(node_attrs="attribute")
(0)             def numeric_assortativity_coefficient(G, attribute, nodes=None):
(4)                 """Compute assortativity for numerical node attributes.
(4)                 Assortativity measures the similarity of connections
(4)                 in the graph with respect to the given numeric attribute.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 attribute : string
(8)                     Node attribute key.
(4)                 nodes: list or iterable (optional)
(8)                     Compute numeric assortativity only for attributes of nodes in
(8)                     container. The default is all nodes.
(4)                 Returns
(4)                 -------
(4)                 r: float
(7)                    Assortativity of graph for given attribute
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_nodes_from([0, 1], size=2)
(4)                 >>> G.add_nodes_from([2, 3], size=3)
(4)                 >>> G.add_edges_from([(0, 1), (2, 3)])
(4)                 >>> print(nx.numeric_assortativity_coefficient(G, "size"))
(4)                 1.0
(4)                 Notes
(4)                 -----
(4)                 This computes Eq. (21) in Ref. [1]_ , which is the Pearson correlation
(4)                 coefficient of the specified (scalar valued) attribute across edges.
(4)                 References
(4)                 ----------
(4)                 .. [1] M. E. J. Newman, Mixing patterns in networks
(11)                       Physical Review E, 67 026126, 2003
(4)                 """
(4)                 if nodes is None:
(8)                     nodes = G.nodes
(4)                 vals = {G.nodes[n][attribute] for n in nodes}
(4)                 mapping = {d: i for i, d in enumerate(vals)}
(4)                 M = attribute_mixing_matrix(G, attribute, nodes, mapping)
(4)                 return _numeric_ac(M, mapping)
(0)             def attribute_ac(M):
(4)                 """Compute assortativity for attribute matrix M.
(4)                 Parameters
(4)                 ----------
(4)                 M : numpy.ndarray
(8)                     2D ndarray representing the attribute mixing matrix.
(4)                 Notes
(4)                 -----
(4)                 This computes Eq. (2) in Ref. [1]_ , (trace(e)-sum(e^2))/(1-sum(e^2)),
(4)                 where e is the joint probability distribution (mixing matrix)
(4)                 of the specified attribute.
(4)                 References
(4)                 ----------
(4)                 .. [1] M. E. J. Newman, Mixing patterns in networks,
(7)                    Physical Review E, 67 026126, 2003
(4)                 """
```

```
221: (4)                if M.sum() != 1.0:
222: (8)                    M = M / M.sum()
223: (4)                s = (M @ M).sum()
224: (4)                t = M.trace()
225: (4)                r = (t - s) / (1 - s)
226: (4)                return float(r)
227: (0)            def _numeric_ac(M, mapping):
228: (4)                import numpy as np
229: (4)                if M.sum() != 1.0:
230: (8)                    M = M / M.sum()
231: (4)                x = np.array(list(mapping.keys()))
232: (4)                y = x  # x and y have the same support
233: (4)                idx = list(mapping.values())
234: (4)                a = M.sum(axis=0)
235: (4)                b = M.sum(axis=1)
236: (4)                vara = (a[idx] * x**2).sum() - ((a[idx] * x).sum()) ** 2
237: (4)                varb = (b[idx] * y**2).sum() - ((b[idx] * y).sum()) ** 2
238: (4)                xy = np.outer(x, y)
239: (4)                ab = np.outer(a[idx], b[idx])
240: (4)                return float((xy * (M - ab)).sum() / np.sqrt(vara * varb))

----------------------------------------

File 95 - mixing.py:

1: (0)              """
2: (0)              Mixing matrices for node attributes and degree.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.assortativity.pairs import node_attribute_xy,
node_degree_xy
6: (0)              from networkx.utils import dict_to_numpy_array
7: (0)              __all__ = [
8: (4)                  "attribute_mixing_matrix",
9: (4)                  "attribute_mixing_dict",
10: (4)                 "degree_mixing_matrix",
11: (4)                 "degree_mixing_dict",
12: (4)                 "mixing_dict",
13: (0)             ]
14: (0)             @nx._dispatchable(node_attrs="attribute")
15: (0)             def attribute_mixing_dict(G, attribute, nodes=None, normalized=False):
16: (4)                 """Returns dictionary representation of mixing matrix for attribute.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : graph
20: (7)                    NetworkX graph object.
21: (4)                 attribute : string
22: (7)                    Node attribute key.
23: (4)                 nodes: list or iterable (optional)
24: (8)                     Unse nodes in container to build the dict. The default is all nodes.
25: (4)                 normalized : bool (default=False)
26: (7)                    Return counts if False or probabilities if True.
27: (4)                 Examples
28: (4)                 --------
29: (4)                 >>> G = nx.Graph()
30: (4)                 >>> G.add_nodes_from([0, 1], color="red")
31: (4)                 >>> G.add_nodes_from([2, 3], color="blue")
32: (4)                 >>> G.add_edge(1, 3)
33: (4)                 >>> d = nx.attribute_mixing_dict(G, "color")
34: (4)                 >>> print(d["red"]["blue"])
35: (4)                 1
36: (4)                 >>> print(d["blue"]["red"])  # d symmetric for undirected graphs
37: (4)                 1
38: (4)                 Returns
39: (4)                 -------
40: (4)                 d : dictionary
41: (7)                    Counts or joint probability of occurrence of attribute pairs.
42: (4)                 """
43: (4)                 xy_iter = node_attribute_xy(G, attribute, nodes)
```

```
    return mixing_dict(xy_iter, normalized=normalized)
@nx._dispatchable(node_attrs="attribute")
def attribute_mixing_matrix(G, attribute, nodes=None, mapping=None, normalized=True):
    """Returns mixing matrix for attribute.
    Parameters
    ----------
    G : graph
       NetworkX graph object.
    attribute : string
       Node attribute key.
    nodes: list or iterable (optional)
        Use only nodes in container to build the matrix. The default is
        all nodes.
    mapping : dictionary, optional
       Mapping from node attribute to integer index in matrix.
       If not specified, an arbitrary ordering will be used.
    normalized : bool (default=True)
       Return counts if False or probabilities if True.
    Returns
    -------
    m: numpy array
       Counts or joint probability of occurrence of attribute pairs.
    Notes
    -----
    If each node has a unique attribute value, the unnormalized mixing matrix
    will be equal to the adjacency matrix. To get a denser mixing matrix,
    the rounding can be performed to form groups of nodes with equal values.
    For example, the exact height of persons in cm (180.79155222, 163.9080892,
    163.30095355, 167.99016217, 168.21590163, ...) can be rounded to (180, 163,
    163, 168, 168, ...).
    Definitions of attribute mixing matrix vary on whether the matrix
    should include rows for attribute values that don't arise. Here we
    do not include such empty-rows. But you can force them to appear
    by inputting a `mapping` that includes those values.
    Examples
    --------
    >>> G = nx.path_graph(3)
    >>> gender = {0: "male", 1: "female", 2: "female"}
    >>> nx.set_node_attributes(G, gender, "gender")
    >>> mapping = {"male": 0, "female": 1}
    >>> mix_mat = nx.attribute_mixing_matrix(G, "gender", mapping=mapping)
    >>> mix_mat
    array([[0.  , 0.25],
           [0.25, 0.5 ]])
    """
    d = attribute_mixing_dict(G, attribute, nodes)
    a = dict_to_numpy_array(d, mapping=mapping)
    if normalized:
        a = a / a.sum()
    return a
@nx._dispatchable(edge_attrs="weight")
def degree_mixing_dict(G, x="out", y="in", weight=None, nodes=None, normalized=False):
    """Returns dictionary representation of mixing matrix for degree.
    Parameters
    ----------
    G : graph
        NetworkX graph object.
    x: string ('in','out')
       The degree type for source node (directed graphs only).
    y: string ('in','out')
       The degree type for target node (directed graphs only).
    weight: string or None, optional (default=None)
       The edge attribute that holds the numerical value used
       as a weight.  If None, then each edge has weight 1.
       The degree is the sum of the edge weights adjacent to the node.
    normalized : bool (default=False)
```

```
(8)                 Return counts if False or probabilities if True.
(4)             Returns
(4)             -------
(4)             d: dictionary
(7)                Counts or joint probability of occurrence of degree pairs.
(4)             """
(4)             xy_iter = node_degree_xy(G, x=x, y=y, nodes=nodes, weight=weight)
(4)             return mixing_dict(xy_iter, normalized=normalized)
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def degree_mixing_matrix(
(4)             G, x="out", y="in", weight=None, nodes=None, normalized=True, mapping=None
(0)         ):
(4)             """Returns mixing matrix for attribute.
(4)             Parameters
(4)             ----------
(4)             G : graph
(7)                NetworkX graph object.
(4)             x: string ('in','out')
(7)                The degree type for source node (directed graphs only).
(4)             y: string ('in','out')
(7)                The degree type for target node (directed graphs only).
(4)             nodes: list or iterable (optional)
(8)                 Build the matrix using only nodes in container.
(8)                 The default is all nodes.
(4)             weight: string or None, optional (default=None)
(7)                The edge attribute that holds the numerical value used
(7)                as a weight.  If None, then each edge has weight 1.
(7)                The degree is the sum of the edge weights adjacent to the node.
(4)             normalized : bool (default=True)
(7)                Return counts if False or probabilities if True.
(4)             mapping : dictionary, optional
(7)                Mapping from node degree to integer index in matrix.
(7)                If not specified, an arbitrary ordering will be used.
(4)             Returns
(4)             -------
(4)             m: numpy array
(7)                Counts, or joint probability, of occurrence of node degree.
(4)             Notes
(4)             -----
(4)             Definitions of degree mixing matrix vary on whether the matrix
(4)             should include rows for degree values that don't arise. Here we
(4)             do not include such empty-rows. But you can force them to appear
(4)             by inputting a `mapping` that includes those values. See examples.
(4)             Examples
(4)             --------
(4)             >>> G = nx.star_graph(3)
(4)             >>> mix_mat = nx.degree_mixing_matrix(G)
(4)             >>> mix_mat
(4)             array([[0. , 0.5],
(11)                   [0.5, 0. ]])
(4)             If you want every possible degree to appear as a row, even if no nodes
(4)             have that degree, use `mapping` as follows,
(4)             >>> max_degree = max(deg for n, deg in G.degree)
(4)             >>> mapping = {x: x for x in range(max_degree + 1)}  # identity mapping
(4)             >>> mix_mat = nx.degree_mixing_matrix(G, mapping=mapping)
(4)             >>> mix_mat
(4)             array([[0. , 0. , 0. , 0. ],
(11)                   [0. , 0. , 0. , 0.5],
(11)                   [0. , 0. , 0. , 0. ],
(11)                   [0. , 0.5, 0. , 0. ]])
(4)             """
(4)             d = degree_mixing_dict(G, x=x, y=y, nodes=nodes, weight=weight)
(4)             a = dict_to_numpy_array(d, mapping=mapping)
(4)             if normalized:
(8)                 a = a / a.sum()
(4)             return a
(0)         def mixing_dict(xy, normalized=False):
(4)             """Returns a dictionary representation of mixing matrix.
(4)             Parameters
```

```
179: (4)                ----------
180: (4)                xy : list or container of two-tuples
181: (7)                   Pairs of (x,y) items.
182: (4)                attribute : string
183: (7)                   Node attribute key
184: (4)                normalized : bool (default=False)
185: (7)                   Return counts if False or probabilities if True.
186: (4)                Returns
187: (4)                -------
188: (4)                d: dictionary
189: (7)                   Counts or Joint probability of occurrence of values in xy.
190: (4)                """
191: (4)                d = {}
192: (4)                psum = 0.0
193: (4)                for x, y in xy:
194: (8)                    if x not in d:
195: (12)                       d[x] = {}
196: (8)                    if y not in d:
197: (12)                       d[y] = {}
198: (8)                    v = d[x].get(y, 0)
199: (8)                    d[x][y] = v + 1
200: (8)                    psum += 1
201: (4)                if normalized:
202: (8)                    for _, jdict in d.items():
203: (12)                       for j in jdict:
204: (16)                           jdict[j] /= psum
205: (4)                return d
```

File 96 - neighbor_degree.py:

```
1: (0)              import networkx as nx
2: (0)              __all__ = ["average_neighbor_degree"]
3: (0)              @nx._dispatchable(edge_attrs="weight")
4: (0)              def average_neighbor_degree(G, source="out", target="out", nodes=None,
weight=None):
5: (4)                  r"""Returns the average degree of the neighborhood of each node.
6: (4)                  In an undirected graph, the neighborhood `N(i)` of node `i` contains the
7: (4)                  nodes that are connected to `i` by an edge.
8: (4)                  For directed graphs, `N(i)` is defined according to the parameter
`source`:
9: (8)                      - if source is 'in', then `N(i)` consists of predecessors of node `i`.
10: (8)                     - if source is 'out', then `N(i)` consists of successors of node `i`.
11: (8)                     - if source is 'in+out', then `N(i)` is both predecessors and
successors.
12: (4)                 The average neighborhood degree of a node `i` is
13: (4)                 .. math::
14: (8)                     k_{nn,i} = \frac{1}{|N(i)|} \sum_{j \in N(i)} k_j
15: (4)                 where `N(i)` are the neighbors of node `i` and `k_j` is
16: (4)                 the degree of node `j` which belongs to `N(i)`. For weighted
17: (4)                 graphs, an analogous measure can be defined [1]_,
18: (4)                 .. math::
19: (8)                     k_{nn,i}^{w} = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij} k_j
20: (4)                 where `s_i` is the weighted degree of node `i`, `w_{ij}`
21: (4)                 is the weight of the edge that links `i` and `j` and
22: (4)                 `N(i)` are the neighbors of node `i`.
23: (4)                 Parameters
24: (4)                 ----------
25: (4)                 G : NetworkX graph
26: (4)                 source : string ("in"|"out"|"in+out"), optional (default="out")
27: (7)                    Directed graphs only.
28: (7)                    Use "in"- or "out"-neighbors of source node.
29: (4)                 target : string ("in"|"out"|"in+out"), optional (default="out")
30: (7)                    Directed graphs only.
31: (7)                    Use "in"- or "out"-degree for target node.
32: (4)                 nodes : list or iterable, optional (default=G.nodes)
33: (8)                     Compute neighbor degree only for specified nodes.
34: (4)                 weight : string or None, optional (default=None)
```

```
 35: (7)                    The edge attribute that holds the numerical value used as a weight.
 36: (7)                    If None, then each edge has weight 1.
 37: (4)                Returns
 38: (4)                -------
 39: (4)                d: dict
 40: (7)                    A dictionary keyed by node to the average degree of its neighbors.
 41: (4)                Raises
 42: (4)                ------
 43: (4)                NetworkXError
 44: (8)                    If either `source` or `target` are not one of 'in', 'out', or
'in+out'.
 45: (8)                    If either `source` or `target` is passed for an undirected graph.
 46: (4)                Examples
 47: (4)                --------
 48: (4)                >>> G = nx.path_graph(4)
 49: (4)                >>> G.edges[0, 1]["weight"] = 5
 50: (4)                >>> G.edges[2, 3]["weight"] = 3
 51: (4)                >>> nx.average_neighbor_degree(G)
 52: (4)                {0: 2.0, 1: 1.5, 2: 1.5, 3: 2.0}
 53: (4)                >>> nx.average_neighbor_degree(G, weight="weight")
 54: (4)                {0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}
 55: (4)                >>> G = nx.DiGraph()
 56: (4)                >>> nx.add_path(G, [0, 1, 2, 3])
 57: (4)                >>> nx.average_neighbor_degree(G, source="in", target="in")
 58: (4)                {0: 0.0, 1: 0.0, 2: 1.0, 3: 1.0}
 59: (4)                >>> nx.average_neighbor_degree(G, source="out", target="out")
 60: (4)                {0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}
 61: (4)                See Also
 62: (4)                --------
 63: (4)                average_degree_connectivity
 64: (4)                References
 65: (4)                ----------
 66: (4)                .. [1] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani,
 67: (7)                   "The architecture of complex weighted networks".
 68: (7)                   PNAS 101 (11): 3747–3752 (2004).
 69: (4)                """
 70: (4)                if G.is_directed():
 71: (8)                    if source == "in":
 72: (12)                       source_degree = G.in_degree
 73: (8)                    elif source == "out":
 74: (12)                       source_degree = G.out_degree
 75: (8)                    elif source == "in+out":
 76: (12)                       source_degree = G.degree
 77: (8)                    else:
 78: (12)                       raise nx.NetworkXError(
 79: (16)                           f"source argument {source} must be 'in', 'out' or 'in+out'"
 80: (12)                       )
 81: (8)                    if target == "in":
 82: (12)                       target_degree = G.in_degree
 83: (8)                    elif target == "out":
 84: (12)                       target_degree = G.out_degree
 85: (8)                    elif target == "in+out":
 86: (12)                       target_degree = G.degree
 87: (8)                    else:
 88: (12)                       raise nx.NetworkXError(
 89: (16)                           f"target argument {target} must be 'in', 'out' or 'in+out'"
 90: (12)                       )
 91: (4)                else:
 92: (8)                    if source != "out" or target != "out":
 93: (12)                       raise nx.NetworkXError(
 94: (16)                           f"source and target arguments are only supported for directed
graphs"
 95: (12)                       )
 96: (8)                    source_degree = target_degree = G.degree
 97: (4)                t_deg = dict(target_degree())
 98: (4)                G_P = G_S = {n: {} for n in G}
 99: (4)                if G.is_directed():
100: (8)                    if "in" in source:
101: (12)                       G_P = G.pred
```

```
102: (8)                    if "out" in source:
103: (12)                       G_S = G.succ
104: (4)                else:
105: (8)                    G_S = G.adj
106: (4)                avg = {}
107: (4)                for n, deg in source_degree(nodes, weight=weight):
108: (8)                    if deg == 0:
109: (12)                       avg[n] = 0.0
110: (12)                       continue
111: (8)                    if weight is None:
112: (12)                       avg[n] = (
113: (16)                           sum(t_deg[nbr] for nbr in G_S[n]) + sum(t_deg[nbr] for nbr in
G_P[n])
114: (12)                       ) / deg
115: (8)                    else:
116: (12)                       avg[n] = (
117: (16)                           sum(dd.get(weight, 1) * t_deg[nbr] for nbr, dd in
G_S[n].items())
118: (16)                           + sum(dd.get(weight, 1) * t_deg[nbr] for nbr, dd in
G_P[n].items())
119: (12)                       ) / deg
120: (4)                return avg

----------------------------------------

File 97 - pairs.py:

1: (0)              """Generators of  x-y pairs of node data."""
2: (0)              import networkx as nx
3: (0)              __all__ = ["node_attribute_xy", "node_degree_xy"]
4: (0)              @nx._dispatchable(node_attrs="attribute")
5: (0)              def node_attribute_xy(G, attribute, nodes=None):
6: (4)                  """Returns iterator of node-attribute pairs for all edges in G.
7: (4)                  Parameters
8: (4)                  ----------
9: (4)                  G: NetworkX graph
10: (4)                 attribute: key
11: (7)                    The node attribute key.
12: (4)                 nodes: list or iterable (optional)
13: (8)                     Use only edges that are incident to specified nodes.
14: (8)                     The default is all nodes.
15: (4)                 Returns
16: (4)                 -------
17: (4)                 (x, y): 2-tuple
18: (8)                     Generates 2-tuple of (attribute, attribute) values.
19: (4)                 Examples
20: (4)                 --------
21: (4)                 >>> G = nx.DiGraph()
22: (4)                 >>> G.add_node(1, color="red")
23: (4)                 >>> G.add_node(2, color="blue")
24: (4)                 >>> G.add_edge(1, 2)
25: (4)                 >>> list(nx.node_attribute_xy(G, "color"))
26: (4)                 [('red', 'blue')]
27: (4)                 Notes
28: (4)                 -----
29: (4)                 For undirected graphs each edge is produced twice, once for each edge
30: (4)                 representation (u, v) and (v, u), with the exception of self-loop edges
31: (4)                 which only appear once.
32: (4)                 """
33: (4)                 if nodes is None:
34: (8)                     nodes = set(G)
35: (4)                 else:
36: (8)                     nodes = set(nodes)
37: (4)                 Gnodes = G.nodes
38: (4)                 for u, nbrsdict in G.adjacency():
39: (8)                     if u not in nodes:
40: (12)                        continue
41: (8)                     uattr = Gnodes[u].get(attribute, None)
42: (8)                     if G.is_multigraph():
```

```
43: (12)                        for v, keys in nbrsdict.items():
44: (16)                            vattr = Gnodes[v].get(attribute, None)
45: (16)                            for _ in keys:
46: (20)                                yield (uattr, vattr)
47: (8)                     else:
48: (12)                        for v in nbrsdict:
49: (16)                            vattr = Gnodes[v].get(attribute, None)
50: (16)                            yield (uattr, vattr)
51: (0)             @nx._dispatchable(edge_attrs="weight")
52: (0)             def node_degree_xy(G, x="out", y="in", weight=None, nodes=None):
53: (4)                 """Generate node degree-degree pairs for edges in G.
54: (4)                 Parameters
55: (4)                 ----------
56: (4)                 G: NetworkX graph
57: (4)                 x: string ('in','out')
58: (7)                    The degree type for source node (directed graphs only).
59: (4)                 y: string ('in','out')
60: (7)                    The degree type for target node (directed graphs only).
61: (4)                 weight: string or None, optional (default=None)
62: (7)                    The edge attribute that holds the numerical value used
63: (7)                    as a weight.  If None, then each edge has weight 1.
64: (7)                    The degree is the sum of the edge weights adjacent to the node.
65: (4)                 nodes: list or iterable (optional)
66: (8)                     Use only edges that are adjacency to specified nodes.
67: (8)                     The default is all nodes.
68: (4)                 Returns
69: (4)                 -------
70: (4)                 (x, y): 2-tuple
71: (8)                     Generates 2-tuple of (degree, degree) values.
72: (4)                 Examples
73: (4)                 --------
74: (4)                 >>> G = nx.DiGraph()
75: (4)                 >>> G.add_edge(1, 2)
76: (4)                 >>> list(nx.node_degree_xy(G, x="out", y="in"))
77: (4)                 [(1, 1)]
78: (4)                 >>> list(nx.node_degree_xy(G, x="in", y="out"))
79: (4)                 [(0, 0)]
80: (4)                 Notes
81: (4)                 -----
82: (4)                 For undirected graphs each edge is produced twice, once for each edge
83: (4)                 representation (u, v) and (v, u), with the exception of self-loop edges
84: (4)                 which only appear once.
85: (4)                 """
86: (4)                 nodes = set(G) if nodes is None else set(nodes)
87: (4)                 if G.is_directed():
88: (8)                     direction = {"out": G.out_degree, "in": G.in_degree}
89: (8)                     xdeg = direction[x]
90: (8)                     ydeg = direction[y]
91: (4)                 else:
92: (8)                     xdeg = ydeg = G.degree
93: (4)                 for u, degu in xdeg(nodes, weight=weight):
94: (8)                     neighbors = (nbr for _, nbr in G.edges(u) if nbr in nodes)
95: (8)                     for _, degv in ydeg(neighbors, weight=weight):
96: (12)                        yield degu, degv
```

File 98 - __init__.py:

```
1: (0)
```

File 99 - base_test.py:

```
1: (0)              import networkx as nx
2: (0)              class BaseTestAttributeMixing:
3: (4)                  @classmethod
4: (4)                  def setup_class(cls):
```

```
5: (8)                      G = nx.Graph()
6: (8)                      G.add_nodes_from([0, 1], fish="one")
7: (8)                      G.add_nodes_from([2, 3], fish="two")
8: (8)                      G.add_nodes_from([4], fish="red")
9: (8)                      G.add_nodes_from([5], fish="blue")
10: (8)                     G.add_edges_from([(0, 1), (2, 3), (0, 4), (2, 5)])
11: (8)                     cls.G = G
12: (8)                     D = nx.DiGraph()
13: (8)                     D.add_nodes_from([0, 1], fish="one")
14: (8)                     D.add_nodes_from([2, 3], fish="two")
15: (8)                     D.add_nodes_from([4], fish="red")
16: (8)                     D.add_nodes_from([5], fish="blue")
17: (8)                     D.add_edges_from([(0, 1), (2, 3), (0, 4), (2, 5)])
18: (8)                     cls.D = D
19: (8)                     M = nx.MultiGraph()
20: (8)                     M.add_nodes_from([0, 1], fish="one")
21: (8)                     M.add_nodes_from([2, 3], fish="two")
22: (8)                     M.add_nodes_from([4], fish="red")
23: (8)                     M.add_nodes_from([5], fish="blue")
24: (8)                     M.add_edges_from([(0, 1), (0, 1), (2, 3)])
25: (8)                     cls.M = M
26: (8)                     S = nx.Graph()
27: (8)                     S.add_nodes_from([0, 1], fish="one")
28: (8)                     S.add_nodes_from([2, 3], fish="two")
29: (8)                     S.add_nodes_from([4], fish="red")
30: (8)                     S.add_nodes_from([5], fish="blue")
31: (8)                     S.add_edge(0, 0)
32: (8)                     S.add_edge(2, 2)
33: (8)                     cls.S = S
34: (8)                     N = nx.Graph()
35: (8)                     N.add_nodes_from([0, 1], margin=-2)
36: (8)                     N.add_nodes_from([2, 3], margin=-2)
37: (8)                     N.add_nodes_from([4], margin=-3)
38: (8)                     N.add_nodes_from([5], margin=-4)
39: (8)                     N.add_edges_from([(0, 1), (2, 3), (0, 4), (2, 5)])
40: (8)                     cls.N = N
41: (8)                     F = nx.Graph()
42: (8)                     F.add_edges_from([(0, 3), (1, 3), (2, 3)], weight=0.5)
43: (8)                     F.add_edge(0, 2, weight=1)
44: (8)                     nx.set_node_attributes(F, dict(F.degree(weight="weight")), "margin")
45: (8)                     cls.F = F
46: (8)                     K = nx.Graph()
47: (8)                     K.add_nodes_from([1, 2], margin=-1)
48: (8)                     K.add_nodes_from([3], margin=1)
49: (8)                     K.add_nodes_from([4], margin=2)
50: (8)                     K.add_edges_from([(3, 4), (1, 2), (1, 3)])
51: (8)                     cls.K = K
52: (0)             class BaseTestDegreeMixing:
53: (4)                 @classmethod
54: (4)                 def setup_class(cls):
55: (8)                     cls.P4 = nx.path_graph(4)
56: (8)                     cls.D = nx.DiGraph()
57: (8)                     cls.D.add_edges_from([(0, 2), (0, 3), (1, 3), (2, 3)])
58: (8)                     cls.D2 = nx.DiGraph()
59: (8)                     cls.D2.add_edges_from([(0, 3), (1, 0), (1, 2), (2, 4), (4, 1), (4, 3),
(4, 2)])
60: (8)                     cls.M = nx.MultiGraph()
61: (8)                     nx.add_path(cls.M, range(4))
62: (8)                     cls.M.add_edge(0, 1)
63: (8)                     cls.S = nx.Graph()
64: (8)                     cls.S.add_edges_from([(0, 0), (1, 1)])
65: (8)                     cls.W = nx.Graph()
66: (8)                     cls.W.add_edges_from([(0, 3), (1, 3), (2, 3)], weight=0.5)
67: (8)                     cls.W.add_edge(0, 2, weight=1)
68: (8)                     S1 = nx.star_graph(4)
69: (8)                     S2 = nx.star_graph(4)
70: (8)                     cls.DS = nx.disjoint_union(S1, S2)
71: (8)                     cls.DS.add_edge(4, 5)
```

```
----------------------------------------

File 100 - test_connectivity.py:

1: (0)              from itertools import permutations
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestNeighborConnectivity:
5: (4)                  def test_degree_p4(self):
6: (8)                      G = nx.path_graph(4)
7: (8)                      answer = {1: 2.0, 2: 1.5}
8: (8)                      nd = nx.average_degree_connectivity(G)
9: (8)                      assert nd == answer
10: (8)                     D = G.to_directed()
11: (8)                     answer = {2: 2.0, 4: 1.5}
12: (8)                     nd = nx.average_degree_connectivity(D)
13: (8)                     assert nd == answer
14: (8)                     answer = {1: 2.0, 2: 1.5}
15: (8)                     D = G.to_directed()
16: (8)                     nd = nx.average_degree_connectivity(D, source="in", target="in")
17: (8)                     assert nd == answer
18: (8)                     D = G.to_directed()
19: (8)                     nd = nx.average_degree_connectivity(D, source="in", target="in")
20: (8)                     assert nd == answer
21: (4)                 def test_degree_p4_weighted(self):
22: (8)                     G = nx.path_graph(4)
23: (8)                     G[1][2]["weight"] = 4
24: (8)                     answer = {1: 2.0, 2: 1.8}
25: (8)                     nd = nx.average_degree_connectivity(G, weight="weight")
26: (8)                     assert nd == answer
27: (8)                     answer = {1: 2.0, 2: 1.5}
28: (8)                     nd = nx.average_degree_connectivity(G)
29: (8)                     assert nd == answer
30: (8)                     D = G.to_directed()
31: (8)                     answer = {2: 2.0, 4: 1.8}
32: (8)                     nd = nx.average_degree_connectivity(D, weight="weight")
33: (8)                     assert nd == answer
34: (8)                     answer = {1: 2.0, 2: 1.8}
35: (8)                     D = G.to_directed()
36: (8)                     nd = nx.average_degree_connectivity(
37: (12)                        D, weight="weight", source="in", target="in"
38: (8)                     )
39: (8)                     assert nd == answer
40: (8)                     D = G.to_directed()
41: (8)                     nd = nx.average_degree_connectivity(
42: (12)                        D, source="in", target="out", weight="weight"
43: (8)                     )
44: (8)                     assert nd == answer
45: (4)                 def test_weight_keyword(self):
46: (8)                     G = nx.path_graph(4)
47: (8)                     G[1][2]["other"] = 4
48: (8)                     answer = {1: 2.0, 2: 1.8}
49: (8)                     nd = nx.average_degree_connectivity(G, weight="other")
50: (8)                     assert nd == answer
51: (8)                     answer = {1: 2.0, 2: 1.5}
52: (8)                     nd = nx.average_degree_connectivity(G, weight=None)
53: (8)                     assert nd == answer
54: (8)                     D = G.to_directed()
55: (8)                     answer = {2: 2.0, 4: 1.8}
56: (8)                     nd = nx.average_degree_connectivity(D, weight="other")
57: (8)                     assert nd == answer
58: (8)                     answer = {1: 2.0, 2: 1.8}
59: (8)                     D = G.to_directed()
60: (8)                     nd = nx.average_degree_connectivity(D, weight="other", source="in",
target="in")
61: (8)                     assert nd == answer
62: (8)                     D = G.to_directed()
63: (8)                     nd = nx.average_degree_connectivity(D, weight="other", source="in",
target="in")
```

```
64: (8)                     assert nd == answer
65: (4)                 def test_degree_barrat(self):
66: (8)                     G = nx.star_graph(5)
67: (8)                     G.add_edges_from([(5, 6), (5, 7), (5, 8), (5, 9)])
68: (8)                     G[0][5]["weight"] = 5
69: (8)                     nd = nx.average_degree_connectivity(G)[5]
70: (8)                     assert nd == 1.8
71: (8)                     nd = nx.average_degree_connectivity(G, weight="weight")[5]
72: (8)                     assert nd == pytest.approx(3.222222, abs=1e-5)
73: (4)                 def test_zero_deg(self):
74: (8)                     G = nx.DiGraph()
75: (8)                     G.add_edge(1, 2)
76: (8)                     G.add_edge(1, 3)
77: (8)                     G.add_edge(1, 4)
78: (8)                     c = nx.average_degree_connectivity(G)
79: (8)                     assert c == {1: 0, 3: 1}
80: (8)                     c = nx.average_degree_connectivity(G, source="in", target="in")
81: (8)                     assert c == {0: 0, 1: 0}
82: (8)                     c = nx.average_degree_connectivity(G, source="in", target="out")
83: (8)                     assert c == {0: 0, 1: 3}
84: (8)                     c = nx.average_degree_connectivity(G, source="in", target="in+out")
85: (8)                     assert c == {0: 0, 1: 3}
86: (8)                     c = nx.average_degree_connectivity(G, source="out", target="out")
87: (8)                     assert c == {0: 0, 3: 0}
88: (8)                     c = nx.average_degree_connectivity(G, source="out", target="in")
89: (8)                     assert c == {0: 0, 3: 1}
90: (8)                     c = nx.average_degree_connectivity(G, source="out", target="in+out")
91: (8)                     assert c == {0: 0, 3: 1}
92: (4)                 def test_in_out_weight(self):
93: (8)                     G = nx.DiGraph()
94: (8)                     G.add_edge(1, 2, weight=1)
95: (8)                     G.add_edge(1, 3, weight=1)
96: (8)                     G.add_edge(3, 1, weight=1)
97: (8)                     for s, t in permutations(["in", "out", "in+out"], 2):
98: (12)                        c = nx.average_degree_connectivity(G, source=s, target=t)
99: (12)                        cw = nx.average_degree_connectivity(G, source=s, target=t,
weight="weight")
100: (12)                       assert c == cw
101: (4)                 def test_invalid_source(self):
102: (8)                    with pytest.raises(nx.NetworkXError):
103: (12)                       G = nx.DiGraph()
104: (12)                       nx.average_degree_connectivity(G, source="bogus")
105: (4)                 def test_invalid_target(self):
106: (8)                    with pytest.raises(nx.NetworkXError):
107: (12)                       G = nx.DiGraph()
108: (12)                       nx.average_degree_connectivity(G, target="bogus")
109: (4)                 def test_invalid_undirected_graph(self):
110: (8)                    G = nx.Graph()
111: (8)                    with pytest.raises(nx.NetworkXError):
112: (12)                       nx.average_degree_connectivity(G, target="bogus")
113: (8)                    with pytest.raises(nx.NetworkXError):
114: (12)                       nx.average_degree_connectivity(G, source="bogus")
115: (4)                 def test_single_node(self):
116: (8)                    G = nx.trivial_graph()
117: (8)                    conn = nx.average_degree_connectivity(G, nodes=0)
118: (8)                    assert conn == {0: 0}

----------------------------------------

File 101 - test_correlation.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.assortativity.correlation import attribute_ac
6: (0)              from .base_test import BaseTestAttributeMixing, BaseTestDegreeMixing
7: (0)              class TestDegreeMixingCorrelation(BaseTestDegreeMixing):
8: (4)                  def test_degree_assortativity_undirected(self):
```

```
 9: (8)                        r = nx.degree_assortativity_coefficient(self.P4)
10: (8)                        np.testing.assert_almost_equal(r, -1.0 / 2, decimal=4)
11: (4)                    def test_degree_assortativity_node_kwargs(self):
12: (8)                        G = nx.Graph()
13: (8)                        edges = [(0, 1), (0, 3), (1, 2), (1, 3), (1, 4), (5, 9), (9, 0)]
14: (8)                        G.add_edges_from(edges)
15: (8)                        r = nx.degree_assortativity_coefficient(G, nodes=[1, 2, 4])
16: (8)                        np.testing.assert_almost_equal(r, -1.0, decimal=4)
17: (4)                    def test_degree_assortativity_directed(self):
18: (8)                        r = nx.degree_assortativity_coefficient(self.D)
19: (8)                        np.testing.assert_almost_equal(r, -0.57735, decimal=4)
20: (4)                    def test_degree_assortativity_directed2(self):
21: (8)                        """Test degree assortativity for a directed graph where the set of
22: (8)                        in/out degree does not equal the total degree."""
23: (8)                        r = nx.degree_assortativity_coefficient(self.D2)
24: (8)                        np.testing.assert_almost_equal(r, 0.14852, decimal=4)
25: (4)                    def test_degree_assortativity_multigraph(self):
26: (8)                        r = nx.degree_assortativity_coefficient(self.M)
27: (8)                        np.testing.assert_almost_equal(r, -1.0 / 7.0, decimal=4)
28: (4)                    def test_degree_pearson_assortativity_undirected(self):
29: (8)                        r = nx.degree_pearson_correlation_coefficient(self.P4)
30: (8)                        np.testing.assert_almost_equal(r, -1.0 / 2, decimal=4)
31: (4)                    def test_degree_pearson_assortativity_directed(self):
32: (8)                        r = nx.degree_pearson_correlation_coefficient(self.D)
33: (8)                        np.testing.assert_almost_equal(r, -0.57735, decimal=4)
34: (4)                    def test_degree_pearson_assortativity_directed2(self):
35: (8)                        """Test degree assortativity with Pearson for a directed graph where
36: (8)                        the set of in/out degree does not equal the total degree."""
37: (8)                        r = nx.degree_pearson_correlation_coefficient(self.D2)
38: (8)                        np.testing.assert_almost_equal(r, 0.14852, decimal=4)
39: (4)                    def test_degree_pearson_assortativity_multigraph(self):
40: (8)                        r = nx.degree_pearson_correlation_coefficient(self.M)
41: (8)                        np.testing.assert_almost_equal(r, -1.0 / 7.0, decimal=4)
42: (4)                    def test_degree_assortativity_weighted(self):
43: (8)                        r = nx.degree_assortativity_coefficient(self.W, weight="weight")
44: (8)                        np.testing.assert_almost_equal(r, -0.1429, decimal=4)
45: (4)                    def test_degree_assortativity_double_star(self):
46: (8)                        r = nx.degree_assortativity_coefficient(self.DS)
47: (8)                        np.testing.assert_almost_equal(r, -0.9339, decimal=4)
48: (0)                class TestAttributeMixingCorrelation(BaseTestAttributeMixing):
49: (4)                    def test_attribute_assortativity_undirected(self):
50: (8)                        r = nx.attribute_assortativity_coefficient(self.G, "fish")
51: (8)                        assert r == 6.0 / 22.0
52: (4)                    def test_attribute_assortativity_directed(self):
53: (8)                        r = nx.attribute_assortativity_coefficient(self.D, "fish")
54: (8)                        assert r == 1.0 / 3.0
55: (4)                    def test_attribute_assortativity_multigraph(self):
56: (8)                        r = nx.attribute_assortativity_coefficient(self.M, "fish")
57: (8)                        assert r == 1.0
58: (4)                    def test_attribute_assortativity_coefficient(self):
59: (8)                        a = np.array([[0.258, 0.016, 0.035, 0.013],
60: (22)                                     [0.012, 0.157, 0.058, 0.019],
61: (22)                                     [0.013, 0.023, 0.306, 0.035],
62: (22)                                     [0.005, 0.007, 0.024, 0.016]])
63: (8)                        r = attribute_ac(a)
64: (8)                        np.testing.assert_almost_equal(r, 0.623, decimal=3)
65: (4)                    def test_attribute_assortativity_coefficient2(self):
66: (8)                        a = np.array([[0.18, 0.02, 0.01, 0.03],
67: (22)                                     [0.02, 0.20, 0.03, 0.02],
68: (22)                                     [0.01, 0.03, 0.16, 0.01],
69: (22)                                     [0.03, 0.02, 0.01, 0.22]])
70: (8)                        r = attribute_ac(a)
71: (8)                        np.testing.assert_almost_equal(r, 0.68, decimal=2)
72: (4)                    def test_attribute_assortativity(self):
73: (8)                        a = np.array([[50, 50, 0], [50, 50, 0], [0, 0, 2]])
74: (8)                        r = attribute_ac(a)
75: (8)                        np.testing.assert_almost_equal(r, 0.029, decimal=3)
76: (4)                    def test_attribute_assortativity_negative(self):
77: (8)                        r = nx.numeric_assortativity_coefficient(self.N, "margin")
```

```
78: (8)                     np.testing.assert_almost_equal(r, -0.2903, decimal=4)
79: (4)                 def test_assortativity_node_kwargs(self):
80: (8)                     G = nx.Graph()
81: (8)                     G.add_nodes_from([0, 1], size=2)
82: (8)                     G.add_nodes_from([2, 3], size=3)
83: (8)                     G.add_edges_from([(0, 1), (2, 3)])
84: (8)                     r = nx.numeric_assortativity_coefficient(G, "size", nodes=[0, 3])
85: (8)                     np.testing.assert_almost_equal(r, 1.0, decimal=4)
86: (4)                 def test_attribute_assortativity_float(self):
87: (8)                     r = nx.numeric_assortativity_coefficient(self.F, "margin")
88: (8)                     np.testing.assert_almost_equal(r, -0.1429, decimal=4)
89: (4)                 def test_attribute_assortativity_mixed(self):
90: (8)                     r = nx.numeric_assortativity_coefficient(self.K, "margin")
91: (8)                     np.testing.assert_almost_equal(r, 0.4340, decimal=4)
```

File 102 - test_mixing.py:

```
1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              import networkx as nx
4: (0)              from .base_test import BaseTestAttributeMixing, BaseTestDegreeMixing
5: (0)              class TestDegreeMixingDict(BaseTestDegreeMixing):
6: (4)                  def test_degree_mixing_dict_undirected(self):
7: (8)                      d = nx.degree_mixing_dict(self.P4)
8: (8)                      d_result = {1: {2: 2}, 2: {1: 2, 2: 2}}
9: (8)                      assert d == d_result
10: (4)                 def test_degree_mixing_dict_undirected_normalized(self):
11: (8)                     d = nx.degree_mixing_dict(self.P4, normalized=True)
12: (8)                     d_result = {1: {2: 1.0 / 3}, 2: {1: 1.0 / 3, 2: 1.0 / 3}}
13: (8)                     assert d == d_result
14: (4)                 def test_degree_mixing_dict_directed(self):
15: (8)                     d = nx.degree_mixing_dict(self.D)
16: (8)                     print(d)
17: (8)                     d_result = {1: {3: 2}, 2: {1: 1, 3: 1}, 3: {}}
18: (8)                     assert d == d_result
19: (4)                 def test_degree_mixing_dict_multigraph(self):
20: (8)                     d = nx.degree_mixing_dict(self.M)
21: (8)                     d_result = {1: {2: 1}, 2: {1: 1, 3: 3}, 3: {2: 3}}
22: (8)                     assert d == d_result
23: (4)                 def test_degree_mixing_dict_weighted(self):
24: (8)                     d = nx.degree_mixing_dict(self.W, weight="weight")
25: (8)                     d_result = {0.5: {1.5: 1}, 1.5: {1.5: 6, 0.5: 1}}
26: (8)                     assert d == d_result
27: (0)             class TestDegreeMixingMatrix(BaseTestDegreeMixing):
28: (4)                 def test_degree_mixing_matrix_undirected(self):
29: (8)                     a_result = np.array([[0, 2],
30: (29)                                          [2, 2]]
31: (28)                                         )
32: (8)                     a = nx.degree_mixing_matrix(self.P4, normalized=False)
33: (8)                     np.testing.assert_equal(a, a_result)
34: (8)                     a = nx.degree_mixing_matrix(self.P4)
35: (8)                     np.testing.assert_equal(a, a_result / a_result.sum())
36: (4)                 def test_degree_mixing_matrix_directed(self):
37: (8)                     a_result = np.array([[0, 0, 2],
38: (29)                                          [1, 0, 1],
39: (29)                                          [0, 0, 0]]
40: (28)                                         )
41: (8)                     a = nx.degree_mixing_matrix(self.D, normalized=False)
42: (8)                     np.testing.assert_equal(a, a_result)
43: (8)                     a = nx.degree_mixing_matrix(self.D)
44: (8)                     np.testing.assert_equal(a, a_result / a_result.sum())
45: (4)                 def test_degree_mixing_matrix_multigraph(self):
46: (8)                     a_result = np.array([[0, 1, 0],
47: (29)                                          [1, 0, 3],
48: (29)                                          [0, 3, 0]]
49: (28)                                         )
50: (8)                     a = nx.degree_mixing_matrix(self.M, normalized=False)
```

```
(8)                     np.testing.assert_equal(a, a_result)
(8)                     a = nx.degree_mixing_matrix(self.M)
(8)                     np.testing.assert_equal(a, a_result / a_result.sum())
(4)                 def test_degree_mixing_matrix_selfloop(self):
(8)                     a_result = np.array([[2]])
(8)                     a = nx.degree_mixing_matrix(self.S, normalized=False)
(8)                     np.testing.assert_equal(a, a_result)
(8)                     a = nx.degree_mixing_matrix(self.S)
(8)                     np.testing.assert_equal(a, a_result / a_result.sum())
(4)                 def test_degree_mixing_matrix_weighted(self):
(8)                     a_result = np.array([[0.0, 1.0], [1.0, 6.0]])
(8)                     a = nx.degree_mixing_matrix(self.W, weight="weight", normalized=False)
(8)                     np.testing.assert_equal(a, a_result)
(8)                     a = nx.degree_mixing_matrix(self.W, weight="weight")
(8)                     np.testing.assert_equal(a, a_result / float(a_result.sum()))
(4)                 def test_degree_mixing_matrix_mapping(self):
(8)                     a_result = np.array([[6.0, 1.0], [1.0, 0.0]])
(8)                     mapping = {0.5: 1, 1.5: 0}
(8)                     a = nx.degree_mixing_matrix(
(12)                        self.W, weight="weight", normalized=False, mapping=mapping
(8)                     )
(8)                     np.testing.assert_equal(a, a_result)
(0)             class TestAttributeMixingDict(BaseTestAttributeMixing):
(4)                 def test_attribute_mixing_dict_undirected(self):
(8)                     d = nx.attribute_mixing_dict(self.G, "fish")
(8)                     d_result = {
(12)                        "one": {"one": 2, "red": 1},
(12)                        "two": {"two": 2, "blue": 1},
(12)                        "red": {"one": 1},
(12)                        "blue": {"two": 1},
(8)                     }
(8)                     assert d == d_result
(4)                 def test_attribute_mixing_dict_directed(self):
(8)                     d = nx.attribute_mixing_dict(self.D, "fish")
(8)                     d_result = {
(12)                        "one": {"one": 1, "red": 1},
(12)                        "two": {"two": 1, "blue": 1},
(12)                        "red": {},
(12)                        "blue": {},
(8)                     }
(8)                     assert d == d_result
(4)                 def test_attribute_mixing_dict_multigraph(self):
(8)                     d = nx.attribute_mixing_dict(self.M, "fish")
(8)                     d_result = {"one": {"one": 4}, "two": {"two": 2}}
(8)                     assert d == d_result
(0)             class TestAttributeMixingMatrix(BaseTestAttributeMixing):
(4)                 def test_attribute_mixing_matrix_undirected(self):
(8)                     mapping = {"one": 0, "two": 1, "red": 2, "blue": 3}
(8)                     a_result = np.array([[2, 0, 1, 0], [0, 2, 0, 1], [1, 0, 0, 0], [0, 1,
0, 0]])
(8)                     a = nx.attribute_mixing_matrix(
(12)                        self.G, "fish", mapping=mapping, normalized=False
(8)                     )
(8)                     np.testing.assert_equal(a, a_result)
(8)                     a = nx.attribute_mixing_matrix(self.G, "fish", mapping=mapping)
(8)                     np.testing.assert_equal(a, a_result / a_result.sum())
(4)                 def test_attribute_mixing_matrix_directed(self):
(8)                     mapping = {"one": 0, "two": 1, "red": 2, "blue": 3}
(8)                     a_result = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 0, 0], [0, 0,
0, 0]])
(8)                     a = nx.attribute_mixing_matrix(
(12)                        self.D, "fish", mapping=mapping, normalized=False
(8)                     )
(8)                     np.testing.assert_equal(a, a_result)
(8)                     a = nx.attribute_mixing_matrix(self.D, "fish", mapping=mapping)
(8)                     np.testing.assert_equal(a, a_result / a_result.sum())
(4)                 def test_attribute_mixing_matrix_multigraph(self):
(8)                     mapping = {"one": 0, "two": 1, "red": 2, "blue": 3}
(8)                     a_result = np.array([[4, 0, 0, 0], [0, 2, 0, 0], [0, 0, 0, 0], [0, 0,
```

```
0, 0]])
118: (8)                    a = nx.attribute_mixing_matrix(
119: (12)                       self.M, "fish", mapping=mapping, normalized=False
120: (8)                    )
121: (8)                    np.testing.assert_equal(a, a_result)
122: (8)                    a = nx.attribute_mixing_matrix(self.M, "fish", mapping=mapping)
123: (8)                    np.testing.assert_equal(a, a_result / a_result.sum())
124: (4)                def test_attribute_mixing_matrix_negative(self):
125: (8)                    mapping = {-2: 0, -3: 1, -4: 2}
126: (8)                    a_result = np.array([[4.0, 1.0, 1.0], [1.0, 0.0, 0.0], [1.0, 0.0,
0.0]])
127: (8)                    a = nx.attribute_mixing_matrix(
128: (12)                       self.N, "margin", mapping=mapping, normalized=False
129: (8)                    )
130: (8)                    np.testing.assert_equal(a, a_result)
131: (8)                    a = nx.attribute_mixing_matrix(self.N, "margin", mapping=mapping)
132: (8)                    np.testing.assert_equal(a, a_result / float(a_result.sum()))
133: (4)                def test_attribute_mixing_matrix_float(self):
134: (8)                    mapping = {0.5: 1, 1.5: 0}
135: (8)                    a_result = np.array([[6.0, 1.0], [1.0, 0.0]])
136: (8)                    a = nx.attribute_mixing_matrix(
137: (12)                       self.F, "margin", mapping=mapping, normalized=False
138: (8)                    )
139: (8)                    np.testing.assert_equal(a, a_result)
140: (8)                    a = nx.attribute_mixing_matrix(self.F, "margin", mapping=mapping)
141: (8)                    np.testing.assert_equal(a, a_result / a_result.sum())

----------------------------------------

File 103 - test_neighbor_degree.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestAverageNeighbor:
4: (4)                  def test_degree_p4(self):
5: (8)                      G = nx.path_graph(4)
6: (8)                      answer = {0: 2, 1: 1.5, 2: 1.5, 3: 2}
7: (8)                      nd = nx.average_neighbor_degree(G)
8: (8)                      assert nd == answer
9: (8)                      D = G.to_directed()
10: (8)                     nd = nx.average_neighbor_degree(D)
11: (8)                     assert nd == answer
12: (8)                     D = nx.DiGraph(G.edges(data=True))
13: (8)                     nd = nx.average_neighbor_degree(D)
14: (8)                     assert nd == {0: 1, 1: 1, 2: 0, 3: 0}
15: (8)                     nd = nx.average_neighbor_degree(D, "in", "out")
16: (8)                     assert nd == {0: 0, 1: 1, 2: 1, 3: 1}
17: (8)                     nd = nx.average_neighbor_degree(D, "out", "in")
18: (8)                     assert nd == {0: 1, 1: 1, 2: 1, 3: 0}
19: (8)                     nd = nx.average_neighbor_degree(D, "in", "in")
20: (8)                     assert nd == {0: 0, 1: 0, 2: 1, 3: 1}
21: (4)                 def test_degree_p4_weighted(self):
22: (8)                     G = nx.path_graph(4)
23: (8)                     G[1][2]["weight"] = 4
24: (8)                     answer = {0: 2, 1: 1.8, 2: 1.8, 3: 2}
25: (8)                     nd = nx.average_neighbor_degree(G, weight="weight")
26: (8)                     assert nd == answer
27: (8)                     D = G.to_directed()
28: (8)                     nd = nx.average_neighbor_degree(D, weight="weight")
29: (8)                     assert nd == answer
30: (8)                     D = nx.DiGraph(G.edges(data=True))
31: (8)                     print(D.edges(data=True))
32: (8)                     nd = nx.average_neighbor_degree(D, weight="weight")
33: (8)                     assert nd == {0: 1, 1: 1, 2: 0, 3: 0}
34: (8)                     nd = nx.average_neighbor_degree(D, "out", "out", weight="weight")
35: (8)                     assert nd == {0: 1, 1: 1, 2: 0, 3: 0}
36: (8)                     nd = nx.average_neighbor_degree(D, "in", "in", weight="weight")
37: (8)                     assert nd == {0: 0, 1: 0, 2: 1, 3: 1}
38: (8)                     nd = nx.average_neighbor_degree(D, "in", "out", weight="weight")
```

```
39: (8)                     assert nd == {0: 0, 1: 1, 2: 1, 3: 1}
40: (8)                     nd = nx.average_neighbor_degree(D, "out", "in", weight="weight")
41: (8)                     assert nd == {0: 1, 1: 1, 2: 1, 3: 0}
42: (8)                     nd = nx.average_neighbor_degree(D, source="in+out", weight="weight")
43: (8)                     assert nd == {0: 1.0, 1: 1.0, 2: 0.8, 3: 1.0}
44: (8)                     nd = nx.average_neighbor_degree(D, target="in+out", weight="weight")
45: (8)                     assert nd == {0: 2.0, 1: 2.0, 2: 1.0, 3: 0.0}
46: (8)                     D = G.to_directed()
47: (8)                     nd = nx.average_neighbor_degree(D, weight="weight")
48: (8)                     assert nd == answer
49: (8)                     nd = nx.average_neighbor_degree(D, source="out", target="out",
weight="weight")
50: (8)                     assert nd == answer
51: (8)                     D = G.to_directed()
52: (8)                     nd = nx.average_neighbor_degree(D, source="in", target="in",
weight="weight")
53: (8)                     assert nd == answer
54: (4)                 def test_degree_k4(self):
55: (8)                     G = nx.complete_graph(4)
56: (8)                     answer = {0: 3, 1: 3, 2: 3, 3: 3}
57: (8)                     nd = nx.average_neighbor_degree(G)
58: (8)                     assert nd == answer
59: (8)                     D = G.to_directed()
60: (8)                     nd = nx.average_neighbor_degree(D)
61: (8)                     assert nd == answer
62: (8)                     D = G.to_directed()
63: (8)                     nd = nx.average_neighbor_degree(D)
64: (8)                     assert nd == answer
65: (8)                     D = G.to_directed()
66: (8)                     nd = nx.average_neighbor_degree(D, source="in", target="in")
67: (8)                     assert nd == answer
68: (4)                 def test_degree_k4_nodes(self):
69: (8)                     G = nx.complete_graph(4)
70: (8)                     answer = {1: 3.0, 2: 3.0}
71: (8)                     nd = nx.average_neighbor_degree(G, nodes=[1, 2])
72: (8)                     assert nd == answer
73: (4)                 def test_degree_barrat(self):
74: (8)                     G = nx.star_graph(5)
75: (8)                     G.add_edges_from([(5, 6), (5, 7), (5, 8), (5, 9)])
76: (8)                     G[0][5]["weight"] = 5
77: (8)                     nd = nx.average_neighbor_degree(G)[5]
78: (8)                     assert nd == 1.8
79: (8)                     nd = nx.average_neighbor_degree(G, weight="weight")[5]
80: (8)                     assert nd == pytest.approx(3.222222, abs=1e-5)
81: (4)                 def test_error_invalid_source_target(self):
82: (8)                     G = nx.path_graph(4)
83: (8)                     with pytest.raises(nx.NetworkXError):
84: (12)                        nx.average_neighbor_degree(G, "error")
85: (8)                     with pytest.raises(nx.NetworkXError):
86: (12)                        nx.average_neighbor_degree(G, "in", "error")
87: (8)                     G = G.to_directed()
88: (8)                     with pytest.raises(nx.NetworkXError):
89: (12)                        nx.average_neighbor_degree(G, "error")
90: (8)                     with pytest.raises(nx.NetworkXError):
91: (12)                        nx.average_neighbor_degree(G, "in", "error")
```

File 104 - test_pairs.py:

```
1: (0)              import networkx as nx
2: (0)              from .base_test import BaseTestAttributeMixing, BaseTestDegreeMixing
3: (0)              class TestAttributeMixingXY(BaseTestAttributeMixing):
4: (4)                  def test_node_attribute_xy_undirected(self):
5: (8)                      attrxy = sorted(nx.node_attribute_xy(self.G, "fish"))
6: (8)                      attrxy_result = sorted(
7: (12)                         [
8: (16)                             ("one", "one"),
9: (16)                             ("one", "one"),
```

```
 10: (16)                            ("two", "two"),
 11: (16)                            ("two", "two"),
 12: (16)                            ("one", "red"),
 13: (16)                            ("red", "one"),
 14: (16)                            ("blue", "two"),
 15: (16)                            ("two", "blue"),
 16: (12)                        ]
 17: (8)                     )
 18: (8)                     assert attrxy == attrxy_result
 19: (4)                 def test_node_attribute_xy_undirected_nodes(self):
 20: (8)                     attrxy = sorted(nx.node_attribute_xy(self.G, "fish", nodes=["one",
 "yellow"]))
 21: (8)                     attrxy_result = sorted([])
 22: (8)                     assert attrxy == attrxy_result
 23: (4)                 def test_node_attribute_xy_directed(self):
 24: (8)                     attrxy = sorted(nx.node_attribute_xy(self.D, "fish"))
 25: (8)                     attrxy_result = sorted(
 26: (12)                        [("one", "one"), ("two", "two"), ("one", "red"), ("two", "blue")]
 27: (8)                     )
 28: (8)                     assert attrxy == attrxy_result
 29: (4)                 def test_node_attribute_xy_multigraph(self):
 30: (8)                     attrxy = sorted(nx.node_attribute_xy(self.M, "fish"))
 31: (8)                     attrxy_result = [
 32: (12)                        ("one", "one"),
 33: (12)                        ("one", "one"),
 34: (12)                        ("one", "one"),
 35: (12)                        ("one", "one"),
 36: (12)                        ("two", "two"),
 37: (12)                        ("two", "two"),
 38: (8)                     ]
 39: (8)                     assert attrxy == attrxy_result
 40: (4)                 def test_node_attribute_xy_selfloop(self):
 41: (8)                     attrxy = sorted(nx.node_attribute_xy(self.S, "fish"))
 42: (8)                     attrxy_result = [("one", "one"), ("two", "two")]
 43: (8)                     assert attrxy == attrxy_result
 44: (0)             class TestDegreeMixingXY(BaseTestDegreeMixing):
 45: (4)                 def test_node_degree_xy_undirected(self):
 46: (8)                     xy = sorted(nx.node_degree_xy(self.P4))
 47: (8)                     xy_result = sorted([(1, 2), (2, 1), (2, 2), (2, 2), (1, 2), (2, 1)])
 48: (8)                     assert xy == xy_result
 49: (4)                 def test_node_degree_xy_undirected_nodes(self):
 50: (8)                     xy = sorted(nx.node_degree_xy(self.P4, nodes=[0, 1, -1]))
 51: (8)                     xy_result = sorted([(1, 2), (2, 1)])
 52: (8)                     assert xy == xy_result
 53: (4)                 def test_node_degree_xy_directed(self):
 54: (8)                     xy = sorted(nx.node_degree_xy(self.D))
 55: (8)                     xy_result = sorted([(2, 1), (2, 3), (1, 3), (1, 3)])
 56: (8)                     assert xy == xy_result
 57: (4)                 def test_node_degree_xy_multigraph(self):
 58: (8)                     xy = sorted(nx.node_degree_xy(self.M))
 59: (8)                     xy_result = sorted(
 60: (12)                        [(2, 3), (2, 3), (3, 2), (3, 2), (2, 3), (3, 2), (1, 2), (2, 1)]
 61: (8)                     )
 62: (8)                     assert xy == xy_result
 63: (4)                 def test_node_degree_xy_selfloop(self):
 64: (8)                     xy = sorted(nx.node_degree_xy(self.S))
 65: (8)                     xy_result = sorted([(2, 2), (2, 2)])
 66: (8)                     assert xy == xy_result
 67: (4)                 def test_node_degree_xy_weighted(self):
 68: (8)                     G = nx.Graph()
 69: (8)                     G.add_edge(1, 2, weight=7)
 70: (8)                     G.add_edge(2, 3, weight=10)
 71: (8)                     xy = sorted(nx.node_degree_xy(G, weight="weight"))
 72: (8)                     xy_result = sorted([(7, 17), (17, 10), (17, 7), (10, 17)])
 73: (8)                     assert xy == xy_result

 ----------------------------------------

 File 105 - __init__.py:
```

```
r""" This module provides functions and operations for bipartite
graphs.  Bipartite graphs `B = (U, V, E)` have two node sets `U,V` and edges in
`E` that only connect nodes from opposite sets. It is common in the literature
to use an spatial analogy referring to the two node sets as top and bottom nodes.
The bipartite algorithms are not imported into the networkx namespace
at the top level so the easiest way to use them is with:
>>> from networkx.algorithms import bipartite
NetworkX does not have a custom bipartite graph class but the Graph()
or DiGraph() classes can be used to represent bipartite graphs. However,
you have to keep track of which set each node belongs to, and make
sure that there is no edge between nodes of the same set. The convention used
in NetworkX is to use a node attribute named `bipartite` with values 0 or 1 to
identify the sets each node belongs to. This convention is not enforced in
the source code of bipartite functions, it's only a recommendation.
For example:
>>> B = nx.Graph()
>>> # Add nodes with the node attribute "bipartite"
>>> B.add_nodes_from([1, 2, 3, 4], bipartite=0)
>>> B.add_nodes_from(["a", "b", "c"], bipartite=1)
>>> # Add edges only between nodes of opposite node sets
>>> B.add_edges_from([(1, "a"), (1, "b"), (2, "b"), (2, "c"), (3, "c"), (4, "a")])
Many algorithms of the bipartite module of NetworkX require, as an argument, a
container with all the nodes that belong to one set, in addition to the bipartite
graph `B`. The functions in the bipartite package do not check that the node set
is actually correct nor that the input graph is actually bipartite.
If `B` is connected, you can find the two node sets using a two-coloring
algorithm:
>>> nx.is_connected(B)
True
>>> bottom_nodes, top_nodes = bipartite.sets(B)
However, if the input graph is not connected, there are more than one possible
colorations. This is the reason why we require the user to pass a container
with all nodes of one bipartite node set as an argument to most bipartite
functions. In the face of ambiguity, we refuse the temptation to guess and
raise an :exc:`AmbiguousSolution <networkx.AmbiguousSolution>`
Exception if the input graph for
:func:`bipartite.sets <networkx.algorithms.bipartite.basic.sets>`
is disconnected.
Using the `bipartite` node attribute, you can easily get the two node sets:
>>> top_nodes = {n for n, d in B.nodes(data=True) if d["bipartite"] == 0}
>>> bottom_nodes = set(B) - top_nodes
So you can easily use the bipartite algorithms that require, as an argument, a
container with all nodes that belong to one node set:
>>> print(round(bipartite.density(B, bottom_nodes), 2))
0.5
>>> G = bipartite.projected_graph(B, top_nodes)
All bipartite graph generators in NetworkX build bipartite graphs with the
`bipartite` node attribute. Thus, you can use the same approach:
>>> RB = bipartite.random_graph(5, 7, 0.2)
>>> RB_top = {n for n, d in RB.nodes(data=True) if d["bipartite"] == 0}
>>> RB_bottom = set(RB) - RB_top
>>> list(RB_top)
[0, 1, 2, 3, 4]
>>> list(RB_bottom)
[5, 6, 7, 8, 9, 10, 11]
For other bipartite graph generators see
:mod:`Generators <networkx.algorithms.bipartite.generators>`.
"""
from networkx.algorithms.bipartite.basic import *
from networkx.algorithms.bipartite.centrality import *
from networkx.algorithms.bipartite.cluster import *
from networkx.algorithms.bipartite.covering import *
from networkx.algorithms.bipartite.edgelist import *
```

```
64: (0)             from networkx.algorithms.bipartite.matching import *
65: (0)             from networkx.algorithms.bipartite.matrix import *
66: (0)             from networkx.algorithms.bipartite.projection import *
67: (0)             from networkx.algorithms.bipartite.redundancy import *
68: (0)             from networkx.algorithms.bipartite.spectral import *
69: (0)             from networkx.algorithms.bipartite.generators import *
70: (0)             from networkx.algorithms.bipartite.extendability import *

----------------------------------------

File 106 - basic.py:

1: (0)              """
2: (0)              ==========================
3: (0)              Bipartite Graph Algorithms
4: (0)              ==========================
5: (0)              """
6: (0)              import networkx as nx
7: (0)              from networkx.algorithms.components import connected_components
8: (0)              from networkx.exception import AmbiguousSolution
9: (0)              __all__ = [
10: (4)                 "is_bipartite",
11: (4)                 "is_bipartite_node_set",
12: (4)                 "color",
13: (4)                 "sets",
14: (4)                 "density",
15: (4)                 "degrees",
16: (0)             ]
17: (0)             @nx._dispatchable
18: (0)             def color(G):
19: (4)                 """Returns a two-coloring of the graph.
20: (4)                 Raises an exception if the graph is not bipartite.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : NetworkX graph
24: (4)                 Returns
25: (4)                 -------
26: (4)                 color : dictionary
27: (8)                     A dictionary keyed by node with a 1 or 0 as data for each node color.
28: (4)                 Raises
29: (4)                 ------
30: (4)                 NetworkXError
31: (8)                     If the graph is not two-colorable.
32: (4)                 Examples
33: (4)                 --------
34: (4)                 >>> from networkx.algorithms import bipartite
35: (4)                 >>> G = nx.path_graph(4)
36: (4)                 >>> c = bipartite.color(G)
37: (4)                 >>> print(c)
38: (4)                 {0: 1, 1: 0, 2: 1, 3: 0}
39: (4)                 You can use this to set a node attribute indicating the bipartite set:
40: (4)                 >>> nx.set_node_attributes(G, c, "bipartite")
41: (4)                 >>> print(G.nodes[0]["bipartite"])
42: (4)                 1
43: (4)                 >>> print(G.nodes[1]["bipartite"])
44: (4)                 0
45: (4)                 """
46: (4)                 if G.is_directed():
47: (8)                     import itertools
48: (8)                     def neighbors(v):
49: (12)                        return itertools.chain.from_iterable([G.predecessors(v),
G.successors(v)])
50: (4)                 else:
51: (8)                     neighbors = G.neighbors
52: (4)                 color = {}
53: (4)                 for n in G:  # handle disconnected graphs
54: (8)                     if n in color or len(G[n]) == 0:  # skip isolates
55: (12)                        continue
56: (8)                     queue = [n]
```

```
57: (8)                     color[n] = 1  # nodes seen with color (1 or 0)
58: (8)                     while queue:
59: (12)                        v = queue.pop()
60: (12)                        c = 1 - color[v]  # opposite color of node v
61: (12)                        for w in neighbors(v):
62: (16)                            if w in color:
63: (20)                                if color[w] == color[v]:
64: (24)                                    raise nx.NetworkXError("Graph is not bipartite.")
65: (16)                            else:
66: (20)                                color[w] = c
67: (20)                                queue.append(w)
68: (4)                 color.update(dict.fromkeys(nx.isolates(G), 0))
69: (4)                 return color
70: (0)             @nx._dispatchable
71: (0)             def is_bipartite(G):
72: (4)                 """Returns True if graph G is bipartite, False if not.
73: (4)                 Parameters
74: (4)                 ----------
75: (4)                 G : NetworkX graph
76: (4)                 Examples
77: (4)                 --------
78: (4)                 >>> from networkx.algorithms import bipartite
79: (4)                 >>> G = nx.path_graph(4)
80: (4)                 >>> print(bipartite.is_bipartite(G))
81: (4)                 True
82: (4)                 See Also
83: (4)                 --------
84: (4)                 color, is_bipartite_node_set
85: (4)                 """
86: (4)                 try:
87: (8)                     color(G)
88: (8)                     return True
89: (4)                 except nx.NetworkXError:
90: (8)                     return False
91: (0)             @nx._dispatchable
92: (0)             def is_bipartite_node_set(G, nodes):
93: (4)                 """Returns True if nodes and G/nodes are a bipartition of G.
94: (4)                 Parameters
95: (4)                 ----------
96: (4)                 G : NetworkX graph
97: (4)                 nodes: list or container
98: (6)                   Check if nodes are a one of a bipartite set.
99: (4)                 Examples
100: (4)                --------
101: (4)                >>> from networkx.algorithms import bipartite
102: (4)                >>> G = nx.path_graph(4)
103: (4)                >>> X = set([1, 3])
104: (4)                >>> bipartite.is_bipartite_node_set(G, X)
105: (4)                True
106: (4)                Notes
107: (4)                -----
108: (4)                An exception is raised if the input nodes are not distinct, because in
this
109: (4)                case some bipartite algorithms will yield incorrect results.
110: (4)                For connected graphs the bipartite sets are unique.  This function handles
111: (4)                disconnected graphs.
112: (4)                """
113: (4)                S = set(nodes)
114: (4)                if len(S) < len(nodes):
115: (8)                    raise AmbiguousSolution(
116: (12)                       "The input node set contains duplicates.\n"
117: (12)                       "This may lead to incorrect results when using it in bipartite
algorithms.\n"
118: (12)                       "Consider using set(nodes) as the input"
119: (8)                    )
120: (4)                for CC in (G.subgraph(c).copy() for c in connected_components(G)):
121: (8)                    X, Y = sets(CC)
122: (8)                    if not (
123: (12)                       (X.issubset(S) and Y.isdisjoint(S)) or (Y.issubset(S) and
```

```
X.isdisjoint(S))
    ):
        return False
    return True
@nx._dispatchable
def sets(G, top_nodes=None):
    """Returns bipartite node sets of graph G.
    Raises an exception if the graph is not bipartite or if the input
    graph is disconnected and thus more than one valid solution exists.
    See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
    for further details on how bipartite graphs are handled in NetworkX.
    Parameters
    ----------
    G : NetworkX graph
    top_nodes : container, optional
      Container with all nodes in one bipartite node set. If not supplied
      it will be computed. But if more than one solution exists an exception
      will be raised.
    Returns
    -------
    X : set
      Nodes from one side of the bipartite graph.
    Y : set
      Nodes from the other side.
    Raises
    ------
    AmbiguousSolution
      Raised if the input bipartite graph is disconnected and no container
      with all nodes in one bipartite set is provided. When determining
      the nodes in each bipartite set more than one valid solution is
      possible if the input graph is disconnected.
    NetworkXError
      Raised if the input graph is not bipartite.
    Examples
    --------
    >>> from networkx.algorithms import bipartite
    >>> G = nx.path_graph(4)
    >>> X, Y = bipartite.sets(G)
    >>> list(X)
    [0, 2]
    >>> list(Y)
    [1, 3]
    See Also
    --------
    color
    """
    if G.is_directed():
        is_connected = nx.is_weakly_connected
    else:
        is_connected = nx.is_connected
    if top_nodes is not None:
        X = set(top_nodes)
        Y = set(G) - X
    else:
        if not is_connected(G):
            msg = "Disconnected graph: Ambiguous solution for bipartite sets."
            raise nx.AmbiguousSolution(msg)
        c = color(G)
        X = {n for n, is_top in c.items() if is_top}
        Y = {n for n, is_top in c.items() if not is_top}
    return (X, Y)
@nx._dispatchable(graphs="B")
def density(B, nodes):
    """Returns density of bipartite graph B.
    Parameters
    ----------
    B : NetworkX graph
    nodes: list or container
      Nodes in one node set of the bipartite graph.
```

```
(4)                 Returns
(4)                 -------
(4)                 d : float
(7)                    The bipartite density
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> G = nx.complete_bipartite_graph(3, 2)
(4)                 >>> X = set([0, 1, 2])
(4)                 >>> bipartite.density(G, X)
(4)                 1.0
(4)                 >>> Y = set([3, 4])
(4)                 >>> bipartite.density(G, Y)
(4)                 1.0
(4)                 Notes
(4)                 -----
(4)                 The container of nodes passed as argument must contain all nodes
(4)                 in one of the two bipartite node sets to avoid ambiguity in the
(4)                 case of disconnected graphs.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 See Also
(4)                 --------
(4)                 color
(4)                 """
(4)                 n = len(B)
(4)                 m = nx.number_of_edges(B)
(4)                 nb = len(nodes)
(4)                 nt = n - nb
(4)                 if m == 0:  # includes cases n==0 and n==1
(8)                     d = 0.0
(4)                 else:
(8)                     if B.is_directed():
(12)                        d = m / (2 * nb * nt)
(8)                     else:
(12)                        d = m / (nb * nt)
(4)                 return d
(0)             @nx._dispatchable(graphs="B", edge_attrs="weight")
(0)             def degrees(B, nodes, weight=None):
(4)                 """Returns the degrees of the two node sets in the bipartite graph B.
(4)                 Parameters
(4)                 ----------
(4)                 B : NetworkX graph
(4)                 nodes: list or container
(6)                   Nodes in one node set of the bipartite graph.
(4)                 weight : string or None, optional (default=None)
(7)                    The edge attribute that holds the numerical value used as a weight.
(7)                    If None, then each edge has weight 1.
(7)                    The degree is the sum of the edge weights adjacent to the node.
(4)                 Returns
(4)                 -------
(4)                 (degX,degY) : tuple of dictionaries
(7)                    The degrees of the two bipartite sets as dictionaries keyed by node.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> G = nx.complete_bipartite_graph(3, 2)
(4)                 >>> Y = set([3, 4])
(4)                 >>> degX, degY = bipartite.degrees(G, Y)
(4)                 >>> dict(degX)
(4)                 {0: 2, 1: 2, 2: 2}
(4)                 Notes
(4)                 -----
(4)                 The container of nodes passed as argument must contain all nodes
(4)                 in one of the two bipartite node sets to avoid ambiguity in the
(4)                 case of disconnected graphs.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 See Also
```

```
261: (4)                --------
262: (4)                color, density
263: (4)                """
264: (4)                bottom = set(nodes)
265: (4)                top = set(B) - bottom
266: (4)                return (B.degree(top, weight), B.degree(bottom, weight))
```

File 107 - centrality.py:

```
1: (0)              import networkx as nx
2: (0)              __all__ = ["degree_centrality", "betweenness_centrality",
"closeness_centrality"]
3: (0)              @nx._dispatchable(name="bipartite_degree_centrality")
4: (0)              def degree_centrality(G, nodes):
5: (4)                  r"""Compute the degree centrality for nodes in a bipartite network.
6: (4)                  The degree centrality for a node `v` is the fraction of nodes
7: (4)                  connected to it.
8: (4)                  Parameters
9: (4)                  ----------
10: (4)                 G : graph
11: (7)                    A bipartite network
12: (4)                 nodes : list or container
13: (6)                   Container with all nodes in one bipartite node set.
14: (4)                 Returns
15: (4)                 -------
16: (4)                 centrality : dictionary
17: (7)                    Dictionary keyed by node with bipartite degree centrality as the value.
18: (4)                 Examples
19: (4)                 --------
20: (4)                 >>> G = nx.wheel_graph(5)
21: (4)                 >>> top_nodes = {0, 1, 2}
22: (4)                 >>> nx.bipartite.degree_centrality(G, nodes=top_nodes)
23: (4)                 {0: 2.0, 1: 1.5, 2: 1.5, 3: 1.0, 4: 1.0}
24: (4)                 See Also
25: (4)                 --------
26: (4)                 betweenness_centrality
27: (4)                 closeness_centrality
28: (4)                 :func:`~networkx.algorithms.bipartite.basic.sets`
29: (4)                 :func:`~networkx.algorithms.bipartite.basic.is_bipartite`
30: (4)                 Notes
31: (4)                 -----
32: (4)                 The nodes input parameter must contain all nodes in one bipartite node
set,
33: (4)                 but the dictionary returned contains all nodes from both bipartite node
34: (4)                 sets. See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
35: (4)                 for further details on how bipartite graphs are handled in NetworkX.
36: (4)                 For unipartite networks, the degree centrality values are
37: (4)                 normalized by dividing by the maximum possible degree (which is
38: (4)                 `n-1` where `n` is the number of nodes in G).
39: (4)                 In the bipartite case, the maximum possible degree of a node in a
40: (4)                 bipartite node set is the number of nodes in the opposite node set
41: (4)                 [1]_.  The degree centrality for a node `v` in the bipartite
42: (4)                 sets `U` with `n` nodes and `V` with `m` nodes is
43: (4)                 .. math::
44: (8)                     d_{v} = \frac{deg(v)}{m}, \mbox{for} v \in U ,
45: (8)                     d_{v} = \frac{deg(v)}{n}, \mbox{for} v \in V ,
46: (4)                 where `deg(v)` is the degree of node `v`.
47: (4)                 References
48: (4)                 ----------
49: (4)                 .. [1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation
50: (8)                     Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook
51: (8)                     of Social Network Analysis. Sage Publications.
52: (8)                     https://dx.doi.org/10.4135/9781446294413.n28
53: (4)                 """
54: (4)                 top = set(nodes)
55: (4)                 bottom = set(G) - top
56: (4)                 s = 1.0 / len(bottom)
```

```
(4)                 centrality = {n: d * s for n, d in G.degree(top)}
(4)                 s = 1.0 / len(top)
(4)                 centrality.update({n: d * s for n, d in G.degree(bottom)})
(4)                 return centrality
(0)             @nx._dispatchable(name="bipartite_betweenness_centrality")
(0)             def betweenness_centrality(G, nodes):
(4)                 r"""Compute betweenness centrality for nodes in a bipartite network.
(4)                 Betweenness centrality of a node `v` is the sum of the
(4)                 fraction of all-pairs shortest paths that pass through `v`.
(4)                 Values of betweenness are normalized by the maximum possible
(4)                 value which for bipartite graphs is limited by the relative size
(4)                 of the two node sets [1]_.
(4)                 Let `n` be the number of nodes in the node set `U` and
(4)                 `m` be the number of nodes in the node set `V`, then
(4)                 nodes in `U` are normalized by dividing by
(4)                 .. math::
(7)                    \frac{1}{2} [m^2 (s + 1)^2 + m (s + 1)(2t - s - 1) - t (2s - t + 3)] ,
(4)                 where
(4)                 .. math::
(8)                     s = (n - 1) \div m , t = (n - 1) \mod m ,
(4)                 and nodes in `V` are normalized by dividing by
(4)                 .. math::
(8)                     \frac{1}{2} [n^2 (p + 1)^2 + n (p + 1)(2r - p - 1) - r (2p - r + 3)] ,
(4)                 where,
(4)                 .. math::
(8)                     p = (m - 1) \div n , r = (m - 1) \mod n .
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A bipartite graph
(4)                 nodes : list or container
(8)                     Container with all nodes in one bipartite node set.
(4)                 Returns
(4)                 -------
(4)                 betweenness : dictionary
(8)                     Dictionary keyed by node with bipartite betweenness centrality
(8)                     as the value.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.cycle_graph(4)
(4)                 >>> top_nodes = {1, 2}
(4)                 >>> nx.bipartite.betweenness_centrality(G, nodes=top_nodes)
(4)                 {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
(4)                 See Also
(4)                 --------
(4)                 degree_centrality
(4)                 closeness_centrality
(4)                 :func:`~networkx.algorithms.bipartite.basic.sets`
(4)                 :func:`~networkx.algorithms.bipartite.basic.is_bipartite`
(4)                 Notes
(4)                 -----
(4)                 The nodes input parameter must contain all nodes in one bipartite node
set,
(4)                 but the dictionary returned contains all nodes from both node sets.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 References
(4)                 ----------
(4)                 .. [1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation
(8)                     Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook
(8)                     of Social Network Analysis. Sage Publications.
(8)                     https://dx.doi.org/10.4135/9781446294413.n28
(4)                 """
(4)                 top = set(nodes)
(4)                 bottom = set(G) - top
(4)                 n = len(top)
(4)                 m = len(bottom)
(4)                 s, t = divmod(n - 1, m)
(4)                 bet_max_top = (
```

```
(8)                 ((m**2) * ((s + 1) ** 2))
(8)                 + (m * (s + 1) * (2 * t - s - 1))
(8)                 - (t * ((2 * s) - t + 3))
(4)             ) / 2.0
(4)             p, r = divmod(m - 1, n)
(4)             bet_max_bot = (
(8)                 ((n**2) * ((p + 1) ** 2))
(8)                 + (n * (p + 1) * (2 * r - p - 1))
(8)                 - (r * ((2 * p) - r + 3))
(4)             ) / 2.0
(4)             betweenness = nx.betweenness_centrality(G, normalized=False, weight=None)
(4)             for node in top:
(8)                 betweenness[node] /= bet_max_top
(4)             for node in bottom:
(8)                 betweenness[node] /= bet_max_bot
(4)             return betweenness
(0)         @nx._dispatchable(name="bipartite_closeness_centrality")
(0)         def closeness_centrality(G, nodes, normalized=True):
(4)             r"""Compute the closeness centrality for nodes in a bipartite network.
(4)             The closeness of a node is the distance to all other nodes in the
(4)             graph or in the case that the graph is not connected to all other nodes
(4)             in the connected component containing that node.
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A bipartite network
(4)             nodes : list or container
(8)                 Container with all nodes in one bipartite node set.
(4)             normalized : bool, optional
(6)               If True (default) normalize by connected component size.
(4)             Returns
(4)             -------
(4)             closeness : dictionary
(8)                 Dictionary keyed by node with bipartite closeness centrality
(8)                 as the value.
(4)             Examples
(4)             --------
(4)             >>> G = nx.wheel_graph(5)
(4)             >>> top_nodes = {0, 1, 2}
(4)             >>> nx.bipartite.closeness_centrality(G, nodes=top_nodes)
(4)             {0: 1.5, 1: 1.2, 2: 1.2, 3: 1.0, 4: 1.0}
(4)             See Also
(4)             --------
(4)             betweenness_centrality
(4)             degree_centrality
(4)             :func:`~networkx.algorithms.bipartite.basic.sets`
(4)             :func:`~networkx.algorithms.bipartite.basic.is_bipartite`
(4)             Notes
(4)             -----
(4)             The nodes input parameter must contain all nodes in one bipartite node
set,
(4)             but the dictionary returned contains all nodes from both node sets.
(4)             See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)             for further details on how bipartite graphs are handled in NetworkX.
(4)             Closeness centrality is normalized by the minimum distance possible.
(4)             In the bipartite case the minimum distance for a node in one bipartite
(4)             node set is 1 from all nodes in the other node set and 2 from all
(4)             other nodes in its own set [1]_. Thus the closeness centrality
(4)             for node `v`  in the two bipartite sets `U` with
(4)             `n` nodes and `V` with `m` nodes is
(4)             .. math::
(8)                 c_{v} = \frac{m + 2(n - 1)}{d}, \mbox{for} v \in U,
(8)                 c_{v} = \frac{n + 2(m - 1)}{d}, \mbox{for} v \in V,
(4)             where `d` is the sum of the distances from `v` to all
(4)             other nodes.
(4)             Higher values of closeness  indicate higher centrality.
(4)             As in the unipartite case, setting normalized=True causes the
(4)             values to normalized further to n-1 / size(G)-1 where n is the
(4)             number of nodes in the connected part of graph containing the
```

```
193: (4)                node.  If the graph is not completely connected, this algorithm
194: (4)                computes the closeness centrality for each connected part
195: (4)                separately.
196: (4)                References
197: (4)                ----------
198: (4)                .. [1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation
199: (8)                    Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook
200: (8)                    of Social Network Analysis. Sage Publications.
201: (8)                    https://dx.doi.org/10.4135/9781446294413.n28
202: (4)                """
203: (4)                closeness = {}
204: (4)                path_length = nx.single_source_shortest_path_length
205: (4)                top = set(nodes)
206: (4)                bottom = set(G) - top
207: (4)                n = len(top)
208: (4)                m = len(bottom)
209: (4)                for node in top:
210: (8)                    sp = dict(path_length(G, node))
211: (8)                    totsp = sum(sp.values())
212: (8)                    if totsp > 0.0 and len(G) > 1:
213: (12)                       closeness[node] = (m + 2 * (n - 1)) / totsp
214: (12)                       if normalized:
215: (16)                           s = (len(sp) - 1) / (len(G) - 1)
216: (16)                           closeness[node] *= s
217: (8)                    else:
218: (12)                       closeness[node] = 0.0
219: (4)                for node in bottom:
220: (8)                    sp = dict(path_length(G, node))
221: (8)                    totsp = sum(sp.values())
222: (8)                    if totsp > 0.0 and len(G) > 1:
223: (12)                       closeness[node] = (n + 2 * (m - 1)) / totsp
224: (12)                       if normalized:
225: (16)                           s = (len(sp) - 1) / (len(G) - 1)
226: (16)                           closeness[node] *= s
227: (8)                    else:
228: (12)                       closeness[node] = 0.0
229: (4)                return closeness
```

File 108 - cluster.py:

```
1: (0)              """Functions for computing clustering of pairs
2: (0)              """
3: (0)              import itertools
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "clustering",
7: (4)                  "average_clustering",
8: (4)                  "latapy_clustering",
9: (4)                  "robins_alexander_clustering",
10: (0)             ]
11: (0)             def cc_dot(nu, nv):
12: (4)                 return len(nu & nv) / len(nu | nv)
13: (0)             def cc_max(nu, nv):
14: (4)                 return len(nu & nv) / max(len(nu), len(nv))
15: (0)             def cc_min(nu, nv):
16: (4)                 return len(nu & nv) / min(len(nu), len(nv))
17: (0)             modes = {"dot": cc_dot, "min": cc_min, "max": cc_max}
18: (0)             @nx._dispatchable
19: (0)             def latapy_clustering(G, nodes=None, mode="dot"):
20: (4)                 r"""Compute a bipartite clustering coefficient for nodes.
21: (4)                 The bipartite clustering coefficient is a measure of local density
22: (4)                 of connections defined as [1]_:
23: (4)                 .. math::
24: (7)                    c_u = \frac{\sum_{v \in N(N(u))} c_{uv} }{|N(N(u))|}
25: (4)                 where `N(N(u))` are the second order neighbors of `u` in `G` excluding
`u`,
26: (4)                 and `c_{uv}` is the pairwise clustering coefficient between nodes
```

```
(4)                 `u` and `v`.
(4)                 The mode selects the function for `c_{uv}` which can be:
(4)                 `dot`:
(4)                 .. math::
(7)                    c_{uv}=\frac{|N(u)\cap N(v)|}{|N(u) \cup N(v)|}
(4)                 `min`:
(4)                 .. math::
(7)                    c_{uv}=\frac{|N(u)\cap N(v)|}{min(|N(u)|,|N(v)|)}
(4)                 `max`:
(4)                 .. math::
(7)                    c_{uv}=\frac{|N(u)\cap N(v)|}{max(|N(u)|,|N(v)|)}
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A bipartite graph
(4)                 nodes : list or iterable (optional)
(8)                     Compute bipartite clustering for these nodes. The default
(8)                     is all nodes in G.
(4)                 mode : string
(8)                     The pairwise bipartite clustering method to be used in the
computation.
(8)                     It must be "dot", "max", or "min".
(4)                 Returns
(4)                 -------
(4)                 clustering : dictionary
(8)                     A dictionary keyed by node with the clustering coefficient value.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> G = nx.path_graph(4)  # path graphs are bipartite
(4)                 >>> c = bipartite.clustering(G)
(4)                 >>> c[0]
(4)                 0.5
(4)                 >>> c = bipartite.clustering(G, mode="min")
(4)                 >>> c[0]
(4)                 1.0
(4)                 See Also
(4)                 --------
(4)                 robins_alexander_clustering
(4)                 average_clustering
(4)                 networkx.algorithms.cluster.square_clustering
(4)                 References
(4)                 ----------
(4)                 .. [1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio
(2008).
(7)                    Basic notions for the analysis of large two-mode networks.
(7)                    Social Networks 30(1), 31--48.
(4)                 """
(4)                 if not nx.algorithms.bipartite.is_bipartite(G):
(8)                     raise nx.NetworkXError("Graph is not bipartite")
(4)                 try:
(8)                     cc_func = modes[mode]
(4)                 except KeyError as err:
(8)                     raise nx.NetworkXError(
(12)                        "Mode for bipartite clustering must be: dot, min or max"
(8)                     ) from err
(4)                 if nodes is None:
(8)                     nodes = G
(4)                 ccs = {}
(4)                 for v in nodes:
(8)                     cc = 0.0
(8)                     nbrs2 = {u for nbr in G[v] for u in G[nbr]} - {v}
(8)                     for u in nbrs2:
(12)                        cc += cc_func(set(G[u]), set(G[v]))
(8)                     if cc > 0.0:  # len(nbrs2)>0
(12)                        cc /= len(nbrs2)
(8)                     ccs[v] = cc
(4)                 return ccs
(0)             clustering = latapy_clustering
```

```
(0)             @nx._dispatchable(name="bipartite_average_clustering")
(0)             def average_clustering(G, nodes=None, mode="dot"):
(4)                 r"""Compute the average bipartite clustering coefficient.
(4)                 A clustering coefficient for the whole graph is the average,
(4)                 .. math::
(7)                    C = \frac{1}{n}\sum_{v \in G} c_v,
(4)                 where `n` is the number of nodes in `G`.
(4)                 Similar measures for the two bipartite sets can be defined [1]_
(4)                 .. math::
(7)                    C_X = \frac{1}{|X|}\sum_{v \in X} c_v,
(4)                 where `X` is a bipartite set of `G`.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     a bipartite graph
(4)                 nodes : list or iterable, optional
(8)                     A container of nodes to use in computing the average.
(8)                     The nodes should be either the entire graph (the default) or one of the
(8)                     bipartite sets.
(4)                 mode : string
(8)                     The pairwise bipartite clustering method.
(8)                     It must be "dot", "max", or "min"
(4)                 Returns
(4)                 -------
(4)                 clustering : float
(7)                    The average bipartite clustering for the given set of nodes or the
(7)                    entire graph if no nodes are specified.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> G = nx.star_graph(3)  # star graphs are bipartite
(4)                 >>> bipartite.average_clustering(G)
(4)                 0.75
(4)                 >>> X, Y = bipartite.sets(G)
(4)                 >>> bipartite.average_clustering(G, X)
(4)                 0.0
(4)                 >>> bipartite.average_clustering(G, Y)
(4)                 1.0
(4)                 See Also
(4)                 --------
(4)                 clustering
(4)                 Notes
(4)                 -----
(4)                 The container of nodes passed to this function must contain all of the nodes
(4)                 in one of the bipartite sets ("top" or "bottom") in order to compute
(4)                 the correct average bipartite clustering coefficients.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 References
(4)                 ----------
(4)                 .. [1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio (2008).
(8)                     Basic notions for the analysis of large two-mode networks.
(8)                     Social Networks 30(1), 31--48.
(4)                 """
(4)                 if nodes is None:
(8)                     nodes = G
(4)                 ccs = latapy_clustering(G, nodes=nodes, mode=mode)
(4)                 return sum(ccs[v] for v in nodes) / len(nodes)
(0)             @nx._dispatchable
(0)             def robins_alexander_clustering(G):
(4)                 r"""Compute the bipartite clustering of G.
(4)                 Robins and Alexander [1]_ defined bipartite clustering coefficient as
(4)                 four times the number of four cycles `C_4` divided by the number of
(4)                 three paths `L_3` in a bipartite graph:
(4)                 .. math::
(7)                    CC_4 = \frac{4 * C_4}{L_3}
```

```
160: (4)                Parameters
161: (4)                ----------
162: (4)                G : graph
163: (8)                    a bipartite graph
164: (4)                Returns
165: (4)                -------
166: (4)                clustering : float
167: (7)                   The Robins and Alexander bipartite clustering for the input graph.
168: (4)                Examples
169: (4)                --------
170: (4)                >>> from networkx.algorithms import bipartite
171: (4)                >>> G = nx.davis_southern_women_graph()
172: (4)                >>> print(round(bipartite.robins_alexander_clustering(G), 3))
173: (4)                0.468
174: (4)                See Also
175: (4)                --------
176: (4)                latapy_clustering
177: (4)                networkx.algorithms.cluster.square_clustering
178: (4)                References
179: (4)                ----------
180: (4)                .. [1] Robins, G. and M. Alexander (2004). Small worlds among interlocking
181: (11)                      directors: Network structure and distance in bipartite graphs.
182: (11)                      Computational & Mathematical Organization Theory 10(1), 69–94.
183: (4)                """
184: (4)                if G.order() < 4 or G.size() < 3:
185: (8)                    return 0
186: (4)                L_3 = _threepaths(G)
187: (4)                if L_3 == 0:
188: (8)                    return 0
189: (4)                C_4 = _four_cycles(G)
190: (4)                return (4.0 * C_4) / L_3
191: (0)            def _four_cycles(G):
192: (4)                cycles = 0
193: (4)                for v in G:
194: (8)                    for u, w in itertools.combinations(G[v], 2):
195: (12)                       cycles += len((set(G[u]) & set(G[w])) - {v})
196: (4)                return cycles / 4
197: (0)            def _threepaths(G):
198: (4)                paths = 0
199: (4)                for v in G:
200: (8)                    for u in G[v]:
201: (12)                       for w in set(G[u]) - {v}:
202: (16)                           paths += len(set(G[w]) - {v, u})
203: (4)                return paths / 2
```

File 109 - covering.py:

```
1: (0)              """ Functions related to graph covers."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.bipartite.matching import hopcroft_karp_matching
4: (0)              from networkx.algorithms.covering import min_edge_cover as _min_edge_cover
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = ["min_edge_cover"]
7: (0)              @not_implemented_for("directed")
8: (0)              @not_implemented_for("multigraph")
9: (0)              @nx._dispatchable(name="bipartite_min_edge_cover")
10: (0)             def min_edge_cover(G, matching_algorithm=None):
11: (4)                 """Returns a set of edges which constitutes
12: (4)                 the minimum edge cover of the graph.
13: (4)                 The smallest edge cover can be found in polynomial time by finding
14: (4)                 a maximum matching and extending it greedily so that all nodes
15: (4)                 are covered.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G : NetworkX graph
19: (8)                     An undirected bipartite graph.
20: (4)                 matching_algorithm : function
```

```
21: (8)                     A function that returns a maximum cardinality matching in a
22: (8)                     given bipartite graph. The function must take one input, the
23: (8)                     graph ``G``, and return a dictionary mapping each node to its
24: (8)                     mate. If not specified,
25: (8)                     :func:`~networkx.algorithms.bipartite.matching.hopcroft_karp_matching`
26: (8)                     will be used. Other possibilities include
27: (8)                     :func:`~networkx.algorithms.bipartite.matching.eppstein_matching`,
28: (4)                 Returns
29: (4)                 -------
30: (4)                 set
31: (8)                     A set of the edges in a minimum edge cover of the graph, given as
32: (8)                     pairs of nodes. It contains both the edges `(u, v)` and `(v, u)`
33: (8)                     for given nodes `u` and `v` among the edges of minimum edge cover.
34: (4)                 Notes
35: (4)                 -----
36: (4)                 An edge cover of a graph is a set of edges such that every node of
37: (4)                 the graph is incident to at least one edge of the set.
38: (4)                 A minimum edge cover is an edge covering of smallest cardinality.
39: (4)                 Due to its implementation, the worst-case running time of this algorithm
40: (4)                 is bounded by the worst-case running time of the function
41: (4)                 ``matching_algorithm``.
42: (4)                 """
43: (4)                 if G.order() == 0:  # Special case for the empty graph
44: (8)                     return set()
45: (4)                 if matching_algorithm is None:
46: (8)                     matching_algorithm = hopcroft_karp_matching
47: (4)                 return _min_edge_cover(G, matching_algorithm=matching_algorithm)
```

File 110 - edgelist.py:

```
1: (0)              """
2: (0)              ********************
3: (0)              Bipartite Edge Lists
4: (0)              ********************
5: (0)              Read and write NetworkX graphs as bipartite edge lists.
6: (0)              Format
7: (0)              ------
8: (0)              You can read or write three formats of edge lists with these functions.
9: (0)              Node pairs with no data::
10: (1)              1 2
11: (0)             Python dictionary as data::
12: (1)              1 2 {'weight':7, 'color':'green'}
13: (0)             Arbitrary data::
14: (1)              1 2 7 green
15: (0)             For each edge (u, v) the node u is assigned to part 0 and the node v to part
1.
16: (0)             """
17: (0)             __all__ = ["generate_edgelist", "write_edgelist", "parse_edgelist",
"read_edgelist"]
18: (0)             import networkx as nx
19: (0)             from networkx.utils import not_implemented_for, open_file
20: (0)             @open_file(1, mode="wb")
21: (0)             def write_edgelist(G, path, comments="#", delimiter=" ", data=True,
encoding="utf-8"):
22: (4)                 """Write a bipartite graph as a list of edges.
23: (4)                 Parameters
24: (4)                 ----------
25: (4)                 G : Graph
26: (7)                    A NetworkX bipartite graph
27: (4)                 path : file or string
28: (7)                    File or filename to write. If a file is provided, it must be
29: (7)                    opened in 'wb' mode. Filenames ending in .gz or .bz2 will be
compressed.
30: (4)                 comments : string, optional
31: (7)                    The character used to indicate the start of a comment
32: (4)                 delimiter : string, optional
33: (7)                    The string used to separate values.  The default is whitespace.
```

```
(4)                 data : bool or list, optional
(7)                    If False write no edge data.
(7)                    If True write a string representation of the edge data dictionary..
(7)                    If a list (or other iterable) is provided, write the  keys specified
(7)                    in the list.
(4)                 encoding: string, optional
(7)                    Specify which encoding to use when writing file.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> G.add_nodes_from([0, 2], bipartite=0)
(4)                 >>> G.add_nodes_from([1, 3], bipartite=1)
(4)                 >>> nx.write_edgelist(G, "test.edgelist")
(4)                 >>> fh = open("test.edgelist", "wb")
(4)                 >>> nx.write_edgelist(G, fh)
(4)                 >>> nx.write_edgelist(G, "test.edgelist.gz")
(4)                 >>> nx.write_edgelist(G, "test.edgelist.gz", data=False)
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edge(1, 2, weight=7, color="red")
(4)                 >>> nx.write_edgelist(G, "test.edgelist", data=False)
(4)                 >>> nx.write_edgelist(G, "test.edgelist", data=["color"])
(4)                 >>> nx.write_edgelist(G, "test.edgelist", data=["color", "weight"])
(4)                 See Also
(4)                 --------
(4)                 write_edgelist
(4)                 generate_edgelist
(4)                 """
(4)                 for line in generate_edgelist(G, delimiter, data):
(8)                     line += "\n"
(8)                     path.write(line.encode(encoding))
(0)             @not_implemented_for("directed")
(0)             def generate_edgelist(G, delimiter=" ", data=True):
(4)                 """Generate a single line of the bipartite graph G in edge list format.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    The graph is assumed to have node attribute `part` set to 0,1
representing
(7)                    the two graph parts
(4)                 delimiter : string, optional
(7)                    Separator for node labels
(4)                 data : bool or list of keys
(7)                    If False generate no edge data.  If True use a dictionary
(7)                    representation of edge data.  If a list of keys use a list of data
(7)                    values corresponding to the keys.
(4)                 Returns
(4)                 -------
(4)                 lines : string
(8)                     Lines of data in adjlist format.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> G.add_nodes_from([0, 2], bipartite=0)
(4)                 >>> G.add_nodes_from([1, 3], bipartite=1)
(4)                 >>> G[1][2]["weight"] = 3
(4)                 >>> G[2][3]["capacity"] = 12
(4)                 >>> for line in bipartite.generate_edgelist(G, data=False):
(4)                 ...     print(line)
(4)                 0 1
(4)                 2 1
(4)                 2 3
(4)                 >>> for line in bipartite.generate_edgelist(G):
(4)                 ...     print(line)
(4)                 0 1 {}
(4)                 2 1 {'weight': 3}
(4)                 2 3 {'capacity': 12}
(4)                 >>> for line in bipartite.generate_edgelist(G, data=["weight"]):
(4)                 ...     print(line)
```

```
(4)                 0 1
(4)                 2 1 3
(4)                 2 3
(4)                 """
(4)                 try:
(8)                     part0 = [n for n, d in G.nodes.items() if d["bipartite"] == 0]
(4)                 except BaseException as err:
(8)                     raise AttributeError("Missing node attribute `bipartite`") from err
(4)                 if data is True or data is False:
(8)                     for n in part0:
(12)                        for edge in G.edges(n, data=data):
(16)                            yield delimiter.join(map(str, edge))
(4)                 else:
(8)                     for n in part0:
(12)                        for u, v, d in G.edges(n, data=True):
(16)                            edge = [u, v]
(16)                            try:
(20)                                edge.extend(d[k] for k in data)
(16)                            except KeyError:
(20)                                pass  # missing data for this edge, should warn?
(16)                            yield delimiter.join(map(str, edge))
(0)             @nx._dispatchable(name="bipartite_parse_edgelist", graphs=None,
returns_graph=True)
(0)             def parse_edgelist(
(4)                 lines, comments="#", delimiter=None, create_using=None, nodetype=None,
data=True
(0)             ):
(4)                 """Parse lines of an edge list representation of a bipartite graph.
(4)                 Parameters
(4)                 ----------
(4)                 lines : list or iterator of strings
(8)                     Input data in edgelist format
(4)                 comments : string, optional
(7)                    Marker for comment lines
(4)                 delimiter : string, optional
(7)                    Separator for node labels
(4)                 create_using: NetworkX graph container, optional
(7)                    Use given NetworkX graph for holding nodes or edges.
(4)                 nodetype : Python type, optional
(7)                    Convert nodes to this type.
(4)                 data : bool or list of (label,type) tuples
(7)                    If False generate no edge data or if True use a dictionary
(7)                    representation of edge data or a list tuples specifying dictionary
(7)                    key names and types for edge data.
(4)                 Returns
(4)                 -------
(4)                 G: NetworkX Graph
(8)                     The bipartite graph corresponding to lines
(4)                 Examples
(4)                 --------
(4)                 Edgelist with no data:
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> lines = ["1 2", "2 3", "3 4"]
(4)                 >>> G = bipartite.parse_edgelist(lines, nodetype=int)
(4)                 >>> sorted(G.nodes())
(4)                 [1, 2, 3, 4]
(4)                 >>> sorted(G.nodes(data=True))
(4)                 [(1, {'bipartite': 0}), (2, {'bipartite': 0}), (3, {'bipartite': 0}), (4,
{'bipartite': 1})]
(4)                 >>> sorted(G.edges())
(4)                 [(1, 2), (2, 3), (3, 4)]
(4)                 Edgelist with data in Python dictionary representation:
(4)                 >>> lines = ["1 2 {'weight':3}", "2 3 {'weight':27}", "3 4
{'weight':3.0}"]
(4)                 >>> G = bipartite.parse_edgelist(lines, nodetype=int)
(4)                 >>> sorted(G.nodes())
(4)                 [1, 2, 3, 4]
(4)                 >>> sorted(G.edges(data=True))
(4)                 [(1, 2, {'weight': 3}), (2, 3, {'weight': 27}), (3, 4, {'weight': 3.0})]
```

```
    Edgelist with data in a list:
    >>> lines = ["1 2 3", "2 3 27", "3 4 3.0"]
    >>> G = bipartite.parse_edgelist(lines, nodetype=int, data=(("weight", float),))
    >>> sorted(G.nodes())
    [1, 2, 3, 4]
    >>> sorted(G.edges(data=True))
    [(1, 2, {'weight': 3.0}), (2, 3, {'weight': 27.0}), (3, 4, {'weight': 3.0})]
    See Also
    --------
    """
    from ast import literal_eval
    G = nx.empty_graph(0, create_using)
    for line in lines:
        p = line.find(comments)
        if p >= 0:
            line = line[:p]
        if not len(line):
            continue
        s = line.strip().split(delimiter)
        if len(s) < 2:
            continue
        u = s.pop(0)
        v = s.pop(0)
        d = s
        if nodetype is not None:
            try:
                u = nodetype(u)
                v = nodetype(v)
            except BaseException as err:
                raise TypeError(
                    f"Failed to convert nodes {u},{v} to type {nodetype}."
                ) from err
        if len(d) == 0 or data is False:
            edgedata = {}
        elif data is True:
            try:  # try to evaluate as dictionary
                edgedata = dict(literal_eval(" ".join(d)))
            except BaseException as err:
                raise TypeError(
                    f"Failed to convert edge data ({d}) to dictionary."
                ) from err
        else:
            if len(d) != len(data):
                raise IndexError(
                    f"Edge data {d} and data_keys {data} are not the same length"
                )
            edgedata = {}
            for (edge_key, edge_type), edge_value in zip(data, d):
                try:
                    edge_value = edge_type(edge_value)
                except BaseException as err:
                    raise TypeError(
                        f"Failed to convert {edge_key} data "
                        f"{edge_value} to type {edge_type}."
                    ) from err
                edgedata.update({edge_key: edge_value})
        G.add_node(u, bipartite=0)
        G.add_node(v, bipartite=1)
        G.add_edge(u, v, **edgedata)
    return G
@open_file(0, mode="rb")
@nx._dispatchable(name="bipartite_read_edgelist", graphs=None, returns_graph=True)
def read_edgelist(
    path,
    comments="#",
```

```
(4)                 delimiter=None,
(4)                 create_using=None,
(4)                 nodetype=None,
(4)                 data=True,
(4)                 edgetype=None,
(4)                 encoding="utf-8",
(0)             ):
(4)                 """Read a bipartite graph from a list of edges.
(4)                 Parameters
(4)                 ----------
(4)                 path : file or string
(7)                    File or filename to read. If a file is provided, it must be
(7)                    opened in 'rb' mode.
(7)                    Filenames ending in .gz or .bz2 will be uncompressed.
(4)                 comments : string, optional
(7)                    The character used to indicate the start of a comment.
(4)                 delimiter : string, optional
(7)                    The string used to separate values.  The default is whitespace.
(4)                 create_using : Graph container, optional,
(7)                    Use specified container to build graph.  The default is networkx.Graph,
(7)                    an undirected graph.
(4)                 nodetype : int, float, str, Python type, optional
(7)                    Convert node data from strings to specified type
(4)                 data : bool or list of (label,type) tuples
(7)                    Tuples specifying dictionary key names and types for edge data
(4)                 edgetype : int, float, str, Python type, optional OBSOLETE
(7)                    Convert edge data from strings to specified type and use as 'weight'
(4)                 encoding: string, optional
(7)                    Specify which encoding to use when reading file.
(4)                 Returns
(4)                 -------
(4)                 G : graph
(7)                    A networkx Graph or other type specified with create_using
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> G.add_nodes_from([0, 2], bipartite=0)
(4)                 >>> G.add_nodes_from([1, 3], bipartite=1)
(4)                 >>> bipartite.write_edgelist(G, "test.edgelist")
(4)                 >>> G = bipartite.read_edgelist("test.edgelist")
(4)                 >>> fh = open("test.edgelist", "rb")
(4)                 >>> G = bipartite.read_edgelist(fh)
(4)                 >>> fh.close()
(4)                 >>> G = bipartite.read_edgelist("test.edgelist", nodetype=int)
(4)                 Edgelist with data in a list:
(4)                 >>> textline = "1 2 3"
(4)                 >>> fh = open("test.edgelist", "w")
(4)                 >>> d = fh.write(textline)
(4)                 >>> fh.close()
(4)                 >>> G = bipartite.read_edgelist(
(4)                 ...     "test.edgelist", nodetype=int, data=(("weight", float),)
(4)                 ... )
(4)                 >>> list(G)
(4)                 [1, 2]
(4)                 >>> list(G.edges(data=True))
(4)                 [(1, 2, {'weight': 3.0})]
(4)                 See parse_edgelist() for more examples of formatting.
(4)                 See Also
(4)                 --------
(4)                 parse_edgelist
(4)                 Notes
(4)                 -----
(4)                 Since nodes must be hashable, the function nodetype must return hashable
(4)                 types (e.g. int, float, str, frozenset - or tuples of those, etc.)
(4)                 """
(4)                 lines = (line.decode(encoding) for line in path)
(4)                 return parse_edgelist(
(8)                     lines,
```

```
301: (8)                    comments=comments,
302: (8)                    delimiter=delimiter,
303: (8)                    create_using=create_using,
304: (8)                    nodetype=nodetype,
305: (8)                    data=data,
306: (4)                )
```

File 111 - generators.py:

```
1: (0)              """
2: (0)              Generators and functions for bipartite graphs.
3: (0)              """
4: (0)              import math
5: (0)              import numbers
6: (0)              from functools import reduce
7: (0)              import networkx as nx
8: (0)              from networkx.utils import nodes_or_number, py_random_state
9: (0)              __all__ = [
10: (4)                 "configuration_model",
11: (4)                 "havel_hakimi_graph",
12: (4)                 "reverse_havel_hakimi_graph",
13: (4)                 "alternating_havel_hakimi_graph",
14: (4)                 "preferential_attachment_graph",
15: (4)                 "random_graph",
16: (4)                 "gnmk_random_graph",
17: (4)                 "complete_bipartite_graph",
18: (0)             ]
19: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
20: (0)             @nodes_or_number([0, 1])
21: (0)             def complete_bipartite_graph(n1, n2, create_using=None):
22: (4)                 """Returns the complete bipartite graph `K_{n_1,n_2}`.
23: (4)                 The graph is composed of two partitions with nodes 0 to (n1 - 1)
24: (4)                 in the first and nodes n1 to (n1 + n2 - 1) in the second.
25: (4)                 Each node in the first is connected to each node in the second.
26: (4)                 Parameters
27: (4)                 ----------
28: (4)                 n1, n2 : integer or iterable container of nodes
29: (8)                     If integers, nodes are from `range(n1)` and `range(n1, n1 + n2)`.
30: (8)                     If a container, the elements are the nodes.
31: (4)                 create_using : NetworkX graph instance, (default: nx.Graph)
32: (7)                    Return graph of this type.
33: (4)                 Notes
34: (4)                 -----
35: (4)                 Nodes are the integers 0 to `n1 + n2 - 1` unless either n1 or n2 are
36: (4)                 containers of nodes. If only one of n1 or n2 are integers, that
37: (4)                 integer is replaced by `range` of that integer.
38: (4)                 The nodes are assigned the attribute 'bipartite' with the value 0 or 1
39: (4)                 to indicate which bipartite set the node belongs to.
40: (4)                 This function is not imported in the main namespace.
41: (4)                 To use it use nx.bipartite.complete_bipartite_graph
42: (4)                 """
43: (4)                 G = nx.empty_graph(0, create_using)
44: (4)                 if G.is_directed():
45: (8)                     raise nx.NetworkXError("Directed Graph not supported")
46: (4)                 n1, top = n1
47: (4)                 n2, bottom = n2
48: (4)                 if isinstance(n1, numbers.Integral) and isinstance(n2, numbers.Integral):
49: (8)                     bottom = [n1 + i for i in bottom]
50: (4)                 G.add_nodes_from(top, bipartite=0)
51: (4)                 G.add_nodes_from(bottom, bipartite=1)
52: (4)                 if len(G) != len(top) + len(bottom):
53: (8)                     raise nx.NetworkXError("Inputs n1 and n2 must contain distinct nodes")
54: (4)                 G.add_edges_from((u, v) for u in top for v in bottom)
55: (4)                 G.graph["name"] = f"complete_bipartite_graph({n1}, {n2})"
56: (4)                 return G
57: (0)             @py_random_state(3)
58: (0)             @nx._dispatchable(name="bipartite_configuration_model", graphs=None,
```

```
returns_graph=True)
59: (0)             def configuration_model(aseq, bseq, create_using=None, seed=None):
60: (4)                 """Returns a random bipartite graph from two given degree sequences.
61: (4)                 Parameters
62: (4)                 ----------
63: (4)                 aseq : list
64: (7)                    Degree sequence for node set A.
65: (4)                 bseq : list
66: (7)                    Degree sequence for node set B.
67: (4)                 create_using : NetworkX graph instance, optional
68: (7)                    Return graph of this type.
69: (4)                 seed : integer, random_state, or None (default)
70: (8)                     Indicator of random number generation state.
71: (8)                     See :ref:`Randomness<randomness>`.
72: (4)                 The graph is composed of two partitions. Set A has nodes 0 to
73: (4)                 (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1).
74: (4)                 Nodes from set A are connected to nodes in set B by choosing
75: (4)                 randomly from the possible free stubs, one in A and one in B.
76: (4)                 Notes
77: (4)                 -----
78: (4)                 The sum of the two sequences must be equal: sum(aseq)=sum(bseq)
79: (4)                 If no graph type is specified use MultiGraph with parallel edges.
80: (4)                 If you want a graph with no parallel edges use create_using=Graph()
81: (4)                 but then the resulting degree sequences might not be exact.
82: (4)                 The nodes are assigned the attribute 'bipartite' with the value 0 or 1
83: (4)                 to indicate which bipartite set the node belongs to.
84: (4)                 This function is not imported in the main namespace.
85: (4)                 To use it use nx.bipartite.configuration_model
86: (4)                 """
87: (4)                 G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
88: (4)                 if G.is_directed():
89: (8)                     raise nx.NetworkXError("Directed Graph not supported")
90: (4)                 lena = len(aseq)
91: (4)                 lenb = len(bseq)
92: (4)                 suma = sum(aseq)
93: (4)                 sumb = sum(bseq)
94: (4)                 if not suma == sumb:
95: (8)                     raise nx.NetworkXError(
96: (12)                        f"invalid degree sequences, sum(aseq)!=sum(bseq),{suma},{sumb}"
97: (8)                     )
98: (4)                 G = _add_nodes_with_bipartite_label(G, lena, lenb)
99: (4)                 if len(aseq) == 0 or max(aseq) == 0:
100: (8)                    return G  # done if no edges
101: (4)                stubs = [[v] * aseq[v] for v in range(lena)]
102: (4)                astubs = [x for subseq in stubs for x in subseq]
103: (4)                stubs = [[v] * bseq[v - lena] for v in range(lena, lena + lenb)]
104: (4)                bstubs = [x for subseq in stubs for x in subseq]
105: (4)                seed.shuffle(astubs)
106: (4)                seed.shuffle(bstubs)
107: (4)                G.add_edges_from([astubs[i], bstubs[i]] for i in range(suma))
108: (4)                G.name = "bipartite_configuration_model"
109: (4)                return G
110: (0)            @nx._dispatchable(name="bipartite_havel_hakimi_graph", graphs=None,
returns_graph=True)
111: (0)            def havel_hakimi_graph(aseq, bseq, create_using=None):
112: (4)                """Returns a bipartite graph from two given degree sequences using a
113: (4)                Havel-Hakimi style construction.
114: (4)                The graph is composed of two partitions. Set A has nodes 0 to
115: (4)                (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1).
116: (4)                Nodes from the set A are connected to nodes in the set B by
117: (4)                connecting the highest degree nodes in set A to the highest degree
118: (4)                nodes in set B until all stubs are connected.
119: (4)                Parameters
120: (4)                ----------
121: (4)                aseq : list
122: (7)                   Degree sequence for node set A.
123: (4)                bseq : list
124: (7)                   Degree sequence for node set B.
125: (4)                create_using : NetworkX graph instance, optional
```

```
(7)                 Return graph of this type.
(4)             Notes
(4)             -----
(4)             The sum of the two sequences must be equal: sum(aseq)=sum(bseq)
(4)             If no graph type is specified use MultiGraph with parallel edges.
(4)             If you want a graph with no parallel edges use create_using=Graph()
(4)             but then the resulting degree sequences might not be exact.
(4)             The nodes are assigned the attribute 'bipartite' with the value 0 or 1
(4)             to indicate which bipartite set the node belongs to.
(4)             This function is not imported in the main namespace.
(4)             To use it use nx.bipartite.havel_hakimi_graph
(4)             """
(4)             G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
(4)             if G.is_directed():
(8)                 raise nx.NetworkXError("Directed Graph not supported")
(4)             naseq = len(aseq)
(4)             nbseq = len(bseq)
(4)             suma = sum(aseq)
(4)             sumb = sum(bseq)
(4)             if not suma == sumb:
(8)                 raise nx.NetworkXError(
(12)                    f"invalid degree sequences, sum(aseq)!=sum(bseq),{suma},{sumb}"
(8)                 )
(4)             G = _add_nodes_with_bipartite_label(G, naseq, nbseq)
(4)             if len(aseq) == 0 or max(aseq) == 0:
(8)                 return G  # done if no edges
(4)             astubs = [[aseq[v], v] for v in range(naseq)]
(4)             bstubs = [[bseq[v - naseq], v] for v in range(naseq, naseq + nbseq)]
(4)             astubs.sort()
(4)             while astubs:
(8)                 (degree, u) = astubs.pop()  # take of largest degree node in the a set
(8)                 if degree == 0:
(12)                    break  # done, all are zero
(8)                 bstubs.sort()
(8)                 for target in bstubs[-degree:]:
(12)                    v = target[1]
(12)                    G.add_edge(u, v)
(12)                    target[0] -= 1  # note this updates bstubs too.
(12)                    if target[0] == 0:
(16)                        bstubs.remove(target)
(4)             G.name = "bipartite_havel_hakimi_graph"
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def reverse_havel_hakimi_graph(aseq, bseq, create_using=None):
(4)             """Returns a bipartite graph from two given degree sequences using a
(4)             Havel-Hakimi style construction.
(4)             The graph is composed of two partitions. Set A has nodes 0 to
(4)             (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1).
(4)             Nodes from set A are connected to nodes in the set B by connecting
(4)             the highest degree nodes in set A to the lowest degree nodes in
(4)             set B until all stubs are connected.
(4)             Parameters
(4)             ----------
(4)             aseq : list
(7)                Degree sequence for node set A.
(4)             bseq : list
(7)                Degree sequence for node set B.
(4)             create_using : NetworkX graph instance, optional
(7)                Return graph of this type.
(4)             Notes
(4)             -----
(4)             The sum of the two sequences must be equal: sum(aseq)=sum(bseq)
(4)             If no graph type is specified use MultiGraph with parallel edges.
(4)             If you want a graph with no parallel edges use create_using=Graph()
(4)             but then the resulting degree sequences might not be exact.
(4)             The nodes are assigned the attribute 'bipartite' with the value 0 or 1
(4)             to indicate which bipartite set the node belongs to.
(4)             This function is not imported in the main namespace.
(4)             To use it use nx.bipartite.reverse_havel_hakimi_graph
```

```
(4)                 """
(4)                 G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
(4)                 if G.is_directed():
(8)                     raise nx.NetworkXError("Directed Graph not supported")
(4)                 lena = len(aseq)
(4)                 lenb = len(bseq)
(4)                 suma = sum(aseq)
(4)                 sumb = sum(bseq)
(4)                 if not suma == sumb:
(8)                     raise nx.NetworkXError(
(12)                        f"invalid degree sequences, sum(aseq)!=sum(bseq),{suma},{sumb}"
(8)                     )
(4)                 G = _add_nodes_with_bipartite_label(G, lena, lenb)
(4)                 if len(aseq) == 0 or max(aseq) == 0:
(8)                     return G  # done if no edges
(4)                 astubs = [[aseq[v], v] for v in range(lena)]
(4)                 bstubs = [[bseq[v - lena], v] for v in range(lena, lena + lenb)]
(4)                 astubs.sort()
(4)                 bstubs.sort()
(4)                 while astubs:
(8)                     (degree, u) = astubs.pop()  # take of largest degree node in the a set
(8)                     if degree == 0:
(12)                        break  # done, all are zero
(8)                     for target in bstubs[0:degree]:
(12)                        v = target[1]
(12)                        G.add_edge(u, v)
(12)                        target[0] -= 1  # note this updates bstubs too.
(12)                        if target[0] == 0:
(16)                            bstubs.remove(target)
(4)                 G.name = "bipartite_reverse_havel_hakimi_graph"
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def alternating_havel_hakimi_graph(aseq, bseq, create_using=None):
(4)                 """Returns a bipartite graph from two given degree sequences using
(4)                 an alternating Havel-Hakimi style construction.
(4)                 The graph is composed of two partitions. Set A has nodes 0 to
(4)                 (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1).
(4)                 Nodes from the set A are connected to nodes in the set B by
(4)                 connecting the highest degree nodes in set A to alternatively the
(4)                 highest and the lowest degree nodes in set B until all stubs are
(4)                 connected.
(4)                 Parameters
(4)                 ----------
(4)                 aseq : list
(7)                    Degree sequence for node set A.
(4)                 bseq : list
(7)                    Degree sequence for node set B.
(4)                 create_using : NetworkX graph instance, optional
(7)                    Return graph of this type.
(4)                 Notes
(4)                 -----
(4)                 The sum of the two sequences must be equal: sum(aseq)=sum(bseq)
(4)                 If no graph type is specified use MultiGraph with parallel edges.
(4)                 If you want a graph with no parallel edges use create_using=Graph()
(4)                 but then the resulting degree sequences might not be exact.
(4)                 The nodes are assigned the attribute 'bipartite' with the value 0 or 1
(4)                 to indicate which bipartite set the node belongs to.
(4)                 This function is not imported in the main namespace.
(4)                 To use it use nx.bipartite.alternating_havel_hakimi_graph
(4)                 """
(4)                 G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
(4)                 if G.is_directed():
(8)                     raise nx.NetworkXError("Directed Graph not supported")
(4)                 naseq = len(aseq)
(4)                 nbseq = len(bseq)
(4)                 suma = sum(aseq)
(4)                 sumb = sum(bseq)
(4)                 if not suma == sumb:
(8)                     raise nx.NetworkXError(
```

```
(12)                        f"invalid degree sequences, sum(aseq)!=sum(bseq),{suma},{sumb}"
(8)                     )
(4)                 G = _add_nodes_with_bipartite_label(G, naseq, nbseq)
(4)                 if len(aseq) == 0 or max(aseq) == 0:
(8)                     return G  # done if no edges
(4)                 astubs = [[aseq[v], v] for v in range(naseq)]
(4)                 bstubs = [[bseq[v - naseq], v] for v in range(naseq, naseq + nbseq)]
(4)                 while astubs:
(8)                     astubs.sort()
(8)                     (degree, u) = astubs.pop()  # take of largest degree node in the a set
(8)                     if degree == 0:
(12)                        break  # done, all are zero
(8)                     bstubs.sort()
(8)                     small = bstubs[0 : degree // 2]  # add these low degree targets
(8)                     large = bstubs[(-degree + degree // 2) :]  # now high degree targets
(8)                     stubs = [x for z in zip(large, small) for x in z]  # combine, sorry
(8)                     if len(stubs) < len(small) + len(large):  # check for zip truncation
(12)                        stubs.append(large.pop())
(8)                     for target in stubs:
(12)                        v = target[1]
(12)                        G.add_edge(u, v)
(12)                        target[0] -= 1  # note this updates bstubs too.
(12)                        if target[0] == 0:
(16)                            bstubs.remove(target)
(4)                 G.name = "bipartite_alternating_havel_hakimi_graph"
(4)                 return G
(0)             @py_random_state(3)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def preferential_attachment_graph(aseq, p, create_using=None, seed=None):
(4)                 """Create a bipartite graph with a preferential attachment model from
(4)                 a given single degree sequence.
(4)                 The graph is composed of two partitions. Set A has nodes 0 to
(4)                 (len(aseq) - 1) and set B has nodes starting with node len(aseq).
(4)                 The number of nodes in set B is random.
(4)                 Parameters
(4)                 ----------
(4)                 aseq : list
(7)                    Degree sequence for node set A.
(4)                 p :  float
(7)                    Probability that a new bottom node is added.
(4)                 create_using : NetworkX graph instance, optional
(7)                    Return graph of this type.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 References
(4)                 ----------
(4)                 .. [1] Guillaume, J.L. and Latapy, M.,
(7)                    Bipartite graphs as models of complex networks.
(7)                    Physica A: Statistical Mechanics and its Applications,
(7)                    2006, 371(2), pp.795-813.
(4)                 .. [2] Jean-Loup Guillaume and Matthieu Latapy,
(7)                    Bipartite structure of all complex networks,
(7)                    Inf. Process. Lett. 90, 2004, pg. 215-221
(7)                    https://doi.org/10.1016/j.ipl.2004.03.007
(4)                 Notes
(4)                 -----
(4)                 The nodes are assigned the attribute 'bipartite' with the value 0 or 1
(4)                 to indicate which bipartite set the node belongs to.
(4)                 This function is not imported in the main namespace.
(4)                 To use it use nx.bipartite.preferential_attachment_graph
(4)                 """
(4)                 G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
(4)                 if G.is_directed():
(8)                     raise nx.NetworkXError("Directed Graph not supported")
(4)                 if p > 1:
(8)                     raise nx.NetworkXError(f"probability {p} > 1")
(4)                 naseq = len(aseq)
(4)                 G = _add_nodes_with_bipartite_label(G, naseq, 0)
```

```
(4)                 vv = [[v] * aseq[v] for v in range(naseq)]
(4)                 while vv:
(8)                     while vv[0]:
(12)                        source = vv[0][0]
(12)                        vv[0].remove(source)
(12)                        if seed.random() < p or len(G) == naseq:
(16)                            target = len(G)
(16)                            G.add_node(target, bipartite=1)
(16)                            G.add_edge(source, target)
(12)                        else:
(16)                            bb = [[b] * G.degree(b) for b in range(naseq, len(G))]
(16)                            bbstubs = reduce(lambda x, y: x + y, bb)
(16)                            target = seed.choice(bbstubs)
(16)                            G.add_node(target, bipartite=1)
(16)                            G.add_edge(source, target)
(8)                     vv.remove(vv[0])
(4)                 G.name = "bipartite_preferential_attachment_model"
(4)                 return G
(0)             @py_random_state(3)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_graph(n, m, p, seed=None, directed=False):
(4)                 """Returns a bipartite random graph.
(4)                 This is a bipartite version of the binomial (Erdős-Rényi) graph.
(4)                 The graph is composed of two partitions. Set A has nodes 0 to
(4)                 (n - 1) and set B has nodes n to (n + m - 1).
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes in the first bipartite set.
(4)                 m : int
(8)                     The number of nodes in the second bipartite set.
(4)                 p : float
(8)                     Probability for edge creation.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 directed : bool, optional (default=False)
(8)                     If True return a directed graph
(4)                 Notes
(4)                 -----
(4)                 The bipartite random graph algorithm chooses each of the n*m (undirected)
(4)                 or 2*nm (directed) possible edges with probability p.
(4)                 This algorithm is $O(n+m)$ where $m$ is the expected number of edges.
(4)                 The nodes are assigned the attribute 'bipartite' with the value 0 or 1
(4)                 to indicate which bipartite set the node belongs to.
(4)                 This function is not imported in the main namespace.
(4)                 To use it use nx.bipartite.random_graph
(4)                 See Also
(4)                 --------
(4)                 gnp_random_graph, configuration_model
(4)                 References
(4)                 ----------
(4)                 .. [1] Vladimir Batagelj and Ulrik Brandes,
(7)                    "Efficient generation of large random networks",
(7)                    Phys. Rev. E, 71, 036113, 2005.
(4)                 """
(4)                 G = nx.Graph()
(4)                 G = _add_nodes_with_bipartite_label(G, n, m)
(4)                 if directed:
(8)                     G = nx.DiGraph(G)
(4)                 G.name = f"fast_gnp_random_graph({n},{m},{p})"
(4)                 if p <= 0:
(8)                     return G
(4)                 if p >= 1:
(8)                     return nx.complete_bipartite_graph(n, m)
(4)                 lp = math.log(1.0 - p)
(4)                 v = 0
(4)                 w = -1
(4)                 while v < n:
```

```
402: (8)                    lr = math.log(1.0 - seed.random())
403: (8)                    w = w + 1 + int(lr / lp)
404: (8)                    while w >= m and v < n:
405: (12)                       w = w - m
406: (12)                       v = v + 1
407: (8)                    if v < n:
408: (12)                       G.add_edge(v, n + w)
409: (4)                if directed:
410: (8)                    v = 0
411: (8)                    w = -1
412: (8)                    while v < n:
413: (12)                       lr = math.log(1.0 - seed.random())
414: (12)                       w = w + 1 + int(lr / lp)
415: (12)                       while w >= m and v < n:
416: (16)                           w = w - m
417: (16)                           v = v + 1
418: (12)                       if v < n:
419: (16)                           G.add_edge(n + w, v)
420: (4)                return G
421: (0)            @py_random_state(3)
422: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
423: (0)            def gnmk_random_graph(n, m, k, seed=None, directed=False):
424: (4)                """Returns a random bipartite graph G_{n,m,k}.
425: (4)                Produces a bipartite graph chosen randomly out of the set of all graphs
426: (4)                with n top nodes, m bottom nodes, and k edges.
427: (4)                The graph is composed of two sets of nodes.
428: (4)                Set A has nodes 0 to (n - 1) and set B has nodes n to (n + m - 1).
429: (4)                Parameters
430: (4)                ----------
431: (4)                n : int
432: (8)                    The number of nodes in the first bipartite set.
433: (4)                m : int
434: (8)                    The number of nodes in the second bipartite set.
435: (4)                k : int
436: (8)                    The number of edges
437: (4)                seed : integer, random_state, or None (default)
438: (8)                    Indicator of random number generation state.
439: (8)                    See :ref:`Randomness<randomness>`.
440: (4)                directed : bool, optional (default=False)
441: (8)                    If True return a directed graph
442: (4)                Examples
443: (4)                --------
444: (4)                from nx.algorithms import bipartite
445: (4)                G = bipartite.gnmk_random_graph(10,20,50)
446: (4)                See Also
447: (4)                --------
448: (4)                gnm_random_graph
449: (4)                Notes
450: (4)                -----
451: (4)                If k > m * n then a complete bipartite graph is returned.
452: (4)                This graph is a bipartite version of the `G_{nm}` random graph model.
453: (4)                The nodes are assigned the attribute 'bipartite' with the value 0 or 1
454: (4)                to indicate which bipartite set the node belongs to.
455: (4)                This function is not imported in the main namespace.
456: (4)                To use it use nx.bipartite.gnmk_random_graph
457: (4)                """
458: (4)                G = nx.Graph()
459: (4)                G = _add_nodes_with_bipartite_label(G, n, m)
460: (4)                if directed:
461: (8)                    G = nx.DiGraph(G)
462: (4)                G.name = f"bipartite_gnm_random_graph({n},{m},{k})"
463: (4)                if n == 1 or m == 1:
464: (8)                    return G
465: (4)                max_edges = n * m  # max_edges for bipartite networks
466: (4)                if k >= max_edges:  # Maybe we should raise an exception here
467: (8)                    return nx.complete_bipartite_graph(n, m, create_using=G)
468: (4)                top = [n for n, d in G.nodes(data=True) if d["bipartite"] == 0]
469: (4)                bottom = list(set(G) - set(top))
470: (4)                edge_count = 0
```

```
471: (4)                while edge_count < k:
472: (8)                    u = seed.choice(top)
473: (8)                    v = seed.choice(bottom)
474: (8)                    if v in G[u]:
475: (12)                       continue
476: (8)                    else:
477: (12)                       G.add_edge(u, v)
478: (12)                       edge_count += 1
479: (4)                return G
480: (0)            def _add_nodes_with_bipartite_label(G, lena, lenb):
481: (4)                G.add_nodes_from(range(lena + lenb))
482: (4)                b = dict(zip(range(lena), [0] * lena))
483: (4)                b.update(dict(zip(range(lena, lena + lenb), [1] * lenb)))
484: (4)                nx.set_node_attributes(G, b, "bipartite")
485: (4)                return G
```

File 112 - extendability.py:

```
1: (0)              """ Provides a function for computing the extendability of a graph which is
2: (0)              undirected, simple, connected and bipartite and contains at least one perfect
matching."""
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["maximal_extendability"]
6: (0)              @not_implemented_for("directed")
7: (0)              @not_implemented_for("multigraph")
8: (0)              @nx._dispatchable
9: (0)              def maximal_extendability(G):
10: (4)                 """Computes the extendability of a graph.
11: (4)                 The extendability of a graph is defined as the maximum $k$ for which `G`
12: (4)                 is $k$-extendable. Graph `G` is $k$-extendable if and only if `G` has a
13: (4)                 perfect matching and every set of $k$ independent edges can be extended
14: (4)                 to a perfect matching in `G`.
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 G : NetworkX Graph
18: (8)                     A fully-connected bipartite graph without self-loops
19: (4)                 Returns
20: (4)                 -------
21: (4)                 extendability : int
22: (4)                 Raises
23: (4)                 ------
24: (4)                 NetworkXError
25: (7)                    If the graph `G` is disconnected.
26: (7)                    If the graph `G` is not bipartite.
27: (7)                    If the graph `G` does not contain a perfect matching.
28: (7)                    If the residual graph of `G` is not strongly connected.
29: (4)                 Notes
30: (4)                 -----
31: (4)                 Definition:
32: (4)                 Let `G` be a simple, connected, undirected and bipartite graph with a
perfect
33: (4)                 matching M and bipartition (U,V). The residual graph of `G`, denoted by
$G_M$,
34: (4)                 is the graph obtained from G by directing the edges of M from V to U and
the
35: (4)                 edges that do not belong to M from U to V.
36: (4)                 Lemma [1]_ :
37: (4)                 Let M be a perfect matching of `G`. `G` is $k$-extendable if and only if
its residual
38: (4)                 graph $G_M$ is strongly connected and there are $k$ vertex-disjoint
directed
39: (4)                 paths between every vertex of U and every vertex of V.
40: (4)                 Assuming that input graph `G` is undirected, simple, connected, bipartite
and contains
41: (4)                 a perfect matching M, this function constructs the residual graph $G_M$ of
G and
```

```
42: (4)                 returns the minimum value among the maximum vertex-disjoint directed paths
between
43: (4)                 every vertex of U and every vertex of V in $G_M$. By combining the
definitions
44: (4)                 and the lemma, this value represents the extendability of the graph `G`.
45: (4)                 Time complexity O($n^3$ $m^2$)) where $n$ is the number of vertices
46: (4)                 and $m$ is the number of edges.
47: (4)                 References
48: (4)                 ----------
49: (4)                 .. [1] "A polynomial algorithm for the extendability problem in bipartite
graphs",
50: (10)                      J. Lakhal, L. Litzler, Information Processing Letters, 1998.
51: (4)                 .. [2] "On n-extendible graphs", M. D. Plummer, Discrete Mathematics,
31:201–210, 1980
52: (10)                      https://doi.org/10.1016/0012-365X(80)90037-0
53: (4)                 """
54: (4)                 if not nx.is_connected(G):
55: (8)                     raise nx.NetworkXError("Graph G is not connected")
56: (4)                 if not nx.bipartite.is_bipartite(G):
57: (8)                     raise nx.NetworkXError("Graph G is not bipartite")
58: (4)                 U, V = nx.bipartite.sets(G)
59: (4)                 maximum_matching = nx.bipartite.hopcroft_karp_matching(G)
60: (4)                 if not nx.is_perfect_matching(G, maximum_matching):
61: (8)                     raise nx.NetworkXError("Graph G does not contain a perfect matching")
62: (4)                 pm = [(node, maximum_matching[node]) for node in V &
maximum_matching.keys()]
63: (4)                 directed_edges = [
64: (8)                     (x, y) if (x in V and (x, y) in pm) or (x in U and (y, x) not in pm)
else (y, x)
65: (8)                     for x, y in G.edges
66: (4)                 ]
67: (4)                 residual_G = nx.DiGraph()
68: (4)                 residual_G.add_nodes_from(G)
69: (4)                 residual_G.add_edges_from(directed_edges)
70: (4)                 if not nx.is_strongly_connected(residual_G):
71: (8)                     raise nx.NetworkXError("The residual graph of G is not strongly
connected")
72: (4)                 k = float("inf")
73: (4)                 for u in U:
74: (8)                     for v in V:
75: (12)                        num_paths = sum(1 for _ in nx.node_disjoint_paths(residual_G, u,
v))
76: (12)                        k = k if k < num_paths else num_paths
77: (4)                 return k
```

File 113 - matrix.py:

```
1: (0)              """
2: (0)              ====================
3: (0)              Biadjacency matrices
4: (0)              ====================
5: (0)              """
6: (0)              import itertools
7: (0)              import networkx as nx
8: (0)              from networkx.convert_matrix import _generate_weighted_edges
9: (0)              __all__ = ["biadjacency_matrix", "from_biadjacency_matrix"]
10: (0)             @nx._dispatchable(edge_attrs="weight")
11: (0)             def biadjacency_matrix(
12: (4)                 G, row_order, column_order=None, dtype=None, weight="weight", format="csr"
13: (0)             ):
14: (4)                 r"""Returns the biadjacency matrix of the bipartite graph G.
15: (4)                 Let `G = (U, V, E)` be a bipartite graph with node sets
16: (4)                 `U = u_{1},...,u_{r}` and `V = v_{1},...,v_{s}`. The biadjacency
17: (4)                 matrix [1]_ is the `r` x `s` matrix `B` in which `b_{i,j} = 1`
18: (4)                 if, and only if, `(u_i, v_j) \in E`. If the parameter `weight` is
19: (4)                 not `None` and matches the name of an edge attribute, its value is
20: (4)                 used instead of 1.
```

```
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph
(4)                 row_order : list of nodes
(7)                    The rows of the matrix are ordered according to the list of nodes.
(4)                 column_order : list, optional
(7)                    The columns of the matrix are ordered according to the list of nodes.
(7)                    If column_order is None, then the ordering of columns is arbitrary.
(4)                 dtype : NumPy data-type, optional
(8)                     A valid NumPy dtype used to initialize the array. If None, then the
(8)                     NumPy default is used.
(4)                 weight : string or None, optional (default='weight')
(7)                    The edge data key used to provide each value in the matrix.
(7)                    If None, then each edge has weight 1.
(4)                 format : str in {'bsr', 'csr', 'csc', 'coo', 'lil', 'dia', 'dok'}
(8)                     The type of the matrix to be returned (default 'csr').  For
(8)                     some algorithms different implementations of sparse matrices
(8)                     can perform better.  See [2]_ for details.
(4)                 Returns
(4)                 -------
(4)                 M : SciPy sparse array
(8)                     Biadjacency matrix representation of the bipartite graph G.
(4)                 Notes
(4)                 -----
(4)                 No attempt is made to check that the input graph is bipartite.
(4)                 For directed bipartite graphs only successors are considered as neighbors.
(4)                 To obtain an adjacency matrix with ones (or weight values) for both
(4)                 predecessors and successors you have to generate two biadjacency matrices
(4)                 where the rows of one of them are the columns of the other, and then add
(4)                 one to the transpose of the other.
(4)                 See Also
(4)                 --------
(4)                 adjacency_matrix
(4)                 from_biadjacency_matrix
(4)                 References
(4)                 ----------
(4)                 .. [1]
https://en.wikipedia.org/wiki/Adjacency_matrix#Adjacency_matrix_of_a_bipartite_graph
(4)                 .. [2] Scipy Dev. References, "Sparse Matrices",
(7)                    https://docs.scipy.org/doc/scipy/reference/sparse.html
(4)                 """
(4)                 import scipy as sp
(4)                 nlen = len(row_order)
(4)                 if nlen == 0:
(8)                     raise nx.NetworkXError("row_order is empty list")
(4)                 if len(row_order) != len(set(row_order)):
(8)                     msg = "Ambiguous ordering: `row_order` contained duplicates."
(8)                     raise nx.NetworkXError(msg)
(4)                 if column_order is None:
(8)                     column_order = list(set(G) - set(row_order))
(4)                 mlen = len(column_order)
(4)                 if len(column_order) != len(set(column_order)):
(8)                     msg = "Ambiguous ordering: `column_order` contained duplicates."
(8)                     raise nx.NetworkXError(msg)
(4)                 row_index = dict(zip(row_order, itertools.count()))
(4)                 col_index = dict(zip(column_order, itertools.count()))
(4)                 if G.number_of_edges() == 0:
(8)                     row, col, data = [], [], []
(4)                 else:
(8)                     row, col, data = zip(
(12)                        *(
(16)                            (row_index[u], col_index[v], d.get(weight, 1))
(16)                            for u, v, d in G.edges(row_order, data=True)
(16)                            if u in row_index and v in col_index
(12)                        )
(8)                     )
(4)                 A = sp.sparse.coo_array((data, (row, col)), shape=(nlen, mlen),
dtype=dtype)
```

```
88: (4)                 try:
89: (8)                     return A.asformat(format)
90: (4)                 except ValueError as err:
91: (8)                     raise nx.NetworkXError(f"Unknown sparse array format: {format}") from
err
92: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
93: (0)             def from_biadjacency_matrix(A, create_using=None, edge_attribute="weight"):
94: (4)                 r"""Creates a new bipartite graph from a biadjacency matrix given as a
95: (4)                 SciPy sparse array.
96: (4)                 Parameters
97: (4)                 ----------
98: (4)                 A: scipy sparse array
99: (6)                   A biadjacency matrix representation of a graph
100: (4)                create_using: NetworkX graph
101: (7)                   Use specified graph for result.  The default is Graph()
102: (4)                edge_attribute: string
103: (7)                   Name of edge attribute to store matrix numeric value. The data will
104: (7)                   have the same type as the matrix entry (int, float, (real,imag)).
105: (4)                Notes
106: (4)                -----
107: (4)                The nodes are labeled with the attribute `bipartite` set to an integer
108: (4)                0 or 1 representing membership in part 0 or part 1 of the bipartite graph.
109: (4)                If `create_using` is an instance of :class:`networkx.MultiGraph` or
110: (4)                :class:`networkx.MultiDiGraph` and the entries of `A` are of
111: (4)                type :class:`int`, then this function returns a multigraph (of the same
112: (4)                type as `create_using`) with parallel edges. In this case,
`edge_attribute`
113: (4)                will be ignored.
114: (4)                See Also
115: (4)                --------
116: (4)                biadjacency_matrix
117: (4)                from_numpy_array
118: (4)                References
119: (4)                ----------
120: (4)                [1]
https://en.wikipedia.org/wiki/Adjacency_matrix#Adjacency_matrix_of_a_bipartite_graph
121: (4)                """
122: (4)                G = nx.empty_graph(0, create_using)
123: (4)                n, m = A.shape
124: (4)                G.add_nodes_from(range(n), bipartite=0)
125: (4)                G.add_nodes_from(range(n, n + m), bipartite=1)
126: (4)                triples = ((u, n + v, d) for (u, v, d) in _generate_weighted_edges(A))
127: (4)                if A.dtype.kind in ("i", "u") and G.is_multigraph():
128: (8)                    chain = itertools.chain.from_iterable
129: (8)                    triples = chain(((u, v, 1) for d in range(w)) for (u, v, w) in
triples)
130: (4)                G.add_weighted_edges_from(triples, weight=edge_attribute)
131: (4)                return G
```

File 114 - matching.py:

```
1: (0)              """Provides functions for computing maximum cardinality matchings and minimum
2: (0)              weight full matchings in a bipartite graph.
3: (0)              If you don't care about the particular implementation of the maximum matching
4: (0)              algorithm, simply use the :func:`maximum_matching`. If you do care, you can
5: (0)              import one of the named maximum matching algorithms directly.
6: (0)              For example, to find a maximum matching in the complete bipartite graph with
7: (0)              two vertices on the left and three vertices on the right:
8: (0)              >>> G = nx.complete_bipartite_graph(2, 3)
9: (0)              >>> left, right = nx.bipartite.sets(G)
10: (0)             >>> list(left)
11: (0)             [0, 1]
12: (0)             >>> list(right)
13: (0)             [2, 3, 4]
14: (0)             >>> nx.bipartite.maximum_matching(G)
15: (0)             {0: 2, 1: 3, 2: 0, 3: 1}
16: (0)             The dictionary returned by :func:`maximum_matching` includes a mapping for
```

```
vertices in both the left and right vertex sets.
Similarly, :func:`minimum_weight_full_matching` produces, for a complete
weighted bipartite graph, a matching whose cardinality is the cardinality of
the smaller of the two partitions, and for which the sum of the weights of the
edges included in the matching is minimal.
"""
import collections
import itertools
import networkx as nx
from networkx.algorithms.bipartite import sets as bipartite_sets
from networkx.algorithms.bipartite.matrix import biadjacency_matrix
__all__ = [
    "maximum_matching",
    "hopcroft_karp_matching",
    "eppstein_matching",
    "to_vertex_cover",
    "minimum_weight_full_matching",
]
INFINITY = float("inf")
@nx._dispatchable
def hopcroft_karp_matching(G, top_nodes=None):
    """Returns the maximum cardinality matching of the bipartite graph `G`.
    A matching is a set of edges that do not share any nodes. A maximum
    cardinality matching is a matching with the most edges possible. It
    is not always unique. Finding a matching in a bipartite graph can be
    treated as a networkx flow problem.
    The functions ``hopcroft_karp_matching`` and ``maximum_matching``
    are aliases of the same function.
    Parameters
    ----------
    G : NetworkX graph
      Undirected bipartite graph
    top_nodes : container of nodes
      Container with all nodes in one bipartite node set. If not supplied
      it will be computed. But if more than one solution exists an exception
      will be raised.
    Returns
    -------
    matches : dictionary
      The matching is returned as a dictionary, `matches`, such that
      ``matches[v] == w`` if node `v` is matched to node `w`. Unmatched
      nodes do not occur as a key in `matches`.
    Raises
    ------
    AmbiguousSolution
      Raised if the input bipartite graph is disconnected and no container
      with all nodes in one bipartite set is provided. When determining
      the nodes in each bipartite set more than one valid solution is
      possible if the input graph is disconnected.
    Notes
    -----
    This function is implemented with the `Hopcroft--Karp matching algorithm
    <https://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm>`_ for
    bipartite graphs.
    See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
    for further details on how bipartite graphs are handled in NetworkX.
    See Also
    --------
    maximum_matching
    hopcroft_karp_matching
    eppstein_matching
    References
    ----------
    .. [1] John E. Hopcroft and Richard M. Karp. "An n^{5 / 2} Algorithm for
       Maximum Matchings in Bipartite Graphs" In: **SIAM Journal of Computing**
       2.4 (1973), pp. 225--231. <https://doi.org/10.1137/0202019>.
    """
    def breadth_first_search():
```

```
 85: (8)                    for v in left:
 86: (12)                       if leftmatches[v] is None:
 87: (16)                           distances[v] = 0
 88: (16)                           queue.append(v)
 89: (12)                       else:
 90: (16)                           distances[v] = INFINITY
 91: (8)                    distances[None] = INFINITY
 92: (8)                    while queue:
 93: (12)                       v = queue.popleft()
 94: (12)                       if distances[v] < distances[None]:
 95: (16)                           for u in G[v]:
 96: (20)                               if distances[rightmatches[u]] is INFINITY:
 97: (24)                                   distances[rightmatches[u]] = distances[v] + 1
 98: (24)                                   queue.append(rightmatches[u])
 99: (8)                    return distances[None] is not INFINITY
 100: (4)               def depth_first_search(v):
 101: (8)                   if v is not None:
 102: (12)                      for u in G[v]:
 103: (16)                          if distances[rightmatches[u]] == distances[v] + 1:
 104: (20)                              if depth_first_search(rightmatches[u]):
 105: (24)                                  rightmatches[u] = v
 106: (24)                                  leftmatches[v] = u
 107: (24)                                  return True
 108: (12)                      distances[v] = INFINITY
 109: (12)                      return False
 110: (8)                   return True
 111: (4)               left, right = bipartite_sets(G, top_nodes)
 112: (4)               leftmatches = {v: None for v in left}
 113: (4)               rightmatches = {v: None for v in right}
 114: (4)               distances = {}
 115: (4)               queue = collections.deque()
 116: (4)               num_matched_pairs = 0
 117: (4)               while breadth_first_search():
 118: (8)                   for v in left:
 119: (12)                      if leftmatches[v] is None:
 120: (16)                          if depth_first_search(v):
 121: (20)                              num_matched_pairs += 1
 122: (4)               leftmatches = {k: v for k, v in leftmatches.items() if v is not None}
 123: (4)               rightmatches = {k: v for k, v in rightmatches.items() if v is not None}
 124: (4)               return dict(itertools.chain(leftmatches.items(), rightmatches.items()))
 125: (0)           @nx._dispatchable
 126: (0)           def eppstein_matching(G, top_nodes=None):
 127: (4)               """Returns the maximum cardinality matching of the bipartite graph `G`.
 128: (4)               Parameters
 129: (4)               ----------
 130: (4)               G : NetworkX graph
 131: (6)                 Undirected bipartite graph
 132: (4)               top_nodes : container
 133: (6)                 Container with all nodes in one bipartite node set. If not supplied
 134: (6)                 it will be computed. But if more than one solution exists an exception
 135: (6)                 will be raised.
 136: (4)               Returns
 137: (4)               -------
 138: (4)               matches : dictionary
 139: (6)                 The matching is returned as a dictionary, `matching`, such that
 140: (6)                 ``matching[v] == w`` if node `v` is matched to node `w`. Unmatched
 141: (6)                 nodes do not occur as a key in `matching`.
 142: (4)               Raises
 143: (4)               ------
 144: (4)               AmbiguousSolution
 145: (6)                 Raised if the input bipartite graph is disconnected and no container
 146: (6)                 with all nodes in one bipartite set is provided. When determining
 147: (6)                 the nodes in each bipartite set more than one valid solution is
 148: (6)                 possible if the input graph is disconnected.
 149: (4)               Notes
 150: (4)               -----
 151: (4)               This function is implemented with David Eppstein's version of the
algorithm
 152: (4)               Hopcroft--Karp algorithm (see :func:`hopcroft_karp_matching`), which
```

```
(4)                 originally appeared in the `Python Algorithms and Data Structures library
(4)                 (PADS) <http://www.ics.uci.edu/~eppstein/PADS/ABOUT-PADS.txt>`_.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 See Also
(4)                 --------
(4)                 hopcroft_karp_matching
(4)                 """
(4)                 left, right = bipartite_sets(G, top_nodes)
(4)                 G = nx.DiGraph(G.edges(left))
(4)                 matching = {}
(4)                 for u in G:
(8)                     for v in G[u]:
(12)                        if v not in matching:
(16)                            matching[v] = u
(16)                            break
(4)                 while True:
(8)                     preds = {}
(8)                     unmatched = []
(8)                     pred = {u: unmatched for u in G}
(8)                     for v in matching:
(12)                        del pred[matching[v]]
(8)                     layer = list(pred)
(8)                     while layer and not unmatched:
(12)                        newLayer = {}
(12)                        for u in layer:
(16)                            for v in G[u]:
(20)                                if v not in preds:
(24)                                    newLayer.setdefault(v, []).append(u)
(12)                        layer = []
(12)                        for v in newLayer:
(16)                            preds[v] = newLayer[v]
(16)                            if v in matching:
(20)                                layer.append(matching[v])
(20)                                pred[matching[v]] = v
(16)                            else:
(20)                                unmatched.append(v)
(8)                     if not unmatched:
(12)                        for key in matching.copy():
(16)                            matching[matching[key]] = key
(12)                        return matching
(8)                     def recurse(v):
(12)                        if v in preds:
(16)                            L = preds.pop(v)
(16)                            for u in L:
(20)                                if u in pred:
(24)                                    pu = pred.pop(u)
(24)                                    if pu is unmatched or recurse(pu):
(28)                                        matching[v] = u
(28)                                        return True
(12)                        return False
(8)                     for v in unmatched:
(12)                        recurse(v)
(0)             def _is_connected_by_alternating_path(G, v, matched_edges, unmatched_edges,
targets):
(4)                 """Returns True if and only if the vertex `v` is connected to one of
(4)                 the target vertices by an alternating path in `G`.
(4)                 An *alternating path* is a path in which every other edge is in the
(4)                 specified maximum matching (and the remaining edges in the path are not in
(4)                 the matching). An alternating path may have matched edges in the even
(4)                 positions or in the odd positions, as long as the edges alternate between
(4)                 'matched' and 'unmatched'.
(4)                 `G` is an undirected bipartite NetworkX graph.
(4)                 `v` is a vertex in `G`.
(4)                 `matched_edges` is a set of edges present in a maximum matching in `G`.
(4)                 `unmatched_edges` is a set of edges not present in a maximum
(4)                 matching in `G`.
(4)                 `targets` is a set of vertices.
(4)                 """
```

```
(4)                 def _alternating_dfs(u, along_matched=True):
(8)                     """Returns True if and only if `u` is connected to one of the
(8)                     targets by an alternating path.
(8)                     `u` is a vertex in the graph `G`.
(8)                     If `along_matched` is True, this step of the depth-first search
(8)                     will continue only through edges in the given matching. Otherwise, it
(8)                     will continue only through edges *not* in the given matching.
(8)                     """
(8)                     visited = set()
(8)                     initial_depth = 0 if along_matched else 1
(8)                     stack = [(u, iter(G[u]), initial_depth)]
(8)                     while stack:
(12)                        parent, children, depth = stack[-1]
(12)                        valid_edges = matched_edges if depth % 2 else unmatched_edges
(12)                        try:
(16)                            child = next(children)
(16)                            if child not in visited:
(20)                                if (parent, child) in valid_edges or (child, parent) in
valid_edges:
(24)                                    if child in targets:
(28)                                        return True
(24)                                    visited.add(child)
(24)                                    stack.append((child, iter(G[child]), depth + 1))
(12)                        except StopIteration:
(16)                            stack.pop()
(8)                     return False
(4)                 return _alternating_dfs(v, along_matched=True) or _alternating_dfs(
(8)                     v, along_matched=False
(4)                 )
(0)             def _connected_by_alternating_paths(G, matching, targets):
(4)                 """Returns the set of vertices that are connected to one of the target
(4)                 vertices by an alternating path in `G` or are themselves a target.
(4)                 An *alternating path* is a path in which every other edge is in the
(4)                 specified maximum matching (and the remaining edges in the path are not in
(4)                 the matching). An alternating path may have matched edges in the even
(4)                 positions or in the odd positions, as long as the edges alternate between
(4)                 'matched' and 'unmatched'.
(4)                 `G` is an undirected bipartite NetworkX graph.
(4)                 `matching` is a dictionary representing a maximum matching in `G`, as
(4)                 returned by, for example, :func:`maximum_matching`.
(4)                 `targets` is a set of vertices.
(4)                 """
(4)                 edge_sets = {frozenset((u, v)) for u, v in matching.items()}
(4)                 matched_edges = {tuple(edge) for edge in edge_sets}
(4)                 unmatched_edges = {
(8)                     (u, v) for (u, v) in G.edges() if frozenset((u, v)) not in edge_sets
(4)                 }
(4)                 return {
(8)                     v
(8)                     for v in G
(8)                     if v in targets
(8)                     or _is_connected_by_alternating_path(
(12)                        G, v, matched_edges, unmatched_edges, targets
(8)                     )
(4)                 }
(0)             @nx._dispatchable
(0)             def to_vertex_cover(G, matching, top_nodes=None):
(4)                 """Returns the minimum vertex cover corresponding to the given maximum
(4)                 matching of the bipartite graph `G`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(6)                   Undirected bipartite graph
(4)                 matching : dictionary
(6)                   A dictionary whose keys are vertices in `G` and whose values are the
(6)                   distinct neighbors comprising the maximum matching for `G`, as returned
(6)                   by, for example, :func:`maximum_matching`. The dictionary *must*
(6)                   represent the maximum matching.
(4)                 top_nodes : container
```

```
289: (6)                  Container with all nodes in one bipartite node set. If not supplied
290: (6)                  it will be computed. But if more than one solution exists an exception
291: (6)                  will be raised.
292: (4)                Returns
293: (4)                -------
294: (4)                vertex_cover : :class:`set`
295: (6)                  The minimum vertex cover in `G`.
296: (4)                Raises
297: (4)                ------
298: (4)                AmbiguousSolution
299: (6)                  Raised if the input bipartite graph is disconnected and no container
300: (6)                  with all nodes in one bipartite set is provided. When determining
301: (6)                  the nodes in each bipartite set more than one valid solution is
302: (6)                  possible if the input graph is disconnected.
303: (4)                Notes
304: (4)                -----
305: (4)                This function is implemented using the procedure guaranteed by `Konig's
306: (4)                theorem
307: (4)
<https://en.wikipedia.org/wiki/K%C3%B6nig%27s_theorem_%28graph_theory%29>`_,
308: (4)                which proves an equivalence between a maximum matching and a minimum
vertex
309: (4)                cover in bipartite graphs.
310: (4)                Since a minimum vertex cover is the complement of a maximum independent
set
311: (4)                for any graph, one can compute the maximum independent set of a bipartite
312: (4)                graph this way:
313: (4)                >>> G = nx.complete_bipartite_graph(2, 3)
314: (4)                >>> matching = nx.bipartite.maximum_matching(G)
315: (4)                >>> vertex_cover = nx.bipartite.to_vertex_cover(G, matching)
316: (4)                >>> independent_set = set(G) - vertex_cover
317: (4)                >>> print(list(independent_set))
318: (4)                [2, 3, 4]
319: (4)                See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
320: (4)                for further details on how bipartite graphs are handled in NetworkX.
321: (4)                """
322: (4)                L, R = bipartite_sets(G, top_nodes)
323: (4)                unmatched_vertices = set(G) - set(matching)
324: (4)                U = unmatched_vertices & L
325: (4)                Z = _connected_by_alternating_paths(G, matching, U)
326: (4)                return (L - Z) | (R & Z)
327: (0)            maximum_matching = hopcroft_karp_matching
328: (0)            @nx._dispatchable(edge_attrs="weight")
329: (0)            def minimum_weight_full_matching(G, top_nodes=None, weight="weight"):
330: (4)                r"""Returns a minimum weight full matching of the bipartite graph `G`.
331: (4)                Let :math:`G = ((U, V), E)` be a weighted bipartite graph with real
weights
332: (4)                :math:`w : E \to \mathbb{R}`. This function then produces a matching
333: (4)                :math:`M \subseteq E` with cardinality
334: (4)                .. math::
335: (7)                   \lvert M \rvert = \min(\lvert U \rvert, \lvert V \rvert),
336: (4)                which minimizes the sum of the weights of the edges included in the
337: (4)                matching, :math:`\sum_{e \in M} w(e)`, or raises an error if no such
338: (4)                matching exists.
339: (4)                When :math:`\lvert U \rvert = \lvert V \rvert`, this is commonly
340: (4)                referred to as a perfect matching; here, since we allow
341: (4)                :math:`\lvert U \rvert` and :math:`\lvert V \rvert` to differ, we
342: (4)                follow Karp [1]_ and refer to the matching as *full*.
343: (4)                Parameters
344: (4)                ----------
345: (4)                G : NetworkX graph
346: (6)                  Undirected bipartite graph
347: (4)                top_nodes : container
348: (6)                  Container with all nodes in one bipartite node set. If not supplied
349: (6)                  it will be computed.
350: (4)                weight : string, optional (default='weight')
351: (7)                   The edge data key used to provide each value in the matrix.
352: (7)                   If None, then each edge has weight 1.
353: (4)                Returns
```

```
354: (4)                -------
355: (4)                matches : dictionary
356: (6)                  The matching is returned as a dictionary, `matches`, such that
357: (6)                  ``matches[v] == w`` if node `v` is matched to node `w`. Unmatched
358: (6)                  nodes do not occur as a key in `matches`.
359: (4)                Raises
360: (4)                ------
361: (4)                ValueError
362: (6)                  Raised if no full matching exists.
363: (4)                ImportError
364: (6)                  Raised if SciPy is not available.
365: (4)                Notes
366: (4)                -----
367: (4)                The problem of determining a minimum weight full matching is also known as
368: (4)                the rectangular linear assignment problem. This implementation defers the
369: (4)                calculation of the assignment to SciPy.
370: (4)                References
371: (4)                ----------
372: (4)                .. [1] Richard Manning Karp:
373: (7)                   An algorithm to Solve the m x n Assignment Problem in Expected Time
374: (7)                   O(mn log n).
375: (7)                   Networks, 10(2):143–152, 1980.
376: (4)                """
377: (4)                import numpy as np
378: (4)                import scipy as sp
379: (4)                left, right = nx.bipartite.sets(G, top_nodes)
380: (4)                U = list(left)
381: (4)                V = list(right)
382: (4)                weights_sparse = biadjacency_matrix(
383: (8)                    G, row_order=U, column_order=V, weight=weight, format="coo"
384: (4)                )
385: (4)                weights = np.full(weights_sparse.shape, np.inf)
386: (4)                weights[weights_sparse.row, weights_sparse.col] = weights_sparse.data
387: (4)                left_matches = sp.optimize.linear_sum_assignment(weights)
388: (4)                d = {U[u]: V[v] for u, v in zip(*left_matches)}
389: (4)                d.update({v: u for u, v in d.items()})
390: (4)                return d

----------------------------------------

File 115 - projection.py:

1: (0)              """One-mode (unipartite) projections of bipartite graphs."""
2: (0)              import networkx as nx
3: (0)              from networkx.exception import NetworkXAlgorithmError
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = [
6: (4)                  "projected_graph",
7: (4)                  "weighted_projected_graph",
8: (4)                  "collaboration_weighted_projected_graph",
9: (4)                  "overlap_weighted_projected_graph",
10: (4)                 "generic_weighted_projected_graph",
11: (0)             ]
12: (0)             @nx._dispatchable(
13: (4)                 graphs="B", preserve_node_attrs=True, preserve_graph_attrs=True,
returns_graph=True
14: (0)             )
15: (0)             def projected_graph(B, nodes, multigraph=False):
16: (4)                 r"""Returns the projection of B onto one of its node sets.
17: (4)                 Returns the graph G that is the projection of the bipartite graph B
18: (4)                 onto the specified nodes. They retain their attributes and are connected
19: (4)                 in G if they have a common neighbor in B.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 B : NetworkX graph
23: (6)                   The input graph should be bipartite.
24: (4)                 nodes : list or iterable
25: (6)                   Nodes to project onto (the "bottom" nodes).
26: (4)                 multigraph: bool (default=False)
```

```
   If True return a multigraph where the multiple edges represent multiple
   shared neighbors.  They edge key in the multigraph is assigned to the
   label of the neighbor.
Returns
-------
Graph : NetworkX graph or multigraph
   A graph that is the projection onto the given nodes.
Examples
--------
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(4)
>>> G = bipartite.projected_graph(B, [1, 3])
>>> list(G)
[1, 3]
>>> list(G.edges())
[(1, 3)]
If nodes `a`, and `b` are connected through both nodes 1 and 2 then
building a multigraph results in two edges in the projection onto
[`a`, `b`]:
>>> B = nx.Graph()
>>> B.add_edges_from([("a", 1), ("b", 1), ("a", 2), ("b", 2)])
>>> G = bipartite.projected_graph(B, ["a", "b"], multigraph=True)
>>> print([sorted((u, v)) for u, v in G.edges()])
[['a', 'b'], ['a', 'b']]
Notes
-----
No attempt is made to verify that the input graph B is bipartite.
Returns a simple graph that is the projection of the bipartite graph B
onto the set of nodes given in list nodes.  If multigraph=True then
a multigraph is returned with an edge for every shared neighbor.
Directed graphs are allowed as input.  The output will also then
be a directed graph with edges if there is a directed path between
the nodes.
The graph and node properties are (shallow) copied to the projected graph.
See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
for further details on how bipartite graphs are handled in NetworkX.
See Also
--------
is_bipartite,
is_bipartite_node_set,
sets,
weighted_projected_graph,
collaboration_weighted_projected_graph,
overlap_weighted_projected_graph,
generic_weighted_projected_graph
"""
if B.is_multigraph():
    raise nx.NetworkXError("not defined for multigraphs")
if B.is_directed():
    directed = True
    if multigraph:
        G = nx.MultiDiGraph()
    else:
        G = nx.DiGraph()
else:
    directed = False
    if multigraph:
        G = nx.MultiGraph()
    else:
        G = nx.Graph()
G.graph.update(B.graph)
G.add_nodes_from((n, B.nodes[n]) for n in nodes)
for u in nodes:
    nbrs2 = {v for nbr in B[u] for v in B[nbr] if v != u}
    if multigraph:
        for n in nbrs2:
            if directed:
                links = set(B[u]) & set(B.pred[n])
            else:
```

```
96: (20)                                links = set(B[u]) & set(B[n])
97: (16)                            for l in links:
98: (20)                                if not G.has_edge(u, n, l):
99: (24)                                    G.add_edge(u, n, key=l)
100: (8)                    else:
101: (12)                       G.add_edges_from((u, n) for n in nbrs2)
102: (4)                return G
103: (0)            @not_implemented_for("multigraph")
104: (0)            @nx._dispatchable(graphs="B", returns_graph=True)
105: (0)            def weighted_projected_graph(B, nodes, ratio=False):
106: (4)                r"""Returns a weighted projection of B onto one of its node sets.
107: (4)                The weighted projected graph is the projection of the bipartite
108: (4)                network B onto the specified nodes with weights representing the
109: (4)                number of shared neighbors or the ratio between actual shared
110: (4)                neighbors and possible shared neighbors if ``ratio is True`` [1]_.
111: (4)                The nodes retain their attributes and are connected in the resulting
112: (4)                graph if they have an edge to a common node in the original graph.
113: (4)                Parameters
114: (4)                ----------
115: (4)                B : NetworkX graph
116: (8)                    The input graph should be bipartite.
117: (4)                nodes : list or iterable
118: (8)                    Distinct nodes to project onto (the "bottom" nodes).
119: (4)                ratio: Bool (default=False)
120: (8)                    If True, edge weight is the ratio between actual shared neighbors
121: (8)                    and maximum possible shared neighbors (i.e., the size of the other
122: (8)                    node set). If False, edges weight is the number of shared neighbors.
123: (4)                Returns
124: (4)                -------
125: (4)                Graph : NetworkX graph
126: (7)                   A graph that is the projection onto the given nodes.
127: (4)                Examples
128: (4)                --------
129: (4)                >>> from networkx.algorithms import bipartite
130: (4)                >>> B = nx.path_graph(4)
131: (4)                >>> G = bipartite.weighted_projected_graph(B, [1, 3])
132: (4)                >>> list(G)
133: (4)                [1, 3]
134: (4)                >>> list(G.edges(data=True))
135: (4)                [(1, 3, {'weight': 1})]
136: (4)                >>> G = bipartite.weighted_projected_graph(B, [1, 3], ratio=True)
137: (4)                >>> list(G.edges(data=True))
138: (4)                [(1, 3, {'weight': 0.5})]
139: (4)                Notes
140: (4)                -----
141: (4)                No attempt is made to verify that the input graph B is bipartite, or that
142: (4)                the input nodes are distinct. However, if the length of the input nodes is
143: (4)                greater than or equal to the nodes in the graph B, an exception is raised.
144: (4)                If the nodes are not distinct but don't raise this error, the output
weights
145: (4)                will be incorrect.
146: (4)                The graph and node properties are (shallow) copied to the projected graph.
147: (4)                See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
148: (4)                for further details on how bipartite graphs are handled in NetworkX.
149: (4)                See Also
150: (4)                --------
151: (4)                is_bipartite,
152: (4)                is_bipartite_node_set,
153: (4)                sets,
154: (4)                collaboration_weighted_projected_graph,
155: (4)                overlap_weighted_projected_graph,
156: (4)                generic_weighted_projected_graph
157: (4)                projected_graph
158: (4)                References
159: (4)                ----------
160: (4)                .. [1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation
161: (8)                    Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook
162: (8)                    of Social Network Analysis. Sage Publications.
163: (4)                """
```

```
(4)                 if B.is_directed():
(8)                     pred = B.pred
(8)                     G = nx.DiGraph()
(4)                 else:
(8)                     pred = B.adj
(8)                     G = nx.Graph()
(4)                 G.graph.update(B.graph)
(4)                 G.add_nodes_from((n, B.nodes[n]) for n in nodes)
(4)                 n_top = len(B) - len(nodes)
(4)                 if n_top < 1:
(8)                     raise NetworkXAlgorithmError(
(12)                        f"the size of the nodes to project onto ({len(nodes)}) is >= the graph size ({len(B)}).\n"
(12)                        "They are either not a valid bipartite partition or contain duplicates"
(8)                     )
(4)                 for u in nodes:
(8)                     unbrs = set(B[u])
(8)                     nbrs2 = {n for nbr in unbrs for n in B[nbr]} - {u}
(8)                     for v in nbrs2:
(12)                        vnbrs = set(pred[v])
(12)                        common = unbrs & vnbrs
(12)                        if not ratio:
(16)                            weight = len(common)
(12)                        else:
(16)                            weight = len(common) / n_top
(12)                        G.add_edge(u, v, weight=weight)
(4)                 return G
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(graphs="B", returns_graph=True)
(0)             def collaboration_weighted_projected_graph(B, nodes):
(4)                 r"""Newman's weighted projection of B onto one of its node sets.
(4)                 The collaboration weighted projection is the projection of the
(4)                 bipartite network B onto the specified nodes with weights assigned
(4)                 using Newman's collaboration model [1]_:
(4)                 .. math::
(8)                     w_{u, v} = \sum_k \frac{\delta_{u}^{k} \delta_{v}^{k}}{d_k - 1}
(4)                 where `u` and `v` are nodes from the bottom bipartite node set,
(4)                 and `k` is a node of the top node set.
(4)                 The value `d_k` is the degree of node `k` in the bipartite
(4)                 network and `\delta_{u}^{k}` is 1 if node `u` is
(4)                 linked to node `k` in the original bipartite graph or 0 otherwise.
(4)                 The nodes retain their attributes and are connected in the resulting
(4)                 graph if have an edge to a common node in the original bipartite
(4)                 graph.
(4)                 Parameters
(4)                 ----------
(4)                 B : NetworkX graph
(6)                   The input graph should be bipartite.
(4)                 nodes : list or iterable
(6)                   Nodes to project onto (the "bottom" nodes).
(4)                 Returns
(4)                 -------
(4)                 Graph : NetworkX graph
(7)                    A graph that is the projection onto the given nodes.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
(4)                 >>> B = nx.path_graph(5)
(4)                 >>> B.add_edge(1, 5)
(4)                 >>> G = bipartite.collaboration_weighted_projected_graph(B, [0, 2, 4, 5])
(4)                 >>> list(G)
(4)                 [0, 2, 4, 5]
(4)                 >>> for edge in sorted(G.edges(data=True)):
(4)                 ...     print(edge)
(4)                 (0, 2, {'weight': 0.5})
(4)                 (0, 5, {'weight': 0.5})
(4)                 (2, 4, {'weight': 1.0})
(4)                 (2, 5, {'weight': 0.5})
```

```
(4)                 Notes
(4)                 -----
(4)                 No attempt is made to verify that the input graph B is bipartite.
(4)                 The graph and node properties are (shallow) copied to the projected graph.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 See Also
(4)                 --------
(4)                 is_bipartite,
(4)                 is_bipartite_node_set,
(4)                 sets,
(4)                 weighted_projected_graph,
(4)                 overlap_weighted_projected_graph,
(4)                 generic_weighted_projected_graph,
(4)                 projected_graph
(4)                 References
(4)                 ----------
(4)                 .. [1] Scientific collaboration networks: II.
(8)                     Shortest paths, weighted networks, and centrality,
(8)                     M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).
(4)                 """
(4)                 if B.is_directed():
(8)                     pred = B.pred
(8)                     G = nx.DiGraph()
(4)                 else:
(8)                     pred = B.adj
(8)                     G = nx.Graph()
(4)                 G.graph.update(B.graph)
(4)                 G.add_nodes_from((n, B.nodes[n]) for n in nodes)
(4)                 for u in nodes:
(8)                     unbrs = set(B[u])
(8)                     nbrs2 = {n for nbr in unbrs for n in B[nbr] if n != u}
(8)                     for v in nbrs2:
(12)                        vnbrs = set(pred[v])
(12)                        common_degree = (len(B[n]) for n in unbrs & vnbrs)
(12)                        weight = sum(1.0 / (deg - 1) for deg in common_degree if deg > 1)
(12)                        G.add_edge(u, v, weight=weight)
(4)                 return G
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(graphs="B", returns_graph=True)
(0)             def overlap_weighted_projected_graph(B, nodes, jaccard=True):
(4)                 r"""Overlap weighted projection of B onto one of its node sets.
(4)                 The overlap weighted projection is the projection of the bipartite
(4)                 network B onto the specified nodes with weights representing
(4)                 the Jaccard index between the neighborhoods of the two nodes in the
(4)                 original bipartite network [1]_:
(4)                 .. math::
(8)                     w_{v, u} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}
(4)                 or if the parameter 'jaccard' is False, the fraction of common
(4)                 neighbors by minimum of both nodes degree in the original
(4)                 bipartite graph [1]_:
(4)                 .. math::
(8)                     w_{v, u} = \frac{|N(u) \cap N(v)|}{min(|N(u)|, |N(v)|)}
(4)                 The nodes retain their attributes and are connected in the resulting
(4)                 graph if have an edge to a common node in the original bipartite graph.
(4)                 Parameters
(4)                 ----------
(4)                 B : NetworkX graph
(8)                     The input graph should be bipartite.
(4)                 nodes : list or iterable
(8)                     Nodes to project onto (the "bottom" nodes).
(4)                 jaccard: Bool (default=True)
(4)                 Returns
(4)                 -------
(4)                 Graph : NetworkX graph
(7)                    A graph that is the projection onto the given nodes.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import bipartite
```

```
(4)                 >>> B = nx.path_graph(5)
(4)                 >>> nodes = [0, 2, 4]
(4)                 >>> G = bipartite.overlap_weighted_projected_graph(B, nodes)
(4)                 >>> list(G)
(4)                 [0, 2, 4]
(4)                 >>> list(G.edges(data=True))
(4)                 [(0, 2, {'weight': 0.5}), (2, 4, {'weight': 0.5})]
(4)                 >>> G = bipartite.overlap_weighted_projected_graph(B, nodes,
jaccard=False)
(4)                 >>> list(G.edges(data=True))
(4)                 [(0, 2, {'weight': 1.0}), (2, 4, {'weight': 1.0})]
(4)                 Notes
(4)                 -----
(4)                 No attempt is made to verify that the input graph B is bipartite.
(4)                 The graph and node properties are (shallow) copied to the projected graph.
(4)                 See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
(4)                 for further details on how bipartite graphs are handled in NetworkX.
(4)                 See Also
(4)                 --------
(4)                 is_bipartite,
(4)                 is_bipartite_node_set,
(4)                 sets,
(4)                 weighted_projected_graph,
(4)                 collaboration_weighted_projected_graph,
(4)                 generic_weighted_projected_graph,
(4)                 projected_graph
(4)                 References
(4)                 ----------
(4)                 .. [1] Borgatti, S.P. and Halgin, D. In press. Analyzing Affiliation
(8)                     Networks. In Carrington, P. and Scott, J. (eds) The Sage Handbook
(8)                     of Social Network Analysis. Sage Publications.
(4)                 """
(4)                 if B.is_directed():
(8)                     pred = B.pred
(8)                     G = nx.DiGraph()
(4)                 else:
(8)                     pred = B.adj
(8)                     G = nx.Graph()
(4)                 G.graph.update(B.graph)
(4)                 G.add_nodes_from((n, B.nodes[n]) for n in nodes)
(4)                 for u in nodes:
(8)                     unbrs = set(B[u])
(8)                     nbrs2 = {n for nbr in unbrs for n in B[nbr]} - {u}
(8)                     for v in nbrs2:
(12)                        vnbrs = set(pred[v])
(12)                        if jaccard:
(16)                            wt = len(unbrs & vnbrs) / len(unbrs | vnbrs)
(12)                        else:
(16)                            wt = len(unbrs & vnbrs) / min(len(unbrs), len(vnbrs))
(12)                        G.add_edge(u, v, weight=wt)
(4)                 return G
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(graphs="B", preserve_all_attrs=True, returns_graph=True)
(0)             def generic_weighted_projected_graph(B, nodes, weight_function=None):
(4)                 r"""Weighted projection of B with a user-specified weight function.
(4)                 The bipartite network B is projected on to the specified nodes
(4)                 with weights computed by a user-specified function.  This function
(4)                 must accept as a parameter the neighborhood sets of two nodes and
(4)                 return an integer or a float.
(4)                 The nodes retain their attributes and are connected in the resulting graph
(4)                 if they have an edge to a common node in the original graph.
(4)                 Parameters
(4)                 ----------
(4)                 B : NetworkX graph
(8)                     The input graph should be bipartite.
(4)                 nodes : list or iterable
(8)                     Nodes to project onto (the "bottom" nodes).
(4)                 weight_function : function
(8)                     This function must accept as parameters the same input graph
```

```
368: (8)                    that this function, and two nodes; and return an integer or a float.
369: (8)                    The default function computes the number of shared neighbors.
370: (4)                Returns
371: (4)                -------
372: (4)                Graph : NetworkX graph
373: (7)                   A graph that is the projection onto the given nodes.
374: (4)                Examples
375: (4)                --------
376: (4)                >>> from networkx.algorithms import bipartite
377: (4)                >>> # Define some custom weight functions
378: (4)                >>> def jaccard(G, u, v):
379: (4)                ...     unbrs = set(G[u])
380: (4)                ...     vnbrs = set(G[v])
381: (4)                ...     return float(len(unbrs & vnbrs)) / len(unbrs | vnbrs)
382: (4)                >>> def my_weight(G, u, v, weight="weight"):
383: (4)                ...     w = 0
384: (4)                ...     for nbr in set(G[u]) & set(G[v]):
385: (4)                ...         w += G[u][nbr].get(weight, 1) + G[v][nbr].get(weight, 1)
386: (4)                ...     return w
387: (4)                >>> # A complete bipartite graph with 4 nodes and 4 edges
388: (4)                >>> B = nx.complete_bipartite_graph(2, 2)
389: (4)                >>> # Add some arbitrary weight to the edges
390: (4)                >>> for i, (u, v) in enumerate(B.edges()):
391: (4)                ...     B.edges[u, v]["weight"] = i + 1
392: (4)                >>> for edge in B.edges(data=True):
393: (4)                ...     print(edge)
394: (4)                (0, 2, {'weight': 1})
395: (4)                (0, 3, {'weight': 2})
396: (4)                (1, 2, {'weight': 3})
397: (4)                (1, 3, {'weight': 4})
398: (4)                >>> # By default, the weight is the number of shared neighbors
399: (4)                >>> G = bipartite.generic_weighted_projected_graph(B, [0, 1])
400: (4)                >>> print(list(G.edges(data=True)))
401: (4)                [(0, 1, {'weight': 2})]
402: (4)                >>> # To specify a custom weight function use the weight_function
parameter
403: (4)                >>> G = bipartite.generic_weighted_projected_graph(B, [0, 1],
weight_function=jaccard)
404: (4)                >>> print(list(G.edges(data=True)))
405: (4)                [(0, 1, {'weight': 1.0})]
406: (4)                >>> G = bipartite.generic_weighted_projected_graph(B, [0, 1],
weight_function=my_weight)
407: (4)                >>> print(list(G.edges(data=True)))
408: (4)                [(0, 1, {'weight': 10})]
409: (4)                Notes
410: (4)                -----
411: (4)                No attempt is made to verify that the input graph B is bipartite.
412: (4)                The graph and node properties are (shallow) copied to the projected graph.
413: (4)                See :mod:`bipartite documentation <networkx.algorithms.bipartite>`
414: (4)                for further details on how bipartite graphs are handled in NetworkX.
415: (4)                See Also
416: (4)                --------
417: (4)                is_bipartite,
418: (4)                is_bipartite_node_set,
419: (4)                sets,
420: (4)                weighted_projected_graph,
421: (4)                collaboration_weighted_projected_graph,
422: (4)                overlap_weighted_projected_graph,
423: (4)                projected_graph
424: (4)                """
425: (4)                if B.is_directed():
426: (8)                    pred = B.pred
427: (8)                    G = nx.DiGraph()
428: (4)                else:
429: (8)                    pred = B.adj
430: (8)                    G = nx.Graph()
431: (4)                if weight_function is None:
432: (8)                    def weight_function(G, u, v):
433: (12)                       return len(set(G[u]) & set(pred[v]))
```

```
434: (4)                G.graph.update(B.graph)
435: (4)                G.add_nodes_from((n, B.nodes[n]) for n in nodes)
436: (4)                for u in nodes:
437: (8)                    nbrs2 = {n for nbr in set(B[u]) for n in B[nbr]} - {u}
438: (8)                    for v in nbrs2:
439: (12)                       weight = weight_function(B, u, v)
440: (12)                       G.add_edge(u, v, weight=weight)
441: (4)                return G

----------------------------------------

File 116 - redundancy.py:

1: (0)              """Node redundancy for bipartite graphs."""
2: (0)              from itertools import combinations
3: (0)              import networkx as nx
4: (0)              from networkx import NetworkXError
5: (0)              __all__ = ["node_redundancy"]
6: (0)              @nx._dispatchable
7: (0)              def node_redundancy(G, nodes=None):
8: (4)                  r"""Computes the node redundancy coefficients for the nodes in the
bipartite
9: (4)                  graph `G`.
10: (4)                 The redundancy coefficient of a node `v` is the fraction of pairs of
11: (4)                 neighbors of `v` that are both linked to other nodes. In a one-mode
12: (4)                 projection these nodes would be linked together even if `v` were
13: (4)                 not there.
14: (4)                 More formally, for any vertex `v`, the *redundancy coefficient of `v`* is
15: (4)                 defined by
16: (4)                 .. math::
17: (8)                     rc(v) = \frac{|\{\{u, w\} \subseteq N(v),
18: (8)                     \: \exists v' \neq  v,\: (v',u) \in E\:
19: (8)                     \mathrm{and}\: (v',w) \in E\}|}{ \frac{|N(v)|(|N(v)|-1)}{2}},
20: (4)                 where `N(v)` is the set of neighbors of `v` in `G`.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : graph
24: (8)                     A bipartite graph
25: (4)                 nodes : list or iterable (optional)
26: (8)                     Compute redundancy for these nodes. The default is all nodes in G.
27: (4)                 Returns
28: (4)                 -------
29: (4)                 redundancy : dictionary
30: (8)                     A dictionary keyed by node with the node redundancy value.
31: (4)                 Examples
32: (4)                 --------
33: (4)                 Compute the redundancy coefficient of each node in a graph::
34: (8)                     >>> from networkx.algorithms import bipartite
35: (8)                     >>> G = nx.cycle_graph(4)
36: (8)                     >>> rc = bipartite.node_redundancy(G)
37: (8)                     >>> rc[0]
38: (8)                     1.0
39: (4)                 Compute the average redundancy for the graph::
40: (8)                     >>> from networkx.algorithms import bipartite
41: (8)                     >>> G = nx.cycle_graph(4)
42: (8)                     >>> rc = bipartite.node_redundancy(G)
43: (8)                     >>> sum(rc.values()) / len(G)
44: (8)                     1.0
45: (4)                 Compute the average redundancy for a set of nodes::
46: (8)                     >>> from networkx.algorithms import bipartite
47: (8)                     >>> G = nx.cycle_graph(4)
48: (8)                     >>> rc = bipartite.node_redundancy(G)
49: (8)                     >>> nodes = [0, 2]
50: (8)                     >>> sum(rc[n] for n in nodes) / len(nodes)
51: (8)                     1.0
52: (4)                 Raises
53: (4)                 ------
54: (4)                 NetworkXError
55: (8)                     If any of the nodes in the graph (or in `nodes`, if specified) has
```

```
56: (8)                     (out-)degree less than two (which would result in division by zero,
57: (8)                     according to the definition of the redundancy coefficient).
58: (4)                 References
59: (4)                 ----------
60: (4)                 .. [1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio
(2008).
61: (7)                    Basic notions for the analysis of large two-mode networks.
62: (7)                    Social Networks 30(1), 31--48.
63: (4)                 """
64: (4)                 if nodes is None:
65: (8)                     nodes = G
66: (4)                 if any(len(G[v]) < 2 for v in nodes):
67: (8)                     raise NetworkXError(
68: (12)                        "Cannot compute redundancy coefficient for a node"
69: (12)                        " that has fewer than two neighbors."
70: (8)                     )
71: (4)                 return {v: _node_redundancy(G, v) for v in nodes}
72: (0)             def _node_redundancy(G, v):
73: (4)                 """Returns the redundancy of the node `v` in the bipartite graph `G`.
74: (4)                 If `G` is a graph with `n` nodes, the redundancy of a node is the ratio
75: (4)                 of the "overlap" of `v` to the maximum possible overlap of `v`
76: (4)                 according to its degree. The overlap of `v` is the number of pairs of
77: (4)                 neighbors that have mutual neighbors themselves, other than `v`.
78: (4)                 `v` must have at least two neighbors in `G`.
79: (4)                 """
80: (4)                 n = len(G[v])
81: (4)                 overlap = sum(
82: (8)                     1 for (u, w) in combinations(G[v], 2) if (set(G[u]) & set(G[w])) - {v}
83: (4)                 )
84: (4)                 return (2 * overlap) / (n * (n - 1))
```

File 117 - spectral.py:

```
1: (0)              """
2: (0)              Spectral bipartivity measure.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = ["spectral_bipartivity"]
6: (0)              @nx._dispatchable(edge_attrs="weight")
7: (0)              def spectral_bipartivity(G, nodes=None, weight="weight"):
8: (4)                  """Returns the spectral bipartivity.
9: (4)                  Parameters
10: (4)                 ----------
11: (4)                 G : NetworkX graph
12: (4)                 nodes : list or container  optional(default is all nodes)
13: (6)                   Nodes to return value of spectral bipartivity contribution.
14: (4)                 weight : string or None  optional (default = 'weight')
15: (6)                   Edge data key to use for edge weights. If None, weights set to 1.
16: (4)                 Returns
17: (4)                 -------
18: (4)                 sb : float or dict
19: (7)                    A single number if the keyword nodes is not specified, or
20: (7)                    a dictionary keyed by node with the spectral bipartivity contribution
21: (7)                    of that node as the value.
22: (4)                 Examples
23: (4)                 --------
24: (4)                 >>> from networkx.algorithms import bipartite
25: (4)                 >>> G = nx.path_graph(4)
26: (4)                 >>> bipartite.spectral_bipartivity(G)
27: (4)                 1.0
28: (4)                 Notes
29: (4)                 -----
30: (4)                 This implementation uses Numpy (dense) matrices which are not efficient
31: (4)                 for storing large sparse graphs.
32: (4)                 See Also
33: (4)                 --------
34: (4)                 color
```

```
35: (4)                 References
36: (4)                 ----------
37: (4)                 .. [1] E. Estrada and J. A. Rodríguez-Velázquez, "Spectral measures of
38: (7)                    bipartivity in complex networks", PhysRev E 72, 046105 (2005)
39: (4)                 """
40: (4)                 import scipy as sp
41: (4)                 nodelist = list(G)  # ordering of nodes in matrix
42: (4)                 A = nx.to_numpy_array(G, nodelist, weight=weight)
43: (4)                 expA = sp.linalg.expm(A)
44: (4)                 expmA = sp.linalg.expm(-A)
45: (4)                 coshA = 0.5 * (expA + expmA)
46: (4)                 if nodes is None:
47: (8)                     return float(coshA.diagonal().sum() / expA.diagonal().sum())
48: (4)                 else:
49: (8)                     index = dict(zip(nodelist, range(len(nodelist))))
50: (8)                     sb = {}
51: (8)                     for n in nodes:
52: (12)                        i = index[n]
53: (12)                        sb[n] = coshA.item(i, i) / expA.item(i, i)
54: (8)                     return sb
```

File 118 - __init__.py:

```
1: (0)
```

File 119 - test_basic.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import bipartite
4: (0)              class TestBipartiteBasic:
5: (4)                  def test_is_bipartite(self):
6: (8)                      assert bipartite.is_bipartite(nx.path_graph(4))
7: (8)                      assert bipartite.is_bipartite(nx.DiGraph([(1, 0)]))
8: (8)                      assert not bipartite.is_bipartite(nx.complete_graph(3))
9: (4)                  def test_bipartite_color(self):
10: (8)                     G = nx.path_graph(4)
11: (8)                     c = bipartite.color(G)
12: (8)                     assert c == {0: 1, 1: 0, 2: 1, 3: 0}
13: (4)                 def test_not_bipartite_color(self):
14: (8)                     with pytest.raises(nx.NetworkXError):
15: (12)                        c = bipartite.color(nx.complete_graph(4))
16: (4)                 def test_bipartite_directed(self):
17: (8)                     G = bipartite.random_graph(10, 10, 0.1, directed=True)
18: (8)                     assert bipartite.is_bipartite(G)
19: (4)                 def test_bipartite_sets(self):
20: (8)                     G = nx.path_graph(4)
21: (8)                     X, Y = bipartite.sets(G)
22: (8)                     assert X == {0, 2}
23: (8)                     assert Y == {1, 3}
24: (4)                 def test_bipartite_sets_directed(self):
25: (8)                     G = nx.path_graph(4)
26: (8)                     D = G.to_directed()
27: (8)                     X, Y = bipartite.sets(D)
28: (8)                     assert X == {0, 2}
29: (8)                     assert Y == {1, 3}
30: (4)                 def test_bipartite_sets_given_top_nodes(self):
31: (8)                     G = nx.path_graph(4)
32: (8)                     top_nodes = [0, 2]
33: (8)                     X, Y = bipartite.sets(G, top_nodes)
34: (8)                     assert X == {0, 2}
35: (8)                     assert Y == {1, 3}
36: (4)                 def test_bipartite_sets_disconnected(self):
37: (8)                     with pytest.raises(nx.AmbiguousSolution):
38: (12)                        G = nx.path_graph(4)
```

```
39: (12)                        G.add_edges_from([(5, 6), (6, 7)])
40: (12)                        X, Y = bipartite.sets(G)
41: (4)                 def test_is_bipartite_node_set(self):
42: (8)                     G = nx.path_graph(4)
43: (8)                     with pytest.raises(nx.AmbiguousSolution):
44: (12)                        bipartite.is_bipartite_node_set(G, [1, 1, 2, 3])
45: (8)                     assert bipartite.is_bipartite_node_set(G, [0, 2])
46: (8)                     assert bipartite.is_bipartite_node_set(G, [1, 3])
47: (8)                     assert not bipartite.is_bipartite_node_set(G, [1, 2])
48: (8)                     G.add_edge(10, 20)
49: (8)                     assert bipartite.is_bipartite_node_set(G, [0, 2, 10])
50: (8)                     assert bipartite.is_bipartite_node_set(G, [0, 2, 20])
51: (8)                     assert bipartite.is_bipartite_node_set(G, [1, 3, 10])
52: (8)                     assert bipartite.is_bipartite_node_set(G, [1, 3, 20])
53: (4)                 def test_bipartite_density(self):
54: (8)                     G = nx.path_graph(5)
55: (8)                     X, Y = bipartite.sets(G)
56: (8)                     density = len(list(G.edges())) / (len(X) * len(Y))
57: (8)                     assert bipartite.density(G, X) == density
58: (8)                     D = nx.DiGraph(G.edges())
59: (8)                     assert bipartite.density(D, X) == density / 2.0
60: (8)                     assert bipartite.density(nx.Graph(), {}) == 0.0
61: (4)                 def test_bipartite_degrees(self):
62: (8)                     G = nx.path_graph(5)
63: (8)                     X = {1, 3}
64: (8)                     Y = {0, 2, 4}
65: (8)                     u, d = bipartite.degrees(G, Y)
66: (8)                     assert dict(u) == {1: 2, 3: 2}
67: (8)                     assert dict(d) == {0: 1, 2: 2, 4: 1}
68: (4)                 def test_bipartite_weighted_degrees(self):
69: (8)                     G = nx.path_graph(5)
70: (8)                     G.add_edge(0, 1, weight=0.1, other=0.2)
71: (8)                     X = {1, 3}
72: (8)                     Y = {0, 2, 4}
73: (8)                     u, d = bipartite.degrees(G, Y, weight="weight")
74: (8)                     assert dict(u) == {1: 1.1, 3: 2}
75: (8)                     assert dict(d) == {0: 0.1, 2: 2, 4: 1}
76: (8)                     u, d = bipartite.degrees(G, Y, weight="other")
77: (8)                     assert dict(u) == {1: 1.2, 3: 2}
78: (8)                     assert dict(d) == {0: 0.2, 2: 2, 4: 1}
79: (4)                 def test_biadjacency_matrix_weight(self):
80: (8)                     pytest.importorskip("scipy")
81: (8)                     G = nx.path_graph(5)
82: (8)                     G.add_edge(0, 1, weight=2, other=4)
83: (8)                     X = [1, 3]
84: (8)                     Y = [0, 2, 4]
85: (8)                     M = bipartite.biadjacency_matrix(G, X, weight="weight")
86: (8)                     assert M[0, 0] == 2
87: (8)                     M = bipartite.biadjacency_matrix(G, X, weight="other")
88: (8)                     assert M[0, 0] == 4
89: (4)                 def test_biadjacency_matrix(self):
90: (8)                     pytest.importorskip("scipy")
91: (8)                     tops = [2, 5, 10]
92: (8)                     bots = [5, 10, 15]
93: (8)                     for i in range(len(tops)):
94: (12)                        G = bipartite.random_graph(tops[i], bots[i], 0.2)
95: (12)                        top = [n for n, d in G.nodes(data=True) if d["bipartite"] == 0]
96: (12)                        M = bipartite.biadjacency_matrix(G, top)
97: (12)                        assert M.shape[0] == tops[i]
98: (12)                        assert M.shape[1] == bots[i]
99: (4)                 def test_biadjacency_matrix_order(self):
100: (8)                    pytest.importorskip("scipy")
101: (8)                    G = nx.path_graph(5)
102: (8)                    G.add_edge(0, 1, weight=2)
103: (8)                    X = [3, 1]
104: (8)                    Y = [4, 2, 0]
105: (8)                    M = bipartite.biadjacency_matrix(G, X, Y, weight="weight")
106: (8)                    assert M[1, 2] == 2
```

File 120 - test_centrality.py:

```
import pytest
import networkx as nx
from networkx.algorithms import bipartite
class TestBipartiteCentrality:
    @classmethod
    def setup_class(cls):
        cls.P4 = nx.path_graph(4)
        cls.K3 = nx.complete_bipartite_graph(3, 3)
        cls.C4 = nx.cycle_graph(4)
        cls.davis = nx.davis_southern_women_graph()
        cls.top_nodes = [
            n for n, d in cls.davis.nodes(data=True) if d["bipartite"] == 0
        ]
    def test_degree_centrality(self):
        d = bipartite.degree_centrality(self.P4, [1, 3])
        answer = {0: 0.5, 1: 1.0, 2: 1.0, 3: 0.5}
        assert d == answer
        d = bipartite.degree_centrality(self.K3, [0, 1, 2])
        answer = {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0, 5: 1.0}
        assert d == answer
        d = bipartite.degree_centrality(self.C4, [0, 2])
        answer = {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0}
        assert d == answer
    def test_betweenness_centrality(self):
        c = bipartite.betweenness_centrality(self.P4, [1, 3])
        answer = {0: 0.0, 1: 1.0, 2: 1.0, 3: 0.0}
        assert c == answer
        c = bipartite.betweenness_centrality(self.K3, [0, 1, 2])
        answer = {0: 0.125, 1: 0.125, 2: 0.125, 3: 0.125, 4: 0.125, 5: 0.125}
        assert c == answer
        c = bipartite.betweenness_centrality(self.C4, [0, 2])
        answer = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
        assert c == answer
    def test_closeness_centrality(self):
        c = bipartite.closeness_centrality(self.P4, [1, 3])
        answer = {0: 2.0 / 3, 1: 1.0, 2: 1.0, 3: 2.0 / 3}
        assert c == answer
        c = bipartite.closeness_centrality(self.K3, [0, 1, 2])
        answer = {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0, 5: 1.0}
        assert c == answer
        c = bipartite.closeness_centrality(self.C4, [0, 2])
        answer = {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0}
        assert c == answer
        G = nx.Graph()
        G.add_node(0)
        G.add_node(1)
        c = bipartite.closeness_centrality(G, [0])
        assert c == {0: 0.0, 1: 0.0}
        c = bipartite.closeness_centrality(G, [1])
        assert c == {0: 0.0, 1: 0.0}
    def test_bipartite_closeness_centrality_unconnected(self):
        G = nx.complete_bipartite_graph(3, 3)
        G.add_edge(6, 7)
        c = bipartite.closeness_centrality(G, [0, 2, 4, 6], normalized=False)
        answer = {
            0: 10.0 / 7,
            2: 10.0 / 7,
            4: 10.0 / 7,
            6: 10.0,
            1: 10.0 / 7,
            3: 10.0 / 7,
            5: 10.0 / 7,
            7: 10.0,
        }
        assert c == answer
```

```
(4)     def test_davis_degree_centrality(self):
(8)         G = self.davis
(8)         deg = bipartite.degree_centrality(G, self.top_nodes)
(8)         answer = {
(12)            "E8": 0.78,
(12)            "E9": 0.67,
(12)            "E7": 0.56,
(12)            "Nora Fayette": 0.57,
(12)            "Evelyn Jefferson": 0.57,
(12)            "Theresa Anderson": 0.57,
(12)            "E6": 0.44,
(12)            "Sylvia Avondale": 0.50,
(12)            "Laura Mandeville": 0.50,
(12)            "Brenda Rogers": 0.50,
(12)            "Katherina Rogers": 0.43,
(12)            "E5": 0.44,
(12)            "Helen Lloyd": 0.36,
(12)            "E3": 0.33,
(12)            "Ruth DeSand": 0.29,
(12)            "Verne Sanderson": 0.29,
(12)            "E12": 0.33,
(12)            "Myra Liddel": 0.29,
(12)            "E11": 0.22,
(12)            "Eleanor Nye": 0.29,
(12)            "Frances Anderson": 0.29,
(12)            "Pearl Oglethorpe": 0.21,
(12)            "E4": 0.22,
(12)            "Charlotte McDowd": 0.29,
(12)            "E10": 0.28,
(12)            "Olivia Carleton": 0.14,
(12)            "Flora Price": 0.14,
(12)            "E2": 0.17,
(12)            "E1": 0.17,
(12)            "Dorothy Murchison": 0.14,
(12)            "E13": 0.17,
(12)            "E14": 0.17,
(8)         }
(8)         for node, value in answer.items():
(12)            assert value == pytest.approx(deg[node], abs=1e-2)
(4)     def test_davis_betweenness_centrality(self):
(8)         G = self.davis
(8)         bet = bipartite.betweenness_centrality(G, self.top_nodes)
(8)         answer = {
(12)            "E8": 0.24,
(12)            "E9": 0.23,
(12)            "E7": 0.13,
(12)            "Nora Fayette": 0.11,
(12)            "Evelyn Jefferson": 0.10,
(12)            "Theresa Anderson": 0.09,
(12)            "E6": 0.07,
(12)            "Sylvia Avondale": 0.07,
(12)            "Laura Mandeville": 0.05,
(12)            "Brenda Rogers": 0.05,
(12)            "Katherina Rogers": 0.05,
(12)            "E5": 0.04,
(12)            "Helen Lloyd": 0.04,
(12)            "E3": 0.02,
(12)            "Ruth DeSand": 0.02,
(12)            "Verne Sanderson": 0.02,
(12)            "E12": 0.02,
(12)            "Myra Liddel": 0.02,
(12)            "E11": 0.02,
(12)            "Eleanor Nye": 0.01,
(12)            "Frances Anderson": 0.01,
(12)            "Pearl Oglethorpe": 0.01,
(12)            "E4": 0.01,
(12)            "Charlotte McDowd": 0.01,
(12)            "E10": 0.01,
(12)            "Olivia Carleton": 0.01,
```

```
135: (12)                        "Flora Price": 0.01,
136: (12)                        "E2": 0.00,
137: (12)                        "E1": 0.00,
138: (12)                        "Dorothy Murchison": 0.00,
139: (12)                        "E13": 0.00,
140: (12)                        "E14": 0.00,
141: (8)                     }
142: (8)                     for node, value in answer.items():
143: (12)                        assert value == pytest.approx(bet[node], abs=1e-2)
144: (4)                 def test_davis_closeness_centrality(self):
145: (8)                     G = self.davis
146: (8)                     clos = bipartite.closeness_centrality(G, self.top_nodes)
147: (8)                     answer = {
148: (12)                        "E8": 0.85,
149: (12)                        "E9": 0.79,
150: (12)                        "E7": 0.73,
151: (12)                        "Nora Fayette": 0.80,
152: (12)                        "Evelyn Jefferson": 0.80,
153: (12)                        "Theresa Anderson": 0.80,
154: (12)                        "E6": 0.69,
155: (12)                        "Sylvia Avondale": 0.77,
156: (12)                        "Laura Mandeville": 0.73,
157: (12)                        "Brenda Rogers": 0.73,
158: (12)                        "Katherina Rogers": 0.73,
159: (12)                        "E5": 0.59,
160: (12)                        "Helen Lloyd": 0.73,
161: (12)                        "E3": 0.56,
162: (12)                        "Ruth DeSand": 0.71,
163: (12)                        "Verne Sanderson": 0.71,
164: (12)                        "E12": 0.56,
165: (12)                        "Myra Liddel": 0.69,
166: (12)                        "E11": 0.54,
167: (12)                        "Eleanor Nye": 0.67,
168: (12)                        "Frances Anderson": 0.67,
169: (12)                        "Pearl Oglethorpe": 0.67,
170: (12)                        "E4": 0.54,
171: (12)                        "Charlotte McDowd": 0.60,
172: (12)                        "E10": 0.55,
173: (12)                        "Olivia Carleton": 0.59,
174: (12)                        "Flora Price": 0.59,
175: (12)                        "E2": 0.52,
176: (12)                        "E1": 0.52,
177: (12)                        "Dorothy Murchison": 0.65,
178: (12)                        "E13": 0.52,
179: (12)                        "E14": 0.52,
180: (8)                     }
181: (8)                     for node, value in answer.items():
182: (12)                        assert value == pytest.approx(clos[node], abs=1e-2)

----------------------------------------

File 121 - test_cluster.py:

1: (0)               import pytest
2: (0)               import networkx as nx
3: (0)               from networkx.algorithms import bipartite
4: (0)               from networkx.algorithms.bipartite.cluster import cc_dot, cc_max, cc_min
5: (0)               def test_pairwise_bipartite_cc_functions():
6: (4)                   G1 = nx.Graph([(0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (1, 5), (1, 6), (1,
7)])
7: (4)                   G2 = nx.Graph([(0, 2), (0, 3), (0, 4), (1, 3), (1, 4), (1, 5)])
8: (4)                   G3 = nx.Graph(
9: (8)                       [(0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (1, 5), (1, 6), (1, 7), (1,
8), (1, 9)]
10: (4)                  )
11: (4)                  result = {
12: (8)                      0: [1 / 3.0, 2 / 3.0, 2 / 5.0],
13: (8)                      1: [1 / 2.0, 2 / 3.0, 2 / 3.0],
14: (8)                      2: [2 / 8.0, 2 / 5.0, 2 / 5.0],
```

```
15: (4)                 }
16: (4)                 for i, G in enumerate([G1, G2, G3]):
17: (8)                     assert bipartite.is_bipartite(G)
18: (8)                     assert cc_dot(set(G[0]), set(G[1])) == result[i][0]
19: (8)                     assert cc_min(set(G[0]), set(G[1])) == result[i][1]
20: (8)                     assert cc_max(set(G[0]), set(G[1])) == result[i][2]
21: (0)             def test_star_graph():
22: (4)                 G = nx.star_graph(3)
23: (4)                 answer = {0: 0, 1: 1, 2: 1, 3: 1}
24: (4)                 assert bipartite.clustering(G, mode="dot") == answer
25: (4)                 assert bipartite.clustering(G, mode="min") == answer
26: (4)                 assert bipartite.clustering(G, mode="max") == answer
27: (0)             def test_not_bipartite():
28: (4)                 with pytest.raises(nx.NetworkXError):
29: (8)                     bipartite.clustering(nx.complete_graph(4))
30: (0)             def test_bad_mode():
31: (4)                 with pytest.raises(nx.NetworkXError):
32: (8)                     bipartite.clustering(nx.path_graph(4), mode="foo")
33: (0)             def test_path_graph():
34: (4)                 G = nx.path_graph(4)
35: (4)                 answer = {0: 0.5, 1: 0.5, 2: 0.5, 3: 0.5}
36: (4)                 assert bipartite.clustering(G, mode="dot") == answer
37: (4)                 assert bipartite.clustering(G, mode="max") == answer
38: (4)                 answer = {0: 1, 1: 1, 2: 1, 3: 1}
39: (4)                 assert bipartite.clustering(G, mode="min") == answer
40: (0)             def test_average_path_graph():
41: (4)                 G = nx.path_graph(4)
42: (4)                 assert bipartite.average_clustering(G, mode="dot") == 0.5
43: (4)                 assert bipartite.average_clustering(G, mode="max") == 0.5
44: (4)                 assert bipartite.average_clustering(G, mode="min") == 1
45: (0)             def test_ra_clustering_davis():
46: (4)                 G = nx.davis_southern_women_graph()
47: (4)                 cc4 = round(bipartite.robins_alexander_clustering(G), 3)
48: (4)                 assert cc4 == 0.468
49: (0)             def test_ra_clustering_square():
50: (4)                 G = nx.path_graph(4)
51: (4)                 G.add_edge(0, 3)
52: (4)                 assert bipartite.robins_alexander_clustering(G) == 1.0
53: (0)             def test_ra_clustering_zero():
54: (4)                 G = nx.Graph()
55: (4)                 assert bipartite.robins_alexander_clustering(G) == 0
56: (4)                 G.add_nodes_from(range(4))
57: (4)                 assert bipartite.robins_alexander_clustering(G) == 0
58: (4)                 G.add_edges_from([(0, 1), (2, 3), (3, 4)])
59: (4)                 assert bipartite.robins_alexander_clustering(G) == 0
60: (4)                 G.add_edge(1, 2)
61: (4)                 assert bipartite.robins_alexander_clustering(G) == 0

----------------------------------------

File 122 - test_covering.py:

1: (0)              import networkx as nx
2: (0)              from networkx.algorithms import bipartite
3: (0)              class TestMinEdgeCover:
4: (4)                  """Tests for :func:`networkx.algorithms.bipartite.min_edge_cover`"""
5: (4)                  def test_empty_graph(self):
6: (8)                      G = nx.Graph()
7: (8)                      assert bipartite.min_edge_cover(G) == set()
8: (4)                  def test_graph_single_edge(self):
9: (8)                      G = nx.Graph()
10: (8)                     G.add_edge(0, 1)
11: (8)                     assert bipartite.min_edge_cover(G) == {(0, 1), (1, 0)}
12: (4)                 def test_bipartite_default(self):
13: (8)                     G = nx.Graph()
14: (8)                     G.add_nodes_from([1, 2, 3, 4], bipartite=0)
15: (8)                     G.add_nodes_from(["a", "b", "c"], bipartite=1)
16: (8)                     G.add_edges_from([(1, "a"), (1, "b"), (2, "b"), (2, "c"), (3, "c"),
(4, "a")])
```

```
17: (8)                     min_cover = bipartite.min_edge_cover(G)
18: (8)                     assert nx.is_edge_cover(G, min_cover)
19: (8)                     assert len(min_cover) == 8
20: (4)                 def test_bipartite_explicit(self):
21: (8)                     G = nx.Graph()
22: (8)                     G.add_nodes_from([1, 2, 3, 4], bipartite=0)
23: (8)                     G.add_nodes_from(["a", "b", "c"], bipartite=1)
24: (8)                     G.add_edges_from([(1, "a"), (1, "b"), (2, "b"), (2, "c"), (3, "c"),
(4, "a")])
25: (8)                     min_cover = bipartite.min_edge_cover(G, bipartite.eppstein_matching)
26: (8)                     assert nx.is_edge_cover(G, min_cover)
27: (8)                     assert len(min_cover) == 8
```

File 123 - test_edgelist.py:

```
1: (0)              """
2: (4)                  Unit tests for bipartite edgelists.
3: (0)              """
4: (0)              import io
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              from networkx.algorithms import bipartite
8: (0)              from networkx.utils import edges_equal, graphs_equal, nodes_equal
9: (0)              class TestEdgelist:
10: (4)                 @classmethod
11: (4)                 def setup_class(cls):
12: (8)                     cls.G = nx.Graph(name="test")
13: (8)                     e = [("a", "b"), ("b", "c"), ("c", "d"), ("d", "e"), ("e", "f"), ("a",
"f")]
14: (8)                     cls.G.add_edges_from(e)
15: (8)                     cls.G.add_nodes_from(["a", "c", "e"], bipartite=0)
16: (8)                     cls.G.add_nodes_from(["b", "d", "f"], bipartite=1)
17: (8)                     cls.G.add_node("g", bipartite=0)
18: (8)                     cls.DG = nx.DiGraph(cls.G)
19: (8)                     cls.MG = nx.MultiGraph()
20: (8)                     cls.MG.add_edges_from([(1, 2), (1, 2), (1, 2)])
21: (8)                     cls.MG.add_node(1, bipartite=0)
22: (8)                     cls.MG.add_node(2, bipartite=1)
23: (4)                 def test_read_edgelist_1(self):
24: (8)                     s = b"""\
25: (0)             1 2
26: (0)             2 3
27: (0)             """
28: (8)                     bytesIO = io.BytesIO(s)
29: (8)                     G = bipartite.read_edgelist(bytesIO, nodetype=int)
30: (8)                     assert edges_equal(G.edges(), [(1, 2), (2, 3)])
31: (4)                 def test_read_edgelist_3(self):
32: (8)                     s = b"""\
33: (0)             1 2 {'weight':2.0}
34: (0)             2 3 {'weight':3.0}
35: (0)             """
36: (8)                     bytesIO = io.BytesIO(s)
37: (8)                     G = bipartite.read_edgelist(bytesIO, nodetype=int, data=False)
38: (8)                     assert edges_equal(G.edges(), [(1, 2), (2, 3)])
39: (8)                     bytesIO = io.BytesIO(s)
40: (8)                     G = bipartite.read_edgelist(bytesIO, nodetype=int, data=True)
41: (8)                     assert edges_equal(
42: (12)                        G.edges(data=True), [(1, 2, {"weight": 2.0}), (2, 3, {"weight":
3.0})]
43: (8)                     )
44: (4)                 def test_write_edgelist_1(self):
45: (8)                     fh = io.BytesIO()
46: (8)                     G = nx.Graph()
47: (8)                     G.add_edges_from([(1, 2), (2, 3)])
48: (8)                     G.add_node(1, bipartite=0)
49: (8)                     G.add_node(2, bipartite=1)
50: (8)                     G.add_node(3, bipartite=0)
```

```
 51: (8)                     bipartite.write_edgelist(G, fh, data=False)
 52: (8)                     fh.seek(0)
 53: (8)                     assert fh.read() == b"1 2\n3 2\n"
 54: (4)                 def test_write_edgelist_2(self):
 55: (8)                     fh = io.BytesIO()
 56: (8)                     G = nx.Graph()
 57: (8)                     G.add_edges_from([(1, 2), (2, 3)])
 58: (8)                     G.add_node(1, bipartite=0)
 59: (8)                     G.add_node(2, bipartite=1)
 60: (8)                     G.add_node(3, bipartite=0)
 61: (8)                     bipartite.write_edgelist(G, fh, data=True)
 62: (8)                     fh.seek(0)
 63: (8)                     assert fh.read() == b"1 2 {}\n3 2 {}\n"
 64: (4)                 def test_write_edgelist_3(self):
 65: (8)                     fh = io.BytesIO()
 66: (8)                     G = nx.Graph()
 67: (8)                     G.add_edge(1, 2, weight=2.0)
 68: (8)                     G.add_edge(2, 3, weight=3.0)
 69: (8)                     G.add_node(1, bipartite=0)
 70: (8)                     G.add_node(2, bipartite=1)
 71: (8)                     G.add_node(3, bipartite=0)
 72: (8)                     bipartite.write_edgelist(G, fh, data=True)
 73: (8)                     fh.seek(0)
 74: (8)                     assert fh.read() == b"1 2 {'weight': 2.0}\n3 2 {'weight': 3.0}\n"
 75: (4)                 def test_write_edgelist_4(self):
 76: (8)                     fh = io.BytesIO()
 77: (8)                     G = nx.Graph()
 78: (8)                     G.add_edge(1, 2, weight=2.0)
 79: (8)                     G.add_edge(2, 3, weight=3.0)
 80: (8)                     G.add_node(1, bipartite=0)
 81: (8)                     G.add_node(2, bipartite=1)
 82: (8)                     G.add_node(3, bipartite=0)
 83: (8)                     bipartite.write_edgelist(G, fh, data=[("weight")])
 84: (8)                     fh.seek(0)
 85: (8)                     assert fh.read() == b"1 2 2.0\n3 2 3.0\n"
 86: (4)                 def test_unicode(self, tmp_path):
 87: (8)                     G = nx.Graph()
 88: (8)                     name1 = chr(2344) + chr(123) + chr(6543)
 89: (8)                     name2 = chr(5543) + chr(1543) + chr(324)
 90: (8)                     G.add_edge(name1, "Radiohead", **{name2: 3})
 91: (8)                     G.add_node(name1, bipartite=0)
 92: (8)                     G.add_node("Radiohead", bipartite=1)
 93: (8)                     fname = tmp_path / "edgelist.txt"
 94: (8)                     bipartite.write_edgelist(G, fname)
 95: (8)                     H = bipartite.read_edgelist(fname)
 96: (8)                     assert graphs_equal(G, H)
 97: (4)                 def test_latin1_issue(self, tmp_path):
 98: (8)                     G = nx.Graph()
 99: (8)                     name1 = chr(2344) + chr(123) + chr(6543)
100: (8)                     name2 = chr(5543) + chr(1543) + chr(324)
101: (8)                     G.add_edge(name1, "Radiohead", **{name2: 3})
102: (8)                     G.add_node(name1, bipartite=0)
103: (8)                     G.add_node("Radiohead", bipartite=1)
104: (8)                     fname = tmp_path / "edgelist.txt"
105: (8)                     with pytest.raises(UnicodeEncodeError):
106: (12)                        bipartite.write_edgelist(G, fname, encoding="latin-1")
107: (4)                 def test_latin1(self, tmp_path):
108: (8)                     G = nx.Graph()
109: (8)                     name1 = "Bj" + chr(246) + "rk"
110: (8)                     name2 = chr(220) + "ber"
111: (8)                     G.add_edge(name1, "Radiohead", **{name2: 3})
112: (8)                     G.add_node(name1, bipartite=0)
113: (8)                     G.add_node("Radiohead", bipartite=1)
114: (8)                     fname = tmp_path / "edgelist.txt"
115: (8)                     bipartite.write_edgelist(G, fname, encoding="latin-1")
116: (8)                     H = bipartite.read_edgelist(fname, encoding="latin-1")
117: (8)                     assert graphs_equal(G, H)
118: (4)                 def test_edgelist_graph(self, tmp_path):
119: (8)                     G = self.G
```

```
120: (8)                    fname = tmp_path / "edgelist.txt"
121: (8)                    bipartite.write_edgelist(G, fname)
122: (8)                    H = bipartite.read_edgelist(fname)
123: (8)                    H2 = bipartite.read_edgelist(fname)
124: (8)                    assert H is not H2  # they should be different graphs
125: (8)                    G.remove_node("g")  # isolated nodes are not written in edgelist
126: (8)                    assert nodes_equal(list(H), list(G))
127: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
128: (4)                def test_edgelist_integers(self, tmp_path):
129: (8)                    G = nx.convert_node_labels_to_integers(self.G)
130: (8)                    fname = tmp_path / "edgelist.txt"
131: (8)                    bipartite.write_edgelist(G, fname)
132: (8)                    H = bipartite.read_edgelist(fname, nodetype=int)
133: (8)                    G.remove_nodes_from(list(nx.isolates(G)))
134: (8)                    assert nodes_equal(list(H), list(G))
135: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
136: (4)                def test_edgelist_multigraph(self, tmp_path):
137: (8)                    G = self.MG
138: (8)                    fname = tmp_path / "edgelist.txt"
139: (8)                    bipartite.write_edgelist(G, fname)
140: (8)                    H = bipartite.read_edgelist(fname, nodetype=int,
create_using=nx.MultiGraph())
141: (8)                    H2 = bipartite.read_edgelist(fname, nodetype=int,
create_using=nx.MultiGraph())
142: (8)                    assert H is not H2  # they should be different graphs
143: (8)                    assert nodes_equal(list(H), list(G))
144: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
145: (4)                def test_empty_digraph(self):
146: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
147: (12)                       bytesIO = io.BytesIO()
148: (12)                       bipartite.write_edgelist(nx.DiGraph(), bytesIO)
149: (4)                def test_raise_attribute(self):
150: (8)                    with pytest.raises(AttributeError):
151: (12)                       G = nx.path_graph(4)
152: (12)                       bytesIO = io.BytesIO()
153: (12)                       bipartite.write_edgelist(G, bytesIO)
154: (4)                def test_parse_edgelist(self):
155: (8)                    """Tests for conditions specific to
156: (8)                    parse_edge_list method"""
157: (8)                    lines = ["1 2", "2 3", "3 1", "4", " "]
158: (8)                    G = bipartite.parse_edgelist(lines, nodetype=int)
159: (8)                    assert list(G.nodes) == [1, 2, 3]
160: (8)                    with pytest.raises(TypeError, match=".*Failed to convert nodes"):
161: (12)                       lines = ["a b", "b c", "c a"]
162: (12)                       G = bipartite.parse_edgelist(lines, nodetype=int)
163: (8)                    with pytest.raises(TypeError, match=".*Failed to convert edge data"):
164: (12)                       lines = ["1 2 3", "2 3 4", "3 1 2"]
165: (12)                       G = bipartite.parse_edgelist(lines, nodetype=int)
166: (8)                    with pytest.raises(IndexError):
167: (12)                       lines = ["1 2 3 4", "2 3 4"]
168: (12)                       G = bipartite.parse_edgelist(
169: (16)                           lines, nodetype=int, data=[("weight", int), ("key", int)]
170: (12)                       )
171: (8)                    with pytest.raises(TypeError, match=".*Failed to convert key data"):
172: (12)                       lines = ["1 2 3 a", "2 3 4 b"]
173: (12)                       G = bipartite.parse_edgelist(
174: (16)                           lines, nodetype=int, data=[("weight", int), ("key", int)]
175: (12)                       )
```

File 124 - test_extendability.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_selfloops_raises():
4: (4)                  G = nx.ladder_graph(3)
5: (4)                  G.add_edge(0, 0)
6: (4)                  with pytest.raises(nx.NetworkXError, match=".*not bipartite"):
```

```
 7: (8)                     nx.bipartite.maximal_extendability(G)
 8: (0)             def test_disconnected_raises():
 9: (4)                 G = nx.ladder_graph(3)
 10: (4)                G.add_node("a")
 11: (4)                with pytest.raises(nx.NetworkXError, match=".*not connected"):
 12: (8)                    nx.bipartite.maximal_extendability(G)
 13: (0)            def test_not_bipartite_raises():
 14: (4)                G = nx.complete_graph(5)
 15: (4)                with pytest.raises(nx.NetworkXError, match=".*not bipartite"):
 16: (8)                    nx.bipartite.maximal_extendability(G)
 17: (0)            def test_no_perfect_matching_raises():
 18: (4)                G = nx.Graph([(0, 1), (0, 2)])
 19: (4)                with pytest.raises(nx.NetworkXError, match=".*not contain a perfect
 matching"):
 20: (8)                    nx.bipartite.maximal_extendability(G)
 21: (0)            def test_residual_graph_not_strongly_connected_raises():
 22: (4)                G = nx.Graph([(1, 2), (2, 3), (3, 4)])
 23: (4)                with pytest.raises(
 24: (8)                    nx.NetworkXError, match="The residual graph of G is not strongly
 connected"
 25: (4)                ):
 26: (8)                    nx.bipartite.maximal_extendability(G)
 27: (0)            def test_ladder_graph_is_1():
 28: (4)                G = nx.ladder_graph(3)
 29: (4)                assert nx.bipartite.maximal_extendability(G) == 1
 30: (0)            def test_cubical_graph_is_2():
 31: (4)                G = nx.cubical_graph()
 32: (4)                assert nx.bipartite.maximal_extendability(G) == 2
 33: (0)            def test_k_is_3():
 34: (4)                G = nx.Graph(
 35: (8)                    [
 36: (12)                       (1, 6),
 37: (12)                       (1, 7),
 38: (12)                       (1, 8),
 39: (12)                       (1, 9),
 40: (12)                       (2, 6),
 41: (12)                       (2, 7),
 42: (12)                       (2, 8),
 43: (12)                       (2, 10),
 44: (12)                       (3, 6),
 45: (12)                       (3, 8),
 46: (12)                       (3, 9),
 47: (12)                       (3, 10),
 48: (12)                       (4, 7),
 49: (12)                       (4, 8),
 50: (12)                       (4, 9),
 51: (12)                       (4, 10),
 52: (12)                       (5, 6),
 53: (12)                       (5, 7),
 54: (12)                       (5, 9),
 55: (12)                       (5, 10),
 56: (8)                    ]
 57: (4)                )
 58: (4)                assert nx.bipartite.maximal_extendability(G) == 3
 59: (0)            def test_k_is_4():
 60: (4)                G = nx.Graph(
 61: (8)                    [
 62: (12)                       (8, 1),
 63: (12)                       (8, 2),
 64: (12)                       (8, 3),
 65: (12)                       (8, 4),
 66: (12)                       (8, 5),
 67: (12)                       (9, 1),
 68: (12)                       (9, 2),
 69: (12)                       (9, 3),
 70: (12)                       (9, 4),
 71: (12)                       (9, 7),
 72: (12)                       (10, 1),
 73: (12)                       (10, 2),
```

```
(12)                        (10, 3),
(12)                        (10, 4),
(12)                        (10, 6),
(12)                        (11, 1),
(12)                        (11, 2),
(12)                        (11, 5),
(12)                        (11, 6),
(12)                        (11, 7),
(12)                        (12, 1),
(12)                        (12, 3),
(12)                        (12, 5),
(12)                        (12, 6),
(12)                        (12, 7),
(12)                        (13, 2),
(12)                        (13, 4),
(12)                        (13, 5),
(12)                        (13, 6),
(12)                        (13, 7),
(12)                        (14, 3),
(12)                        (14, 4),
(12)                        (14, 5),
(12)                        (14, 6),
(12)                        (14, 7),
(8)                     ]
(4)                 )
(4)                 assert nx.bipartite.maximal_extendability(G) == 4
(0)             def test_k_is_5():
(4)                 G = nx.Graph(
(8)                     [
(12)                        (8, 1),
(12)                        (8, 2),
(12)                        (8, 3),
(12)                        (8, 4),
(12)                        (8, 5),
(12)                        (8, 6),
(12)                        (9, 1),
(12)                        (9, 2),
(12)                        (9, 3),
(12)                        (9, 4),
(12)                        (9, 5),
(12)                        (9, 7),
(12)                        (10, 1),
(12)                        (10, 2),
(12)                        (10, 3),
(12)                        (10, 4),
(12)                        (10, 6),
(12)                        (10, 7),
(12)                        (11, 1),
(12)                        (11, 2),
(12)                        (11, 3),
(12)                        (11, 5),
(12)                        (11, 6),
(12)                        (11, 7),
(12)                        (12, 1),
(12)                        (12, 2),
(12)                        (12, 4),
(12)                        (12, 5),
(12)                        (12, 6),
(12)                        (12, 7),
(12)                        (13, 1),
(12)                        (13, 3),
(12)                        (13, 4),
(12)                        (13, 5),
(12)                        (13, 6),
(12)                        (13, 7),
(12)                        (14, 2),
(12)                        (14, 3),
(12)                        (14, 4),
(12)                        (14, 5),
```

```
(12)                        (14, 6),
(12)                        (14, 7),
(8)                     ]
(4)                 )
(4)                 assert nx.bipartite.maximal_extendability(G) == 5
(0)             def test_k_is_6():
(4)                 G = nx.Graph(
(8)                     [
(12)                        (9, 1),
(12)                        (9, 2),
(12)                        (9, 3),
(12)                        (9, 4),
(12)                        (9, 5),
(12)                        (9, 6),
(12)                        (9, 7),
(12)                        (10, 1),
(12)                        (10, 2),
(12)                        (10, 3),
(12)                        (10, 4),
(12)                        (10, 5),
(12)                        (10, 6),
(12)                        (10, 8),
(12)                        (11, 1),
(12)                        (11, 2),
(12)                        (11, 3),
(12)                        (11, 4),
(12)                        (11, 5),
(12)                        (11, 7),
(12)                        (11, 8),
(12)                        (12, 1),
(12)                        (12, 2),
(12)                        (12, 3),
(12)                        (12, 4),
(12)                        (12, 6),
(12)                        (12, 7),
(12)                        (12, 8),
(12)                        (13, 1),
(12)                        (13, 2),
(12)                        (13, 3),
(12)                        (13, 5),
(12)                        (13, 6),
(12)                        (13, 7),
(12)                        (13, 8),
(12)                        (14, 1),
(12)                        (14, 2),
(12)                        (14, 4),
(12)                        (14, 5),
(12)                        (14, 6),
(12)                        (14, 7),
(12)                        (14, 8),
(12)                        (15, 1),
(12)                        (15, 3),
(12)                        (15, 4),
(12)                        (15, 5),
(12)                        (15, 6),
(12)                        (15, 7),
(12)                        (15, 8),
(12)                        (16, 2),
(12)                        (16, 3),
(12)                        (16, 4),
(12)                        (16, 5),
(12)                        (16, 6),
(12)                        (16, 7),
(12)                        (16, 8),
(8)                     ]
(4)                 )
(4)                 assert nx.bipartite.maximal_extendability(G) == 6
(0)             def test_k_is_7():
(4)                 G = nx.Graph(
```

```
(8)                     [
(12)                        (1, 11),
(12)                        (1, 12),
(12)                        (1, 13),
(12)                        (1, 14),
(12)                        (1, 15),
(12)                        (1, 16),
(12)                        (1, 17),
(12)                        (1, 18),
(12)                        (2, 11),
(12)                        (2, 12),
(12)                        (2, 13),
(12)                        (2, 14),
(12)                        (2, 15),
(12)                        (2, 16),
(12)                        (2, 17),
(12)                        (2, 19),
(12)                        (3, 11),
(12)                        (3, 12),
(12)                        (3, 13),
(12)                        (3, 14),
(12)                        (3, 15),
(12)                        (3, 16),
(12)                        (3, 17),
(12)                        (3, 20),
(12)                        (4, 11),
(12)                        (4, 12),
(12)                        (4, 13),
(12)                        (4, 14),
(12)                        (4, 15),
(12)                        (4, 16),
(12)                        (4, 17),
(12)                        (4, 18),
(12)                        (4, 19),
(12)                        (4, 20),
(12)                        (5, 11),
(12)                        (5, 12),
(12)                        (5, 13),
(12)                        (5, 14),
(12)                        (5, 15),
(12)                        (5, 16),
(12)                        (5, 17),
(12)                        (5, 18),
(12)                        (5, 19),
(12)                        (5, 20),
(12)                        (6, 11),
(12)                        (6, 12),
(12)                        (6, 13),
(12)                        (6, 14),
(12)                        (6, 15),
(12)                        (6, 16),
(12)                        (6, 17),
(12)                        (6, 18),
(12)                        (6, 19),
(12)                        (6, 20),
(12)                        (7, 11),
(12)                        (7, 12),
(12)                        (7, 13),
(12)                        (7, 14),
(12)                        (7, 15),
(12)                        (7, 16),
(12)                        (7, 17),
(12)                        (7, 18),
(12)                        (7, 19),
(12)                        (7, 20),
(12)                        (8, 11),
(12)                        (8, 12),
(12)                        (8, 13),
(12)                        (8, 14),
```

```
281: (12)                        (8, 15),
282: (12)                        (8, 16),
283: (12)                        (8, 17),
284: (12)                        (8, 18),
285: (12)                        (8, 19),
286: (12)                        (8, 20),
287: (12)                        (9, 11),
288: (12)                        (9, 12),
289: (12)                        (9, 13),
290: (12)                        (9, 14),
291: (12)                        (9, 15),
292: (12)                        (9, 16),
293: (12)                        (9, 17),
294: (12)                        (9, 18),
295: (12)                        (9, 19),
296: (12)                        (9, 20),
297: (12)                        (10, 11),
298: (12)                        (10, 12),
299: (12)                        (10, 13),
300: (12)                        (10, 14),
301: (12)                        (10, 15),
302: (12)                        (10, 16),
303: (12)                        (10, 17),
304: (12)                        (10, 18),
305: (12)                        (10, 19),
306: (12)                        (10, 20),
307: (8)                    ]
308: (4)                )
309: (4)                assert nx.bipartite.maximal_extendability(G) == 7
```

File 125 - test_generators.py:

```
1: (0)              import numbers
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from ..generators import (
5: (4)                  alternating_havel_hakimi_graph,
6: (4)                  complete_bipartite_graph,
7: (4)                  configuration_model,
8: (4)                  gnmk_random_graph,
9: (4)                  havel_hakimi_graph,
10: (4)                 preferential_attachment_graph,
11: (4)                 random_graph,
12: (4)                 reverse_havel_hakimi_graph,
13: (0)             )
14: (0)             """
15: (0)             Generators - Bipartite
16: (0)             ----------------------
17: (0)             """
18: (0)             class TestGeneratorsBipartite:
19: (4)                 def test_complete_bipartite_graph(self):
20: (8)                     G = complete_bipartite_graph(0, 0)
21: (8)                     assert nx.is_isomorphic(G, nx.null_graph())
22: (8)                     for i in [1, 5]:
23: (12)                        G = complete_bipartite_graph(i, 0)
24: (12)                        assert nx.is_isomorphic(G, nx.empty_graph(i))
25: (12)                        G = complete_bipartite_graph(0, i)
26: (12)                        assert nx.is_isomorphic(G, nx.empty_graph(i))
27: (8)                     G = complete_bipartite_graph(2, 2)
28: (8)                     assert nx.is_isomorphic(G, nx.cycle_graph(4))
29: (8)                     G = complete_bipartite_graph(1, 5)
30: (8)                     assert nx.is_isomorphic(G, nx.star_graph(5))
31: (8)                     G = complete_bipartite_graph(5, 1)
32: (8)                     assert nx.is_isomorphic(G, nx.star_graph(5))
33: (8)                     for m1, m2 in [(5, 11), (7, 3)]:
34: (12)                        G = complete_bipartite_graph(m1, m2)
35: (12)                        assert nx.number_of_nodes(G) == m1 + m2
```

```
36: (12)                        assert nx.number_of_edges(G) == m1 * m2
37: (8)                     with pytest.raises(nx.NetworkXError):
38: (12)                        complete_bipartite_graph(7, 3, create_using=nx.DiGraph)
39: (8)                     with pytest.raises(nx.NetworkXError):
40: (12)                        complete_bipartite_graph(7, 3, create_using=nx.MultiDiGraph)
41: (8)                     mG = complete_bipartite_graph(7, 3, create_using=nx.MultiGraph)
42: (8)                     assert mG.is_multigraph()
43: (8)                     assert sorted(mG.edges()) == sorted(G.edges())
44: (8)                     mG = complete_bipartite_graph(7, 3, create_using=nx.MultiGraph)
45: (8)                     assert mG.is_multigraph()
46: (8)                     assert sorted(mG.edges()) == sorted(G.edges())
47: (8)                     mG = complete_bipartite_graph(7, 3)  # default to Graph
48: (8)                     assert sorted(mG.edges()) == sorted(G.edges())
49: (8)                     assert not mG.is_multigraph()
50: (8)                     assert not mG.is_directed()
51: (8)                     for n1, n2 in [([1, 2], "ab"), (3, 2), (3, "ab"), ("ab", 3)]:
52: (12)                        G = complete_bipartite_graph(n1, n2)
53: (12)                        if isinstance(n1, numbers.Integral):
54: (16)                            if isinstance(n2, numbers.Integral):
55: (20)                                n2 = range(n1, n1 + n2)
56: (16)                            n1 = range(n1)
57: (12)                        elif isinstance(n2, numbers.Integral):
58: (16)                            n2 = range(n2)
59: (12)                        edges = {(u, v) for u in n1 for v in n2}
60: (12)                        assert edges == set(G.edges)
61: (12)                        assert G.size() == len(edges)
62: (8)                     for n1, n2 in [([1, 2], 3), (3, [1, 2]), ("abc", "bcd")]:
63: (12)                        pytest.raises(nx.NetworkXError, complete_bipartite_graph, n1, n2)
64: (4)                 def test_configuration_model(self):
65: (8)                     aseq = []
66: (8)                     bseq = []
67: (8)                     G = configuration_model(aseq, bseq)
68: (8)                     assert len(G) == 0
69: (8)                     aseq = [0, 0]
70: (8)                     bseq = [0, 0]
71: (8)                     G = configuration_model(aseq, bseq)
72: (8)                     assert len(G) == 4
73: (8)                     assert G.number_of_edges() == 0
74: (8)                     aseq = [3, 3, 3, 3]
75: (8)                     bseq = [2, 2, 2, 2, 2]
76: (8)                     pytest.raises(nx.NetworkXError, configuration_model, aseq, bseq)
77: (8)                     aseq = [3, 3, 3, 3]
78: (8)                     bseq = [2, 2, 2, 2, 2, 2]
79: (8)                     G = configuration_model(aseq, bseq)
80: (8)                     assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
81: (8)                     aseq = [2, 2, 2, 2, 2, 2]
82: (8)                     bseq = [3, 3, 3, 3]
83: (8)                     G = configuration_model(aseq, bseq)
84: (8)                     assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
85: (8)                     aseq = [2, 2, 2, 1, 1, 1]
86: (8)                     bseq = [3, 3, 3]
87: (8)                     G = configuration_model(aseq, bseq)
88: (8)                     assert G.is_multigraph()
89: (8)                     assert not G.is_directed()
90: (8)                     assert sorted(d for n, d in G.degree()) == [1, 1, 1, 2, 2, 2, 3, 3, 3]
91: (8)                     GU = nx.projected_graph(nx.Graph(G), range(len(aseq)))
92: (8)                     assert GU.number_of_nodes() == 6
93: (8)                     GD = nx.projected_graph(nx.Graph(G), range(len(aseq), len(aseq) +
len(bseq)))
94: (8)                     assert GD.number_of_nodes() == 3
95: (8)                     G = reverse_havel_hakimi_graph(aseq, bseq, create_using=nx.Graph)
96: (8)                     assert not G.is_multigraph()
97: (8)                     assert not G.is_directed()
98: (8)                     pytest.raises(
99: (12)                        nx.NetworkXError, configuration_model, aseq, bseq,
create_using=nx.DiGraph()
100: (8)                    )
```

```
101: (8)                    pytest.raises(
102: (12)                       nx.NetworkXError, configuration_model, aseq, bseq,
create_using=nx.DiGraph
103: (8)                    )
104: (8)                    pytest.raises(
105: (12)                       nx.NetworkXError,
106: (12)                       configuration_model,
107: (12)                       aseq,
108: (12)                       bseq,
109: (12)                       create_using=nx.MultiDiGraph,
110: (8)                    )
111: (4)                def test_havel_hakimi_graph(self):
112: (8)                    aseq = []
113: (8)                    bseq = []
114: (8)                    G = havel_hakimi_graph(aseq, bseq)
115: (8)                    assert len(G) == 0
116: (8)                    aseq = [0, 0]
117: (8)                    bseq = [0, 0]
118: (8)                    G = havel_hakimi_graph(aseq, bseq)
119: (8)                    assert len(G) == 4
120: (8)                    assert G.number_of_edges() == 0
121: (8)                    aseq = [3, 3, 3, 3]
122: (8)                    bseq = [2, 2, 2, 2, 2]
123: (8)                    pytest.raises(nx.NetworkXError, havel_hakimi_graph, aseq, bseq)
124: (8)                    bseq = [2, 2, 2, 2, 2, 2]
125: (8)                    G = havel_hakimi_graph(aseq, bseq)
126: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
127: (8)                    aseq = [2, 2, 2, 2, 2, 2]
128: (8)                    bseq = [3, 3, 3, 3]
129: (8)                    G = havel_hakimi_graph(aseq, bseq)
130: (8)                    assert G.is_multigraph()
131: (8)                    assert not G.is_directed()
132: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
133: (8)                    GU = nx.projected_graph(nx.Graph(G), range(len(aseq)))
134: (8)                    assert GU.number_of_nodes() == 6
135: (8)                    GD = nx.projected_graph(nx.Graph(G), range(len(aseq), len(aseq) +
len(bseq)))
136: (8)                    assert GD.number_of_nodes() == 4
137: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq, create_using=nx.Graph)
138: (8)                    assert not G.is_multigraph()
139: (8)                    assert not G.is_directed()
140: (8)                    pytest.raises(
141: (12)                       nx.NetworkXError, havel_hakimi_graph, aseq, bseq,
create_using=nx.DiGraph
142: (8)                    )
143: (8)                    pytest.raises(
144: (12)                       nx.NetworkXError, havel_hakimi_graph, aseq, bseq,
create_using=nx.DiGraph
145: (8)                    )
146: (8)                    pytest.raises(
147: (12)                       nx.NetworkXError,
148: (12)                       havel_hakimi_graph,
149: (12)                       aseq,
150: (12)                       bseq,
151: (12)                       create_using=nx.MultiDiGraph,
152: (8)                    )
153: (4)                def test_reverse_havel_hakimi_graph(self):
154: (8)                    aseq = []
155: (8)                    bseq = []
156: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq)
157: (8)                    assert len(G) == 0
158: (8)                    aseq = [0, 0]
159: (8)                    bseq = [0, 0]
160: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq)
161: (8)                    assert len(G) == 4
162: (8)                    assert G.number_of_edges() == 0
163: (8)                    aseq = [3, 3, 3, 3]
```

```
164: (8)                    bseq = [2, 2, 2, 2, 2]
165: (8)                    pytest.raises(nx.NetworkXError, reverse_havel_hakimi_graph, aseq,
bseq)
166: (8)                    bseq = [2, 2, 2, 2, 2, 2]
167: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq)
168: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
169: (8)                    aseq = [2, 2, 2, 2, 2, 2]
170: (8)                    bseq = [3, 3, 3, 3]
171: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq)
172: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
173: (8)                    aseq = [2, 2, 2, 1, 1, 1]
174: (8)                    bseq = [3, 3, 3]
175: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq)
176: (8)                    assert G.is_multigraph()
177: (8)                    assert not G.is_directed()
178: (8)                    assert sorted(d for n, d in G.degree()) == [1, 1, 1, 2, 2, 2, 3, 3, 3]
179: (8)                    GU = nx.projected_graph(nx.Graph(G), range(len(aseq)))
180: (8)                    assert GU.number_of_nodes() == 6
181: (8)                    GD = nx.projected_graph(nx.Graph(G), range(len(aseq), len(aseq) +
len(bseq)))
182: (8)                    assert GD.number_of_nodes() == 3
183: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq, create_using=nx.Graph)
184: (8)                    assert not G.is_multigraph()
185: (8)                    assert not G.is_directed()
186: (8)                    pytest.raises(
187: (12)                       nx.NetworkXError,
188: (12)                       reverse_havel_hakimi_graph,
189: (12)                       aseq,
190: (12)                       bseq,
191: (12)                       create_using=nx.DiGraph,
192: (8)                    )
193: (8)                    pytest.raises(
194: (12)                       nx.NetworkXError,
195: (12)                       reverse_havel_hakimi_graph,
196: (12)                       aseq,
197: (12)                       bseq,
198: (12)                       create_using=nx.DiGraph,
199: (8)                    )
200: (8)                    pytest.raises(
201: (12)                       nx.NetworkXError,
202: (12)                       reverse_havel_hakimi_graph,
203: (12)                       aseq,
204: (12)                       bseq,
205: (12)                       create_using=nx.MultiDiGraph,
206: (8)                    )
207: (4)                def test_alternating_havel_hakimi_graph(self):
208: (8)                    aseq = []
209: (8)                    bseq = []
210: (8)                    G = alternating_havel_hakimi_graph(aseq, bseq)
211: (8)                    assert len(G) == 0
212: (8)                    aseq = [0, 0]
213: (8)                    bseq = [0, 0]
214: (8)                    G = alternating_havel_hakimi_graph(aseq, bseq)
215: (8)                    assert len(G) == 4
216: (8)                    assert G.number_of_edges() == 0
217: (8)                    aseq = [3, 3, 3, 3]
218: (8)                    bseq = [2, 2, 2, 2, 2]
219: (8)                    pytest.raises(nx.NetworkXError, alternating_havel_hakimi_graph, aseq,
bseq)
220: (8)                    bseq = [2, 2, 2, 2, 2, 2]
221: (8)                    G = alternating_havel_hakimi_graph(aseq, bseq)
222: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
3]
223: (8)                    aseq = [2, 2, 2, 2, 2, 2]
224: (8)                    bseq = [3, 3, 3, 3]
225: (8)                    G = alternating_havel_hakimi_graph(aseq, bseq)
226: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 2, 2, 2, 2, 3, 3, 3,
```

```
3]
227: (8)                    aseq = [2, 2, 2, 1, 1, 1]
228: (8)                    bseq = [3, 3, 3]
229: (8)                    G = alternating_havel_hakimi_graph(aseq, bseq)
230: (8)                    assert G.is_multigraph()
231: (8)                    assert not G.is_directed()
232: (8)                    assert sorted(d for n, d in G.degree()) == [1, 1, 1, 2, 2, 2, 3, 3, 3]
233: (8)                    GU = nx.projected_graph(nx.Graph(G), range(len(aseq)))
234: (8)                    assert GU.number_of_nodes() == 6
235: (8)                    GD = nx.projected_graph(nx.Graph(G), range(len(aseq), len(aseq) +
len(bseq)))
236: (8)                    assert GD.number_of_nodes() == 3
237: (8)                    G = reverse_havel_hakimi_graph(aseq, bseq, create_using=nx.Graph)
238: (8)                    assert not G.is_multigraph()
239: (8)                    assert not G.is_directed()
240: (8)                    pytest.raises(
241: (12)                       nx.NetworkXError,
242: (12)                       alternating_havel_hakimi_graph,
243: (12)                       aseq,
244: (12)                       bseq,
245: (12)                       create_using=nx.DiGraph,
246: (8)                    )
247: (8)                    pytest.raises(
248: (12)                       nx.NetworkXError,
249: (12)                       alternating_havel_hakimi_graph,
250: (12)                       aseq,
251: (12)                       bseq,
252: (12)                       create_using=nx.DiGraph,
253: (8)                    )
254: (8)                    pytest.raises(
255: (12)                       nx.NetworkXError,
256: (12)                       alternating_havel_hakimi_graph,
257: (12)                       aseq,
258: (12)                       bseq,
259: (12)                       create_using=nx.MultiDiGraph,
260: (8)                    )
261: (4)                def test_preferential_attachment(self):
262: (8)                    aseq = [3, 2, 1, 1]
263: (8)                    G = preferential_attachment_graph(aseq, 0.5)
264: (8)                    assert G.is_multigraph()
265: (8)                    assert not G.is_directed()
266: (8)                    G = preferential_attachment_graph(aseq, 0.5, create_using=nx.Graph)
267: (8)                    assert not G.is_multigraph()
268: (8)                    assert not G.is_directed()
269: (8)                    pytest.raises(
270: (12)                       nx.NetworkXError,
271: (12)                       preferential_attachment_graph,
272: (12)                       aseq,
273: (12)                       0.5,
274: (12)                       create_using=nx.DiGraph(),
275: (8)                    )
276: (8)                    pytest.raises(
277: (12)                       nx.NetworkXError,
278: (12)                       preferential_attachment_graph,
279: (12)                       aseq,
280: (12)                       0.5,
281: (12)                       create_using=nx.DiGraph(),
282: (8)                    )
283: (8)                    pytest.raises(
284: (12)                       nx.NetworkXError,
285: (12)                       preferential_attachment_graph,
286: (12)                       aseq,
287: (12)                       0.5,
288: (12)                       create_using=nx.DiGraph(),
289: (8)                    )
290: (4)                def test_random_graph(self):
291: (8)                    n = 10
292: (8)                    m = 20
293: (8)                    G = random_graph(n, m, 0.9)
```

```
294: (8)                    assert len(G) == 30
295: (8)                    assert nx.is_bipartite(G)
296: (8)                    X, Y = nx.algorithms.bipartite.sets(G)
297: (8)                    assert set(range(n)) == X
298: (8)                    assert set(range(n, n + m)) == Y
299: (4)                def test_random_digraph(self):
300: (8)                    n = 10
301: (8)                    m = 20
302: (8)                    G = random_graph(n, m, 0.9, directed=True)
303: (8)                    assert len(G) == 30
304: (8)                    assert nx.is_bipartite(G)
305: (8)                    X, Y = nx.algorithms.bipartite.sets(G)
306: (8)                    assert set(range(n)) == X
307: (8)                    assert set(range(n, n + m)) == Y
308: (4)                def test_gnmk_random_graph(self):
309: (8)                    n = 10
310: (8)                    m = 20
311: (8)                    edges = 100
312: (8)                    G = gnmk_random_graph(n, m, edges, seed=1234)
313: (8)                    assert len(G) == n + m
314: (8)                    assert nx.is_bipartite(G)
315: (8)                    X, Y = nx.algorithms.bipartite.sets(G)
316: (8)                    assert set(range(n)) == X
317: (8)                    assert set(range(n, n + m)) == Y
318: (8)                    assert edges == len(list(G.edges()))
319: (4)                def test_gnmk_random_graph_complete(self):
320: (8)                    n = 10
321: (8)                    m = 20
322: (8)                    edges = 200
323: (8)                    G = gnmk_random_graph(n, m, edges)
324: (8)                    assert len(G) == n + m
325: (8)                    assert nx.is_bipartite(G)
326: (8)                    X, Y = nx.algorithms.bipartite.sets(G)
327: (8)                    assert set(range(n)) == X
328: (8)                    assert set(range(n, n + m)) == Y
329: (8)                    assert edges == len(list(G.edges()))

----------------------------------------

File 126 - test_matching.py:

1: (0)              """Unit tests for the :mod:`networkx.algorithms.bipartite.matching` module."""
2: (0)              import itertools
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.bipartite.matching import (
6: (4)                  eppstein_matching,
7: (4)                  hopcroft_karp_matching,
8: (4)                  maximum_matching,
9: (4)                  minimum_weight_full_matching,
10: (4)                 to_vertex_cover,
11: (0)             )
12: (0)             class TestMatching:
13: (4)                 """Tests for bipartite matching algorithms."""
14: (4)                 def setup_method(self):
15: (8)                     """Creates a bipartite graph for use in testing matching algorithms.
16: (8)                     The bipartite graph has a maximum cardinality matching that leaves
17: (8)                     vertex 1 and vertex 10 unmatched. The first six numbers are the left
18: (8)                     vertices and the next six numbers are the right vertices.
19: (8)                     """
20: (8)                     self.simple_graph = nx.complete_bipartite_graph(2, 3)
21: (8)                     self.simple_solution = {0: 2, 1: 3, 2: 0, 3: 1}
22: (8)                     edges = [(0, 7), (0, 8), (2, 6), (2, 9), (3, 8), (4, 8), (4, 9), (5,
11)]
23: (8)                     self.top_nodes = set(range(6))
24: (8)                     self.graph = nx.Graph()
25: (8)                     self.graph.add_nodes_from(range(12))
26: (8)                     self.graph.add_edges_from(edges)
27: (8)                     G = nx.Graph()
```

```
(8)                     G.add_nodes_from(
(12)                        [
(16)                            (1, "C"),
(16)                            (1, "B"),
(16)                            (0, "G"),
(16)                            (1, "F"),
(16)                            (1, "E"),
(16)                            (0, "C"),
(16)                            (1, "D"),
(16)                            (1, "I"),
(16)                            (0, "A"),
(16)                            (0, "D"),
(16)                            (0, "F"),
(16)                            (0, "E"),
(16)                            (0, "H"),
(16)                            (1, "G"),
(16)                            (1, "A"),
(16)                            (0, "I"),
(16)                            (0, "B"),
(16)                            (1, "H"),
(12)                        ]
(8)                     )
(8)                     G.add_edge((1, "C"), (0, "A"))
(8)                     G.add_edge((1, "B"), (0, "A"))
(8)                     G.add_edge((0, "G"), (1, "I"))
(8)                     G.add_edge((0, "G"), (1, "H"))
(8)                     G.add_edge((1, "F"), (0, "A"))
(8)                     G.add_edge((1, "F"), (0, "C"))
(8)                     G.add_edge((1, "F"), (0, "E"))
(8)                     G.add_edge((1, "E"), (0, "A"))
(8)                     G.add_edge((1, "E"), (0, "C"))
(8)                     G.add_edge((0, "C"), (1, "D"))
(8)                     G.add_edge((0, "C"), (1, "I"))
(8)                     G.add_edge((0, "C"), (1, "G"))
(8)                     G.add_edge((0, "C"), (1, "H"))
(8)                     G.add_edge((1, "D"), (0, "A"))
(8)                     G.add_edge((1, "I"), (0, "A"))
(8)                     G.add_edge((1, "I"), (0, "E"))
(8)                     G.add_edge((0, "A"), (1, "G"))
(8)                     G.add_edge((0, "A"), (1, "H"))
(8)                     G.add_edge((0, "E"), (1, "G"))
(8)                     G.add_edge((0, "E"), (1, "H"))
(8)                     self.disconnected_graph = G
(4)                 def check_match(self, matching):
(8)                     """Asserts that the matching is what we expect from the bipartite
graph
(8)                     constructed in the :meth:`setup` fixture.
(8)                     """
(8)                     M = matching
(8)                     matched_vertices = frozenset(itertools.chain(*M.items()))
(8)                     assert matched_vertices == frozenset(range(12)) - {1, 10}
(8)                     assert all(u == M[M[u]] for u in range(12) if u in M)
(4)                 def check_vertex_cover(self, vertices):
(8)                     """Asserts that the given set of vertices is the vertex cover we
(8)                     expected from the bipartite graph constructed in the :meth:`setup`
(8)                     fixture.
(8)                     """
(8)                     assert len(vertices) == 5
(8)                     for u, v in self.graph.edges():
(12)                        assert u in vertices or v in vertices
(4)                 def test_eppstein_matching(self):
(8)                     """Tests that David Eppstein's implementation of the Hopcroft--Karp
(8)                     algorithm produces a maximum cardinality matching.
(8)                     """
(8)                     self.check_match(eppstein_matching(self.graph, self.top_nodes))
(4)                 def test_hopcroft_karp_matching(self):
(8)                     """Tests that the Hopcroft--Karp algorithm produces a maximum
(8)                     cardinality matching in a bipartite graph.
(8)                     """
```

```
96: (8)                     self.check_match(hopcroft_karp_matching(self.graph, self.top_nodes))
97: (4)                 def test_to_vertex_cover(self):
98: (8)                     """Test for converting a maximum matching to a minimum vertex
cover."""
99: (8)                     matching = maximum_matching(self.graph, self.top_nodes)
100: (8)                    vertex_cover = to_vertex_cover(self.graph, matching, self.top_nodes)
101: (8)                    self.check_vertex_cover(vertex_cover)
102: (4)                def test_eppstein_matching_simple(self):
103: (8)                    match = eppstein_matching(self.simple_graph)
104: (8)                    assert match == self.simple_solution
105: (4)                def test_hopcroft_karp_matching_simple(self):
106: (8)                    match = hopcroft_karp_matching(self.simple_graph)
107: (8)                    assert match == self.simple_solution
108: (4)                def test_eppstein_matching_disconnected(self):
109: (8)                    with pytest.raises(nx.AmbiguousSolution):
110: (12)                       match = eppstein_matching(self.disconnected_graph)
111: (4)                def test_hopcroft_karp_matching_disconnected(self):
112: (8)                    with pytest.raises(nx.AmbiguousSolution):
113: (12)                       match = hopcroft_karp_matching(self.disconnected_graph)
114: (4)                def test_issue_2127(self):
115: (8)                    """Test from issue 2127"""
116: (8)                    G = nx.DiGraph()
117: (8)                    G.add_edge("A", "C")
118: (8)                    G.add_edge("A", "B")
119: (8)                    G.add_edge("C", "E")
120: (8)                    G.add_edge("C", "D")
121: (8)                    G.add_edge("E", "G")
122: (8)                    G.add_edge("E", "F")
123: (8)                    G.add_edge("G", "I")
124: (8)                    G.add_edge("G", "H")
125: (8)                    tc = nx.transitive_closure(G)
126: (8)                    btc = nx.Graph()
127: (8)                    for v in tc.nodes():
128: (12)                       btc.add_node((0, v))
129: (12)                       btc.add_node((1, v))
130: (8)                    for u, v in tc.edges():
131: (12)                       btc.add_edge((0, u), (1, v))
132: (8)                    top_nodes = {n for n in btc if n[0] == 0}
133: (8)                    matching = hopcroft_karp_matching(btc, top_nodes)
134: (8)                    vertex_cover = to_vertex_cover(btc, matching, top_nodes)
135: (8)                    independent_set = set(G) - {v for _, v in vertex_cover}
136: (8)                    assert {"B", "D", "F", "I", "H"} == independent_set
137: (4)                def test_vertex_cover_issue_2384(self):
138: (8)                    G = nx.Graph([(0, 3), (1, 3), (1, 4), (2, 3)])
139: (8)                    matching = maximum_matching(G)
140: (8)                    vertex_cover = to_vertex_cover(G, matching)
141: (8)                    for u, v in G.edges():
142: (12)                       assert u in vertex_cover or v in vertex_cover
143: (4)                def test_vertex_cover_issue_3306(self):
144: (8)                    G = nx.Graph()
145: (8)                    edges = [(0, 2), (1, 0), (1, 1), (1, 2), (2, 2)]
146: (8)                    G.add_edges_from([((i, "L"), (j, "R")) for i, j in edges])
147: (8)                    matching = maximum_matching(G)
148: (8)                    vertex_cover = to_vertex_cover(G, matching)
149: (8)                    for u, v in G.edges():
150: (12)                       assert u in vertex_cover or v in vertex_cover
151: (4)                def test_unorderable_nodes(self):
152: (8)                    a = object()
153: (8)                    b = object()
154: (8)                    c = object()
155: (8)                    d = object()
156: (8)                    e = object()
157: (8)                    G = nx.Graph([(a, d), (b, d), (b, e), (c, d)])
158: (8)                    matching = maximum_matching(G)
159: (8)                    vertex_cover = to_vertex_cover(G, matching)
160: (8)                    for u, v in G.edges():
161: (12)                       assert u in vertex_cover or v in vertex_cover
162: (0)            def test_eppstein_matching():
163: (4)                """Test in accordance to issue #1927"""
```

```
(4)                 G = nx.Graph()
(4)                 G.add_nodes_from(["a", 2, 3, 4], bipartite=0)
(4)                 G.add_nodes_from([1, "b", "c"], bipartite=1)
(4)                 G.add_edges_from([("a", 1), ("a", "b"), (2, "b"), (2, "c"), (3, "c"), (4,
1)])
(4)                 matching = eppstein_matching(G)
(4)                 assert len(matching) == len(maximum_matching(G))
(4)                 assert all(x in set(matching.keys()) for x in set(matching.values()))
(0)             class TestMinimumWeightFullMatching:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     pytest.importorskip("scipy")
(4)                 def test_minimum_weight_full_matching_incomplete_graph(self):
(8)                     B = nx.Graph()
(8)                     B.add_nodes_from([1, 2], bipartite=0)
(8)                     B.add_nodes_from([3, 4], bipartite=1)
(8)                     B.add_edge(1, 4, weight=100)
(8)                     B.add_edge(2, 3, weight=100)
(8)                     B.add_edge(2, 4, weight=50)
(8)                     matching = minimum_weight_full_matching(B)
(8)                     assert matching == {1: 4, 2: 3, 4: 1, 3: 2}
(4)                 def test_minimum_weight_full_matching_with_no_full_matching(self):
(8)                     B = nx.Graph()
(8)                     B.add_nodes_from([1, 2, 3], bipartite=0)
(8)                     B.add_nodes_from([4, 5, 6], bipartite=1)
(8)                     B.add_edge(1, 4, weight=100)
(8)                     B.add_edge(2, 4, weight=100)
(8)                     B.add_edge(3, 4, weight=50)
(8)                     B.add_edge(3, 5, weight=50)
(8)                     B.add_edge(3, 6, weight=50)
(8)                     with pytest.raises(ValueError):
(12)                        minimum_weight_full_matching(B)
(4)                 def test_minimum_weight_full_matching_square(self):
(8)                     G = nx.complete_bipartite_graph(3, 3)
(8)                     G.add_edge(0, 3, weight=400)
(8)                     G.add_edge(0, 4, weight=150)
(8)                     G.add_edge(0, 5, weight=400)
(8)                     G.add_edge(1, 3, weight=400)
(8)                     G.add_edge(1, 4, weight=450)
(8)                     G.add_edge(1, 5, weight=600)
(8)                     G.add_edge(2, 3, weight=300)
(8)                     G.add_edge(2, 4, weight=225)
(8)                     G.add_edge(2, 5, weight=300)
(8)                     matching = minimum_weight_full_matching(G)
(8)                     assert matching == {0: 4, 1: 3, 2: 5, 4: 0, 3: 1, 5: 2}
(4)                 def test_minimum_weight_full_matching_smaller_left(self):
(8)                     G = nx.complete_bipartite_graph(3, 4)
(8)                     G.add_edge(0, 3, weight=400)
(8)                     G.add_edge(0, 4, weight=150)
(8)                     G.add_edge(0, 5, weight=400)
(8)                     G.add_edge(0, 6, weight=1)
(8)                     G.add_edge(1, 3, weight=400)
(8)                     G.add_edge(1, 4, weight=450)
(8)                     G.add_edge(1, 5, weight=600)
(8)                     G.add_edge(1, 6, weight=2)
(8)                     G.add_edge(2, 3, weight=300)
(8)                     G.add_edge(2, 4, weight=225)
(8)                     G.add_edge(2, 5, weight=290)
(8)                     G.add_edge(2, 6, weight=3)
(8)                     matching = minimum_weight_full_matching(G)
(8)                     assert matching == {0: 4, 1: 6, 2: 5, 4: 0, 5: 2, 6: 1}
(4)                 def test_minimum_weight_full_matching_smaller_top_nodes_right(self):
(8)                     G = nx.complete_bipartite_graph(3, 4)
(8)                     G.add_edge(0, 3, weight=400)
(8)                     G.add_edge(0, 4, weight=150)
(8)                     G.add_edge(0, 5, weight=400)
(8)                     G.add_edge(0, 6, weight=1)
(8)                     G.add_edge(1, 3, weight=400)
(8)                     G.add_edge(1, 4, weight=450)
```

```
232: (8)                    G.add_edge(1, 5, weight=600)
233: (8)                    G.add_edge(1, 6, weight=2)
234: (8)                    G.add_edge(2, 3, weight=300)
235: (8)                    G.add_edge(2, 4, weight=225)
236: (8)                    G.add_edge(2, 5, weight=290)
237: (8)                    G.add_edge(2, 6, weight=3)
238: (8)                    matching = minimum_weight_full_matching(G, top_nodes=[3, 4, 5, 6])
239: (8)                    assert matching == {0: 4, 1: 6, 2: 5, 4: 0, 5: 2, 6: 1}
240: (4)                def test_minimum_weight_full_matching_smaller_right(self):
241: (8)                    G = nx.complete_bipartite_graph(4, 3)
242: (8)                    G.add_edge(0, 4, weight=400)
243: (8)                    G.add_edge(0, 5, weight=400)
244: (8)                    G.add_edge(0, 6, weight=300)
245: (8)                    G.add_edge(1, 4, weight=150)
246: (8)                    G.add_edge(1, 5, weight=450)
247: (8)                    G.add_edge(1, 6, weight=225)
248: (8)                    G.add_edge(2, 4, weight=400)
249: (8)                    G.add_edge(2, 5, weight=600)
250: (8)                    G.add_edge(2, 6, weight=290)
251: (8)                    G.add_edge(3, 4, weight=1)
252: (8)                    G.add_edge(3, 5, weight=2)
253: (8)                    G.add_edge(3, 6, weight=3)
254: (8)                    matching = minimum_weight_full_matching(G)
255: (8)                    assert matching == {1: 4, 2: 6, 3: 5, 4: 1, 5: 3, 6: 2}
256: (4)                def test_minimum_weight_full_matching_negative_weights(self):
257: (8)                    G = nx.complete_bipartite_graph(2, 2)
258: (8)                    G.add_edge(0, 2, weight=-2)
259: (8)                    G.add_edge(0, 3, weight=0.2)
260: (8)                    G.add_edge(1, 2, weight=-2)
261: (8)                    G.add_edge(1, 3, weight=0.3)
262: (8)                    matching = minimum_weight_full_matching(G)
263: (8)                    assert matching == {0: 3, 1: 2, 2: 1, 3: 0}
264: (4)                def test_minimum_weight_full_matching_different_weight_key(self):
265: (8)                    G = nx.complete_bipartite_graph(2, 2)
266: (8)                    G.add_edge(0, 2, mass=2)
267: (8)                    G.add_edge(0, 3, mass=0.2)
268: (8)                    G.add_edge(1, 2, mass=1)
269: (8)                    G.add_edge(1, 3, mass=2)
270: (8)                    matching = minimum_weight_full_matching(G, weight="mass")
271: (8)                    assert matching == {0: 3, 1: 2, 2: 1, 3: 0}

----------------------------------------

File 127 - test_matrix.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              sp = pytest.importorskip("scipy")
4: (0)              sparse = pytest.importorskip("scipy.sparse")
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms import bipartite
7: (0)              from networkx.utils import edges_equal
8: (0)              class TestBiadjacencyMatrix:
9: (4)                  def test_biadjacency_matrix_weight(self):
10: (8)                     G = nx.path_graph(5)
11: (8)                     G.add_edge(0, 1, weight=2, other=4)
12: (8)                     X = [1, 3]
13: (8)                     Y = [0, 2, 4]
14: (8)                     M = bipartite.biadjacency_matrix(G, X, weight="weight")
15: (8)                     assert M[0, 0] == 2
16: (8)                     M = bipartite.biadjacency_matrix(G, X, weight="other")
17: (8)                     assert M[0, 0] == 4
18: (4)                 def test_biadjacency_matrix(self):
19: (8)                     tops = [2, 5, 10]
20: (8)                     bots = [5, 10, 15]
21: (8)                     for i in range(len(tops)):
22: (12)                        G = bipartite.random_graph(tops[i], bots[i], 0.2)
23: (12)                        top = [n for n, d in G.nodes(data=True) if d["bipartite"] == 0]
24: (12)                        M = bipartite.biadjacency_matrix(G, top)
```

```
25: (12)                        assert M.shape[0] == tops[i]
26: (12)                        assert M.shape[1] == bots[i]
27: (4)                 def test_biadjacency_matrix_order(self):
28: (8)                     G = nx.path_graph(5)
29: (8)                     G.add_edge(0, 1, weight=2)
30: (8)                     X = [3, 1]
31: (8)                     Y = [4, 2, 0]
32: (8)                     M = bipartite.biadjacency_matrix(G, X, Y, weight="weight")
33: (8)                     assert M[1, 2] == 2
34: (4)                 def test_biadjacency_matrix_empty_graph(self):
35: (8)                     G = nx.empty_graph(2)
36: (8)                     M = nx.bipartite.biadjacency_matrix(G, [0])
37: (8)                     assert np.array_equal(M.toarray(), np.array([[0]]))
38: (4)                 def test_null_graph(self):
39: (8)                     with pytest.raises(nx.NetworkXError):
40: (12)                        bipartite.biadjacency_matrix(nx.Graph(), [])
41: (4)                 def test_empty_graph(self):
42: (8)                     with pytest.raises(nx.NetworkXError):
43: (12)                        bipartite.biadjacency_matrix(nx.Graph([(1, 0)]), [])
44: (4)                 def test_duplicate_row(self):
45: (8)                     with pytest.raises(nx.NetworkXError):
46: (12)                        bipartite.biadjacency_matrix(nx.Graph([(1, 0)]), [1, 1])
47: (4)                 def test_duplicate_col(self):
48: (8)                     with pytest.raises(nx.NetworkXError):
49: (12)                        bipartite.biadjacency_matrix(nx.Graph([(1, 0)]), [0], [1, 1])
50: (4)                 def test_format_keyword(self):
51: (8)                     with pytest.raises(nx.NetworkXError):
52: (12)                        bipartite.biadjacency_matrix(nx.Graph([(1, 0)]), [0],
format="foo")
53: (4)                 def test_from_biadjacency_roundtrip(self):
54: (8)                     B1 = nx.path_graph(5)
55: (8)                     M = bipartite.biadjacency_matrix(B1, [0, 2, 4])
56: (8)                     B2 = bipartite.from_biadjacency_matrix(M)
57: (8)                     assert nx.is_isomorphic(B1, B2)
58: (4)                 def test_from_biadjacency_weight(self):
59: (8)                     M = sparse.csc_matrix([[1, 2], [0, 3]])
60: (8)                     B = bipartite.from_biadjacency_matrix(M)
61: (8)                     assert edges_equal(B.edges(), [(0, 2), (0, 3), (1, 3)])
62: (8)                     B = bipartite.from_biadjacency_matrix(M, edge_attribute="weight")
63: (8)                     e = [(0, 2, {"weight": 1}), (0, 3, {"weight": 2}), (1, 3, {"weight":
3})]
64: (8)                     assert edges_equal(B.edges(data=True), e)
65: (4)                 def test_from_biadjacency_multigraph(self):
66: (8)                     M = sparse.csc_matrix([[1, 2], [0, 3]])
67: (8)                     B = bipartite.from_biadjacency_matrix(M, create_using=nx.MultiGraph())
68: (8)                     assert edges_equal(B.edges(), [(0, 2), (0, 3), (0, 3), (1, 3), (1, 3),
(1, 3)])
```

File 128 - test_project.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import bipartite
4: (0)              from networkx.utils import edges_equal, nodes_equal
5: (0)              class TestBipartiteProject:
6: (4)                  def test_path_projected_graph(self):
7: (8)                      G = nx.path_graph(4)
8: (8)                      P = bipartite.projected_graph(G, [1, 3])
9: (8)                      assert nodes_equal(list(P), [1, 3])
10: (8)                     assert edges_equal(list(P.edges()), [(1, 3)])
11: (8)                     P = bipartite.projected_graph(G, [0, 2])
12: (8)                     assert nodes_equal(list(P), [0, 2])
13: (8)                     assert edges_equal(list(P.edges()), [(0, 2)])
14: (8)                     G = nx.MultiGraph([(0, 1)])
15: (8)                     with pytest.raises(nx.NetworkXError, match="not defined for
multigraphs"):
16: (12)                        bipartite.projected_graph(G, [0])
```

```
(4)     def test_path_projected_properties_graph(self):
(8)         G = nx.path_graph(4)
(8)         G.add_node(1, name="one")
(8)         G.add_node(2, name="two")
(8)         P = bipartite.projected_graph(G, [1, 3])
(8)         assert nodes_equal(list(P), [1, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 3)])
(8)         assert P.nodes[1]["name"] == G.nodes[1]["name"]
(8)         P = bipartite.projected_graph(G, [0, 2])
(8)         assert nodes_equal(list(P), [0, 2])
(8)         assert edges_equal(list(P.edges()), [(0, 2)])
(8)         assert P.nodes[2]["name"] == G.nodes[2]["name"]
(4)     def test_path_collaboration_projected_graph(self):
(8)         G = nx.path_graph(4)
(8)         P = bipartite.collaboration_weighted_projected_graph(G, [1, 3])
(8)         assert nodes_equal(list(P), [1, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 3)])
(8)         P[1][3]["weight"] = 1
(8)         P = bipartite.collaboration_weighted_projected_graph(G, [0, 2])
(8)         assert nodes_equal(list(P), [0, 2])
(8)         assert edges_equal(list(P.edges()), [(0, 2)])
(8)         P[0][2]["weight"] = 1
(4)     def test_directed_path_collaboration_projected_graph(self):
(8)         G = nx.DiGraph()
(8)         nx.add_path(G, range(4))
(8)         P = bipartite.collaboration_weighted_projected_graph(G, [1, 3])
(8)         assert nodes_equal(list(P), [1, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 3)])
(8)         P[1][3]["weight"] = 1
(8)         P = bipartite.collaboration_weighted_projected_graph(G, [0, 2])
(8)         assert nodes_equal(list(P), [0, 2])
(8)         assert edges_equal(list(P.edges()), [(0, 2)])
(8)         P[0][2]["weight"] = 1
(4)     def test_path_weighted_projected_graph(self):
(8)         G = nx.path_graph(4)
(8)         with pytest.raises(nx.NetworkXAlgorithmError):
(12)            bipartite.weighted_projected_graph(G, [1, 2, 3, 3])
(8)         P = bipartite.weighted_projected_graph(G, [1, 3])
(8)         assert nodes_equal(list(P), [1, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 3)])
(8)         P[1][3]["weight"] = 1
(8)         P = bipartite.weighted_projected_graph(G, [0, 2])
(8)         assert nodes_equal(list(P), [0, 2])
(8)         assert edges_equal(list(P.edges()), [(0, 2)])
(8)         P[0][2]["weight"] = 1
(4)     def test_digraph_weighted_projection(self):
(8)         G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 4)])
(8)         P = bipartite.overlap_weighted_projected_graph(G, [1, 3])
(8)         assert nx.get_edge_attributes(P, "weight") == {(1, 3): 1.0}
(8)         assert len(P) == 2
(4)     def test_path_weighted_projected_directed_graph(self):
(8)         G = nx.DiGraph()
(8)         nx.add_path(G, range(4))
(8)         P = bipartite.weighted_projected_graph(G, [1, 3])
(8)         assert nodes_equal(list(P), [1, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 3)])
(8)         P[1][3]["weight"] = 1
(8)         P = bipartite.weighted_projected_graph(G, [0, 2])
(8)         assert nodes_equal(list(P), [0, 2])
(8)         assert edges_equal(list(P.edges()), [(0, 2)])
(8)         P[0][2]["weight"] = 1
(4)     def test_star_projected_graph(self):
(8)         G = nx.star_graph(3)
(8)         P = bipartite.projected_graph(G, [1, 2, 3])
(8)         assert nodes_equal(list(P), [1, 2, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 2), (1, 3), (2, 3)])
(8)         P = bipartite.weighted_projected_graph(G, [1, 2, 3])
(8)         assert nodes_equal(list(P), [1, 2, 3])
(8)         assert edges_equal(list(P.edges()), [(1, 2), (1, 3), (2, 3)])
```

```
86: (8)                     P = bipartite.projected_graph(G, [0])
87: (8)                     assert nodes_equal(list(P), [0])
88: (8)                     assert edges_equal(list(P.edges()), [])
89: (4)                 def test_project_multigraph(self):
90: (8)                     G = nx.Graph()
91: (8)                     G.add_edge("a", 1)
92: (8)                     G.add_edge("b", 1)
93: (8)                     G.add_edge("a", 2)
94: (8)                     G.add_edge("b", 2)
95: (8)                     P = bipartite.projected_graph(G, "ab")
96: (8)                     assert edges_equal(list(P.edges()), [("a", "b")])
97: (8)                     P = bipartite.weighted_projected_graph(G, "ab")
98: (8)                     assert edges_equal(list(P.edges()), [("a", "b")])
99: (8)                     P = bipartite.projected_graph(G, "ab", multigraph=True)
100: (8)                    assert edges_equal(list(P.edges()), [("a", "b"), ("a", "b")])
101: (4)                def test_project_collaboration(self):
102: (8)                    G = nx.Graph()
103: (8)                    G.add_edge("a", 1)
104: (8)                    G.add_edge("b", 1)
105: (8)                    G.add_edge("b", 2)
106: (8)                    G.add_edge("c", 2)
107: (8)                    G.add_edge("c", 3)
108: (8)                    G.add_edge("c", 4)
109: (8)                    G.add_edge("b", 4)
110: (8)                    P = bipartite.collaboration_weighted_projected_graph(G, "abc")
111: (8)                    assert P["a"]["b"]["weight"] == 1
112: (8)                    assert P["b"]["c"]["weight"] == 2
113: (4)                def test_directed_projection(self):
114: (8)                    G = nx.DiGraph()
115: (8)                    G.add_edge("A", 1)
116: (8)                    G.add_edge(1, "B")
117: (8)                    G.add_edge("A", 2)
118: (8)                    G.add_edge("B", 2)
119: (8)                    P = bipartite.projected_graph(G, "AB")
120: (8)                    assert edges_equal(list(P.edges()), [("A", "B")])
121: (8)                    P = bipartite.weighted_projected_graph(G, "AB")
122: (8)                    assert edges_equal(list(P.edges()), [("A", "B")])
123: (8)                    assert P["A"]["B"]["weight"] == 1
124: (8)                    P = bipartite.projected_graph(G, "AB", multigraph=True)
125: (8)                    assert edges_equal(list(P.edges()), [("A", "B")])
126: (8)                    G = nx.DiGraph()
127: (8)                    G.add_edge("A", 1)
128: (8)                    G.add_edge(1, "B")
129: (8)                    G.add_edge("A", 2)
130: (8)                    G.add_edge(2, "B")
131: (8)                    P = bipartite.projected_graph(G, "AB")
132: (8)                    assert edges_equal(list(P.edges()), [("A", "B")])
133: (8)                    P = bipartite.weighted_projected_graph(G, "AB")
134: (8)                    assert edges_equal(list(P.edges()), [("A", "B")])
135: (8)                    assert P["A"]["B"]["weight"] == 2
136: (8)                    P = bipartite.projected_graph(G, "AB", multigraph=True)
137: (8)                    assert edges_equal(list(P.edges()), [("A", "B"), ("A", "B")])
138: (0)            class TestBipartiteWeightedProjection:
139: (4)                @classmethod
140: (4)                def setup_class(cls):
141: (8)                    cls.G = nx.Graph()
142: (8)                    cls.G.add_edge("A", 1)
143: (8)                    cls.G.add_edge("A", 2)
144: (8)                    cls.G.add_edge("B", 1)
145: (8)                    cls.G.add_edge("B", 2)
146: (8)                    cls.G.add_edge("B", 3)
147: (8)                    cls.G.add_edge("B", 4)
148: (8)                    cls.G.add_edge("B", 5)
149: (8)                    cls.G.add_edge("C", 1)
150: (8)                    cls.G.add_edge("D", 3)
151: (8)                    cls.G.add_edge("E", 4)
152: (8)                    cls.G.add_edge("E", 5)
153: (8)                    cls.G.add_edge("E", 6)
154: (8)                    cls.G.add_edge("F", 6)
```

```
(8)                     cls.N = nx.Graph()
(8)                     cls.N.add_edge("A", 1)
(8)                     cls.N.add_edge("A", 2)
(8)                     cls.N.add_edge("A", 3)
(8)                     cls.N.add_edge("B", 1)
(8)                     cls.N.add_edge("B", 2)
(8)                     cls.N.add_edge("B", 3)
(8)                     cls.N.add_edge("C", 1)
(8)                     cls.N.add_edge("D", 1)
(8)                     cls.N.add_edge("E", 3)
(4)                 def test_project_weighted_shared(self):
(8)                     edges = [
(12)                        ("A", "B", 2),
(12)                        ("A", "C", 1),
(12)                        ("B", "C", 1),
(12)                        ("B", "D", 1),
(12)                        ("B", "E", 2),
(12)                        ("E", "F", 1),
(8)                     ]
(8)                     Panswer = nx.Graph()
(8)                     Panswer.add_weighted_edges_from(edges)
(8)                     P = bipartite.weighted_projected_graph(self.G, "ABCDEF")
(8)                     assert edges_equal(list(P.edges()), Panswer.edges())
(8)                     for u, v in list(P.edges()):
(12)                        assert P[u][v]["weight"] == Panswer[u][v]["weight"]
(8)                     edges = [
(12)                        ("A", "B", 3),
(12)                        ("A", "E", 1),
(12)                        ("A", "C", 1),
(12)                        ("A", "D", 1),
(12)                        ("B", "E", 1),
(12)                        ("B", "C", 1),
(12)                        ("B", "D", 1),
(12)                        ("C", "D", 1),
(8)                     ]
(8)                     Panswer = nx.Graph()
(8)                     Panswer.add_weighted_edges_from(edges)
(8)                     P = bipartite.weighted_projected_graph(self.N, "ABCDE")
(8)                     assert edges_equal(list(P.edges()), Panswer.edges())
(8)                     for u, v in list(P.edges()):
(12)                        assert P[u][v]["weight"] == Panswer[u][v]["weight"]
(4)                 def test_project_weighted_newman(self):
(8)                     edges = [
(12)                        ("A", "B", 1.5),
(12)                        ("A", "C", 0.5),
(12)                        ("B", "C", 0.5),
(12)                        ("B", "D", 1),
(12)                        ("B", "E", 2),
(12)                        ("E", "F", 1),
(8)                     ]
(8)                     Panswer = nx.Graph()
(8)                     Panswer.add_weighted_edges_from(edges)
(8)                     P = bipartite.collaboration_weighted_projected_graph(self.G, "ABCDEF")
(8)                     assert edges_equal(list(P.edges()), Panswer.edges())
(8)                     for u, v in list(P.edges()):
(12)                        assert P[u][v]["weight"] == Panswer[u][v]["weight"]
(8)                     edges = [
(12)                        ("A", "B", 11 / 6.0),
(12)                        ("A", "E", 1 / 2.0),
(12)                        ("A", "C", 1 / 3.0),
(12)                        ("A", "D", 1 / 3.0),
(12)                        ("B", "E", 1 / 2.0),
(12)                        ("B", "C", 1 / 3.0),
(12)                        ("B", "D", 1 / 3.0),
(12)                        ("C", "D", 1 / 3.0),
(8)                     ]
(8)                     Panswer = nx.Graph()
(8)                     Panswer.add_weighted_edges_from(edges)
(8)                     P = bipartite.collaboration_weighted_projected_graph(self.N, "ABCDE")
```

```
224: (8)                    assert edges_equal(list(P.edges()), Panswer.edges())
225: (8)                    for u, v in list(P.edges()):
226: (12)                       assert P[u][v]["weight"] == Panswer[u][v]["weight"]
227: (4)                def test_project_weighted_ratio(self):
228: (8)                    edges = [
229: (12)                       ("A", "B", 2 / 6.0),
230: (12)                       ("A", "C", 1 / 6.0),
231: (12)                       ("B", "C", 1 / 6.0),
232: (12)                       ("B", "D", 1 / 6.0),
233: (12)                       ("B", "E", 2 / 6.0),
234: (12)                       ("E", "F", 1 / 6.0),
235: (8)                    ]
236: (8)                    Panswer = nx.Graph()
237: (8)                    Panswer.add_weighted_edges_from(edges)
238: (8)                    P = bipartite.weighted_projected_graph(self.G, "ABCDEF", ratio=True)
239: (8)                    assert edges_equal(list(P.edges()), Panswer.edges())
240: (8)                    for u, v in list(P.edges()):
241: (12)                       assert P[u][v]["weight"] == Panswer[u][v]["weight"]
242: (8)                    edges = [
243: (12)                       ("A", "B", 3 / 3.0),
244: (12)                       ("A", "E", 1 / 3.0),
245: (12)                       ("A", "C", 1 / 3.0),
246: (12)                       ("A", "D", 1 / 3.0),
247: (12)                       ("B", "E", 1 / 3.0),
248: (12)                       ("B", "C", 1 / 3.0),
249: (12)                       ("B", "D", 1 / 3.0),
250: (12)                       ("C", "D", 1 / 3.0),
251: (8)                    ]
252: (8)                    Panswer = nx.Graph()
253: (8)                    Panswer.add_weighted_edges_from(edges)
254: (8)                    P = bipartite.weighted_projected_graph(self.N, "ABCDE", ratio=True)
255: (8)                    assert edges_equal(list(P.edges()), Panswer.edges())
256: (8)                    for u, v in list(P.edges()):
257: (12)                       assert P[u][v]["weight"] == Panswer[u][v]["weight"]
258: (4)                def test_project_weighted_overlap(self):
259: (8)                    edges = [
260: (12)                       ("A", "B", 2 / 2.0),
261: (12)                       ("A", "C", 1 / 1.0),
262: (12)                       ("B", "C", 1 / 1.0),
263: (12)                       ("B", "D", 1 / 1.0),
264: (12)                       ("B", "E", 2 / 3.0),
265: (12)                       ("E", "F", 1 / 1.0),
266: (8)                    ]
267: (8)                    Panswer = nx.Graph()
268: (8)                    Panswer.add_weighted_edges_from(edges)
269: (8)                    P = bipartite.overlap_weighted_projected_graph(self.G, "ABCDEF",
jaccard=False)
270: (8)                    assert edges_equal(list(P.edges()), Panswer.edges())
271: (8)                    for u, v in list(P.edges()):
272: (12)                       assert P[u][v]["weight"] == Panswer[u][v]["weight"]
273: (8)                    edges = [
274: (12)                       ("A", "B", 3 / 3.0),
275: (12)                       ("A", "E", 1 / 1.0),
276: (12)                       ("A", "C", 1 / 1.0),
277: (12)                       ("A", "D", 1 / 1.0),
278: (12)                       ("B", "E", 1 / 1.0),
279: (12)                       ("B", "C", 1 / 1.0),
280: (12)                       ("B", "D", 1 / 1.0),
281: (12)                       ("C", "D", 1 / 1.0),
282: (8)                    ]
283: (8)                    Panswer = nx.Graph()
284: (8)                    Panswer.add_weighted_edges_from(edges)
285: (8)                    P = bipartite.overlap_weighted_projected_graph(self.N, "ABCDE",
jaccard=False)
286: (8)                    assert edges_equal(list(P.edges()), Panswer.edges())
287: (8)                    for u, v in list(P.edges()):
288: (12)                       assert P[u][v]["weight"] == Panswer[u][v]["weight"]
289: (4)                def test_project_weighted_jaccard(self):
290: (8)                    edges = [
```

```
291: (12)                        ("A", "B", 2 / 5.0),
292: (12)                        ("A", "C", 1 / 2.0),
293: (12)                        ("B", "C", 1 / 5.0),
294: (12)                        ("B", "D", 1 / 5.0),
295: (12)                        ("B", "E", 2 / 6.0),
296: (12)                        ("E", "F", 1 / 3.0),
297: (8)                     ]
298: (8)                     Panswer = nx.Graph()
299: (8)                     Panswer.add_weighted_edges_from(edges)
300: (8)                     P = bipartite.overlap_weighted_projected_graph(self.G, "ABCDEF")
301: (8)                     assert edges_equal(list(P.edges()), Panswer.edges())
302: (8)                     for u, v in list(P.edges()):
303: (12)                        assert P[u][v]["weight"] == Panswer[u][v]["weight"]
304: (8)                     edges = [
305: (12)                        ("A", "B", 3 / 3.0),
306: (12)                        ("A", "E", 1 / 3.0),
307: (12)                        ("A", "C", 1 / 3.0),
308: (12)                        ("A", "D", 1 / 3.0),
309: (12)                        ("B", "E", 1 / 3.0),
310: (12)                        ("B", "C", 1 / 3.0),
311: (12)                        ("B", "D", 1 / 3.0),
312: (12)                        ("C", "D", 1 / 1.0),
313: (8)                     ]
314: (8)                     Panswer = nx.Graph()
315: (8)                     Panswer.add_weighted_edges_from(edges)
316: (8)                     P = bipartite.overlap_weighted_projected_graph(self.N, "ABCDE")
317: (8)                     assert edges_equal(list(P.edges()), Panswer.edges())
318: (8)                     for u, v in P.edges():
319: (12)                        assert P[u][v]["weight"] == Panswer[u][v]["weight"]
320: (4)                 def test_generic_weighted_projected_graph_simple(self):
321: (8)                     def shared(G, u, v):
322: (12)                        return len(set(G[u]) & set(G[v]))
323: (8)                     B = nx.path_graph(5)
324: (8)                     G = bipartite.generic_weighted_projected_graph(
325: (12)                        B, [0, 2, 4], weight_function=shared
326: (8)                     )
327: (8)                     assert nodes_equal(list(G), [0, 2, 4])
328: (8)                     assert edges_equal(
329: (12)                        list(G.edges(data=True)),
330: (12)                        [(0, 2, {"weight": 1}), (2, 4, {"weight": 1})],
331: (8)                     )
332: (8)                     G = bipartite.generic_weighted_projected_graph(B, [0, 2, 4])
333: (8)                     assert nodes_equal(list(G), [0, 2, 4])
334: (8)                     assert edges_equal(
335: (12)                        list(G.edges(data=True)),
336: (12)                        [(0, 2, {"weight": 1}), (2, 4, {"weight": 1})],
337: (8)                     )
338: (8)                     B = nx.DiGraph()
339: (8)                     nx.add_path(B, range(5))
340: (8)                     G = bipartite.generic_weighted_projected_graph(B, [0, 2, 4])
341: (8)                     assert nodes_equal(list(G), [0, 2, 4])
342: (8)                     assert edges_equal(
343: (12)                        list(G.edges(data=True)), [(0, 2, {"weight": 1}), (2, 4,
{"weight": 1})]
344: (8)                     )
345: (4)                 def test_generic_weighted_projected_graph_custom(self):
346: (8)                     def jaccard(G, u, v):
347: (12)                        unbrs = set(G[u])
348: (12)                        vnbrs = set(G[v])
349: (12)                        return len(unbrs & vnbrs) / len(unbrs | vnbrs)
350: (8)                     def my_weight(G, u, v, weight="weight"):
351: (12)                        w = 0
352: (12)                        for nbr in set(G[u]) & set(G[v]):
353: (16)                            w += G.edges[u, nbr].get(weight, 1) + G.edges[v,
nbr].get(weight, 1)
354: (12)                        return w
355: (8)                     B = nx.bipartite.complete_bipartite_graph(2, 2)
356: (8)                     for i, (u, v) in enumerate(B.edges()):
357: (12)                        B.edges[u, v]["weight"] = i + 1
```

```
358: (8)                    G = bipartite.generic_weighted_projected_graph(
359: (12)                       B, [0, 1], weight_function=jaccard
360: (8)                    )
361: (8)                    assert edges_equal(list(G.edges(data=True)), [(0, 1, {"weight":
1.0})])
362: (8)                    G = bipartite.generic_weighted_projected_graph(
363: (12)                       B, [0, 1], weight_function=my_weight
364: (8)                    )
365: (8)                    assert edges_equal(list(G.edges(data=True)), [(0, 1, {"weight": 10})])
366: (8)                    G = bipartite.generic_weighted_projected_graph(B, [0, 1])
367: (8)                    assert edges_equal(list(G.edges(data=True)), [(0, 1, {"weight": 2})])
```

File 129 - test_redundancy.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.bipartite.redundancy` module.
2: (0)              """
3: (0)              import pytest
4: (0)              from networkx import NetworkXError, cycle_graph
5: (0)              from networkx.algorithms.bipartite import complete_bipartite_graph,
node_redundancy
6: (0)              def test_no_redundant_nodes():
7: (4)                  G = complete_bipartite_graph(2, 2)
8: (4)                  rc = node_redundancy(G)
9: (4)                  assert all(redundancy == 1 for redundancy in rc.values())
10: (4)                 rc = node_redundancy(G, (2, 3))
11: (4)                 assert rc == {2: 1.0, 3: 1.0}
12: (0)             def test_redundant_nodes():
13: (4)                 G = cycle_graph(6)
14: (4)                 edge = {0, 3}
15: (4)                 G.add_edge(*edge)
16: (4)                 redundancy = node_redundancy(G)
17: (4)                 for v in edge:
18: (8)                     assert redundancy[v] == 2 / 3
19: (4)                 for v in set(G) - edge:
20: (8)                     assert redundancy[v] == 1
21: (0)             def test_not_enough_neighbors():
22: (4)                 with pytest.raises(NetworkXError):
23: (8)                     G = complete_bipartite_graph(1, 2)
24: (8)                     node_redundancy(G)
```

File 130 - test_spectral_bipartivity.py:

```
1: (0)              import pytest
2: (0)              pytest.importorskip("scipy")
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.bipartite import spectral_bipartivity as sb
5: (0)              class TestSpectralBipartivity:
6: (4)                  def test_star_like(self):
7: (8)                      G = nx.star_graph(2)
8: (8)                      G.add_edge(1, 2)
9: (8)                      assert sb(G) == pytest.approx(0.843, abs=1e-3)
10: (8)                     G = nx.star_graph(3)
11: (8)                     G.add_edge(1, 2)
12: (8)                     assert sb(G) == pytest.approx(0.871, abs=1e-3)
13: (8)                     G = nx.star_graph(4)
14: (8)                     G.add_edge(1, 2)
15: (8)                     assert sb(G) == pytest.approx(0.890, abs=1e-3)
16: (4)                 def test_k23_like(self):
17: (8)                     G = nx.complete_bipartite_graph(2, 3)
18: (8)                     G.add_edge(0, 1)
19: (8)                     assert sb(G) == pytest.approx(0.769, abs=1e-3)
20: (8)                     G = nx.complete_bipartite_graph(2, 3)
21: (8)                     G.add_edge(2, 4)
22: (8)                     assert sb(G) == pytest.approx(0.829, abs=1e-3)
23: (8)                     G = nx.complete_bipartite_graph(2, 3)
```

```
24: (8)                     G.add_edge(2, 4)
25: (8)                     G.add_edge(3, 4)
26: (8)                     assert sb(G) == pytest.approx(0.731, abs=1e-3)
27: (8)                     G = nx.complete_bipartite_graph(2, 3)
28: (8)                     G.add_edge(0, 1)
29: (8)                     G.add_edge(2, 4)
30: (8)                     assert sb(G) == pytest.approx(0.692, abs=1e-3)
31: (8)                     G = nx.complete_bipartite_graph(2, 3)
32: (8)                     G.add_edge(2, 4)
33: (8)                     G.add_edge(3, 4)
34: (8)                     G.add_edge(0, 1)
35: (8)                     assert sb(G) == pytest.approx(0.645, abs=1e-3)
36: (8)                     G = nx.complete_bipartite_graph(2, 3)
37: (8)                     G.add_edge(2, 4)
38: (8)                     G.add_edge(3, 4)
39: (8)                     G.add_edge(2, 3)
40: (8)                     assert sb(G) == pytest.approx(0.645, abs=1e-3)
41: (8)                     G = nx.complete_bipartite_graph(2, 3)
42: (8)                     G.add_edge(2, 4)
43: (8)                     G.add_edge(3, 4)
44: (8)                     G.add_edge(2, 3)
45: (8)                     G.add_edge(0, 1)
46: (8)                     assert sb(G) == pytest.approx(0.597, abs=1e-3)
47: (4)                 def test_single_nodes(self):
48: (8)                     G = nx.complete_bipartite_graph(2, 3)
49: (8)                     G.add_edge(2, 4)
50: (8)                     sbn = sb(G, nodes=[1, 2])
51: (8)                     assert sbn[1] == pytest.approx(0.85, abs=1e-2)
52: (8)                     assert sbn[2] == pytest.approx(0.77, abs=1e-2)
53: (8)                     G = nx.complete_bipartite_graph(2, 3)
54: (8)                     G.add_edge(0, 1)
55: (8)                     sbn = sb(G, nodes=[1, 2])
56: (8)                     assert sbn[1] == pytest.approx(0.73, abs=1e-2)
57: (8)                     assert sbn[2] == pytest.approx(0.82, abs=1e-2)

----------------------------------------

File 131 - __init__.py:

1: (0)              from .betweenness import *
2: (0)              from .betweenness_subset import *
3: (0)              from .closeness import *
4: (0)              from .current_flow_betweenness import *
5: (0)              from .current_flow_betweenness_subset import *
6: (0)              from .current_flow_closeness import *
7: (0)              from .degree_alg import *
8: (0)              from .dispersion import *
9: (0)              from .eigenvector import *
10: (0)             from .group import *
11: (0)             from .harmonic import *
12: (0)             from .katz import *
13: (0)             from .load import *
14: (0)             from .percolation import *
15: (0)             from .reaching import *
16: (0)             from .second_order import *
17: (0)             from .subgraph_alg import *
18: (0)             from .trophic import *
19: (0)             from .voterank_alg import *
20: (0)             from .laplacian import *

----------------------------------------

File 132 - betweenness.py:

1: (0)              """Betweenness centrality measures."""
2: (0)              from collections import deque
3: (0)              from heapq import heappop, heappush
4: (0)              from itertools import count
5: (0)              import networkx as nx
```

```
from networkx.algorithms.shortest_paths.weighted import _weight_function
from networkx.utils import py_random_state
from networkx.utils.decorators import not_implemented_for
__all__ = ["betweenness_centrality", "edge_betweenness_centrality"]
@py_random_state(5)
@nx._dispatchable(edge_attrs="weight")
def betweenness_centrality(
    G, k=None, normalized=True, weight=None, endpoints=False, seed=None
):
    r"""Compute the shortest-path betweenness centrality for nodes.
    Betweenness centrality of a node $v$ is the sum of the
    fraction of all-pairs shortest paths that pass through $v$
    .. math::
       c_B(v) =\sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}
    where $V$ is the set of nodes, $\sigma(s, t)$ is the number of
    shortest $(s, t)$-paths,  and $\sigma(s, t|v)$ is the number of
    those paths  passing through some  node $v$ other than $s, t$.
    If $s = t$, $\sigma(s, t) = 1$, and if $v \in {s, t}$,
    $\sigma(s, t|v) = 0$ [2]_.
    Parameters
    ----------
    G : graph
      A NetworkX graph.
    k : int, optional (default=None)
      If k is not None use k node samples to estimate betweenness.
      The value of k <= n where n is the number of nodes in the graph.
      Higher values give better approximation.
    normalized : bool, optional
      If True the betweenness values are normalized by `2/((n-1)(n-2))`
      for graphs, and `1/((n-1)(n-2))` for directed graphs where `n`
      is the number of nodes in G.
    weight : None or string, optional (default=None)
      If None, all edge weights are considered equal.
      Otherwise holds the name of the edge attribute used as weight.
      Weights are used to calculate weighted shortest paths, so they are
      interpreted as distances.
    endpoints : bool, optional
      If True include the endpoints in the shortest path counts.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
        Note that this is only used if k is not None.
    Returns
    -------
    nodes : dictionary
       Dictionary of nodes with betweenness centrality as the value.
    See Also
    --------
    edge_betweenness_centrality
    load_centrality
    Notes
    -----
    The algorithm is from Ulrik Brandes [1]_.
    See [4]_ for the original first published version and [2]_ for details on
    algorithms for variations and related metrics.
    For approximate betweenness calculations set k=#samples to use
    k nodes ("pivots") to estimate the betweenness values. For an estimate
    of the number of pivots needed see [3]_.
    For weighted graphs the edge weights must be greater than zero.
    Zero edge weights can produce an infinite number of equal length
    paths between pairs of nodes.
    The total number of paths between source and target is counted
    differently for directed and undirected graphs. Directed paths
    are easy to count. Undirected paths are tricky: should a path
    from "u" to "v" count as 1 undirected path or as 2 directed paths?
    For betweenness_centrality we report the number of undirected
    paths when G is undirected.
    For betweenness_centrality_subset the reporting is different.
    If the source and target subsets are the same, then we want
```

```
 75: (4)                 to count undirected paths. But if the source and target subsets
 76: (4)                 differ -- for example, if sources is {0} and targets is {1},
 77: (4)                 then we are only counting the paths in one direction. They are
 78: (4)                 undirected paths but we are counting them in a directed way.
 79: (4)                 To count them as undirected paths, each should count as half a path.
 80: (4)                 This algorithm is not guaranteed to be correct if edge weights
 81: (4)                 are floating point numbers. As a workaround you can use integer
 82: (4)                 numbers by multiplying the relevant edge attributes by a convenient
 83: (4)                 constant factor (eg 100) and converting to integers.
 84: (4)                 References
 85: (4)                 ----------
 86: (4)                 .. [1] Ulrik Brandes:
 87: (7)                    A Faster Algorithm for Betweenness Centrality.
 88: (7)                    Journal of Mathematical Sociology 25(2):163-177, 2001.
 89: (7)                    https://doi.org/10.1080/0022250X.2001.9990249
 90: (4)                 .. [2] Ulrik Brandes:
 91: (7)                    On Variants of Shortest-Path Betweenness
 92: (7)                    Centrality and their Generic Computation.
 93: (7)                    Social Networks 30(2):136-145, 2008.
 94: (7)                    https://doi.org/10.1016/j.socnet.2007.11.001
 95: (4)                 .. [3] Ulrik Brandes and Christian Pich:
 96: (7)                    Centrality Estimation in Large Networks.
 97: (7)                    International Journal of Bifurcation and Chaos 17(7):2303-2318, 2007.
 98: (7)                    https://dx.doi.org/10.1142/S0218127407018403
 99: (4)                 .. [4] Linton C. Freeman:
100: (7)                    A set of measures of centrality based on betweenness.
101: (7)                    Sociometry 40: 35–41, 1977
102: (7)                    https://doi.org/10.2307/3033543
103: (4)                 """
104: (4)                 betweenness = dict.fromkeys(G, 0.0)  # b[v]=0 for v in G
105: (4)                 if k is None:
106: (8)                     nodes = G
107: (4)                 else:
108: (8)                     nodes = seed.sample(list(G.nodes()), k)
109: (4)                 for s in nodes:
110: (8)                     if weight is None:  # use BFS
111: (12)                        S, P, sigma, _ = _single_source_shortest_path_basic(G, s)
112: (8)                     else:  # use Dijkstra's algorithm
113: (12)                        S, P, sigma, _ = _single_source_dijkstra_path_basic(G, s, weight)
114: (8)                     if endpoints:
115: (12)                        betweenness, _ = _accumulate_endpoints(betweenness, S, P, sigma,
s)
116: (8)                     else:
117: (12)                        betweenness, _ = _accumulate_basic(betweenness, S, P, sigma, s)
118: (4)                 betweenness = _rescale(
119: (8)                     betweenness,
120: (8)                     len(G),
121: (8)                     normalized=normalized,
122: (8)                     directed=G.is_directed(),
123: (8)                     k=k,
124: (8)                     endpoints=endpoints,
125: (4)                 )
126: (4)                 return betweenness
127: (0)             @py_random_state(4)
128: (0)             @nx._dispatchable(edge_attrs="weight")
129: (0)             def edge_betweenness_centrality(G, k=None, normalized=True, weight=None,
seed=None):
130: (4)                 r"""Compute betweenness centrality for edges.
131: (4)                 Betweenness centrality of an edge $e$ is the sum of the
132: (4)                 fraction of all-pairs shortest paths that pass through $e$
133: (4)                 .. math::
134: (7)                    c_B(e) =\sum_{s,t \in V} \frac{\sigma(s, t|e)}{\sigma(s, t)}
135: (4)                 where $V$ is the set of nodes, $\sigma(s, t)$ is the number of
136: (4)                 shortest $(s, t)$-paths, and $\sigma(s, t|e)$ is the number of
137: (4)                 those paths passing through edge $e$ [2]_.
138: (4)                 Parameters
139: (4)                 ----------
140: (4)                 G : graph
141: (6)                   A NetworkX graph.
```

```
(4)                 k : int, optional (default=None)
(6)                   If k is not None use k node samples to estimate betweenness.
(6)                   The value of k <= n where n is the number of nodes in the graph.
(6)                   Higher values give better approximation.
(4)                 normalized : bool, optional
(6)                   If True the betweenness values are normalized by $2/(n(n-1))$
(6)                   for graphs, and $1/(n(n-1))$ for directed graphs where $n$
(6)                   is the number of nodes in G.
(4)                 weight : None or string, optional (default=None)
(6)                   If None, all edge weights are considered equal.
(6)                   Otherwise holds the name of the edge attribute used as weight.
(6)                   Weights are used to calculate weighted shortest paths, so they are
(6)                   interpreted as distances.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(8)                     Note that this is only used if k is not None.
(4)                 Returns
(4)                 -------
(4)                 edges : dictionary
(7)                    Dictionary of edges with betweenness centrality as the value.
(4)                 See Also
(4)                 --------
(4)                 betweenness_centrality
(4)                 edge_load
(4)                 Notes
(4)                 -----
(4)                 The algorithm is from Ulrik Brandes [1]_.
(4)                 For weighted graphs the edge weights must be greater than zero.
(4)                 Zero edge weights can produce an infinite number of equal length
(4)                 paths between pairs of nodes.
(4)                 References
(4)                 ----------
(4)                 .. [1]  A Faster Algorithm for Betweenness Centrality. Ulrik Brandes,
(7)                    Journal of Mathematical Sociology 25(2):163-177, 2001.
(7)                    https://doi.org/10.1080/0022250X.2001.9990249
(4)                 .. [2] Ulrik Brandes: On Variants of Shortest-Path Betweenness
(7)                    Centrality and their Generic Computation.
(7)                    Social Networks 30(2):136-145, 2008.
(7)                    https://doi.org/10.1016/j.socnet.2007.11.001
(4)                 """
(4)                 betweenness = dict.fromkeys(G, 0.0)  # b[v]=0 for v in G
(4)                 betweenness.update(dict.fromkeys(G.edges(), 0.0))
(4)                 if k is None:
(8)                     nodes = G
(4)                 else:
(8)                     nodes = seed.sample(list(G.nodes()), k)
(4)                 for s in nodes:
(8)                     if weight is None:  # use BFS
(12)                        S, P, sigma, _ = _single_source_shortest_path_basic(G, s)
(8)                     else:  # use Dijkstra's algorithm
(12)                        S, P, sigma, _ = _single_source_dijkstra_path_basic(G, s, weight)
(8)                     betweenness = _accumulate_edges(betweenness, S, P, sigma, s)
(4)                 for n in G:  # remove nodes to only return edges
(8)                     del betweenness[n]
(4)                 betweenness = _rescale_e(
(8)                     betweenness, len(G), normalized=normalized, directed=G.is_directed()
(4)                 )
(4)                 if G.is_multigraph():
(8)                     betweenness = _add_edge_keys(G, betweenness, weight=weight)
(4)                 return betweenness
(0)             def _single_source_shortest_path_basic(G, s):
(4)                 S = []
(4)                 P = {}
(4)                 for v in G:
(8)                     P[v] = []
(4)                 sigma = dict.fromkeys(G, 0.0)  # sigma[v]=0 for v in G
(4)                 D = {}
(4)                 sigma[s] = 1.0
```

```
(4)                 D[s] = 0
(4)                 Q = deque([s])
(4)                 while Q:  # use BFS to find shortest paths
(8)                     v = Q.popleft()
(8)                     S.append(v)
(8)                     Dv = D[v]
(8)                     sigmav = sigma[v]
(8)                     for w in G[v]:
(12)                        if w not in D:
(16)                            Q.append(w)
(16)                            D[w] = Dv + 1
(12)                        if D[w] == Dv + 1:  # this is a shortest path, count paths
(16)                            sigma[w] += sigmav
(16)                            P[w].append(v)  # predecessors
(4)                 return S, P, sigma, D
(0)             def _single_source_dijkstra_path_basic(G, s, weight):
(4)                 weight = _weight_function(G, weight)
(4)                 S = []
(4)                 P = {}
(4)                 for v in G:
(8)                     P[v] = []
(4)                 sigma = dict.fromkeys(G, 0.0)  # sigma[v]=0 for v in G
(4)                 D = {}
(4)                 sigma[s] = 1.0
(4)                 push = heappush
(4)                 pop = heappop
(4)                 seen = {s: 0}
(4)                 c = count()
(4)                 Q = []  # use Q as heap with (distance,node id) tuples
(4)                 push(Q, (0, next(c), s, s))
(4)                 while Q:
(8)                     (dist, _, pred, v) = pop(Q)
(8)                     if v in D:
(12)                        continue  # already searched this node.
(8)                     sigma[v] += sigma[pred]  # count paths
(8)                     S.append(v)
(8)                     D[v] = dist
(8)                     for w, edgedata in G[v].items():
(12)                        vw_dist = dist + weight(v, w, edgedata)
(12)                        if w not in D and (w not in seen or vw_dist < seen[w]):
(16)                            seen[w] = vw_dist
(16)                            push(Q, (vw_dist, next(c), v, w))
(16)                            sigma[w] = 0.0
(16)                            P[w] = [v]
(12)                        elif vw_dist == seen[w]:  # handle equal paths
(16)                            sigma[w] += sigma[v]
(16)                            P[w].append(v)
(4)                 return S, P, sigma, D
(0)             def _accumulate_basic(betweenness, S, P, sigma, s):
(4)                 delta = dict.fromkeys(S, 0)
(4)                 while S:
(8)                     w = S.pop()
(8)                     coeff = (1 + delta[w]) / sigma[w]
(8)                     for v in P[w]:
(12)                        delta[v] += sigma[v] * coeff
(8)                     if w != s:
(12)                        betweenness[w] += delta[w]
(4)                 return betweenness, delta
(0)             def _accumulate_endpoints(betweenness, S, P, sigma, s):
(4)                 betweenness[s] += len(S) - 1
(4)                 delta = dict.fromkeys(S, 0)
(4)                 while S:
(8)                     w = S.pop()
(8)                     coeff = (1 + delta[w]) / sigma[w]
(8)                     for v in P[w]:
(12)                        delta[v] += sigma[v] * coeff
(8)                     if w != s:
(12)                        betweenness[w] += delta[w] + 1
(4)                 return betweenness, delta
```

```
(0)             def _accumulate_edges(betweenness, S, P, sigma, s):
(4)                 delta = dict.fromkeys(S, 0)
(4)                 while S:
(8)                     w = S.pop()
(8)                     coeff = (1 + delta[w]) / sigma[w]
(8)                     for v in P[w]:
(12)                        c = sigma[v] * coeff
(12)                        if (v, w) not in betweenness:
(16)                            betweenness[(w, v)] += c
(12)                        else:
(16)                            betweenness[(v, w)] += c
(12)                        delta[v] += c
(8)                     if w != s:
(12)                        betweenness[w] += delta[w]
(4)                 return betweenness
(0)             def _rescale(betweenness, n, normalized, directed=False, k=None,
endpoints=False):
(4)                 if normalized:
(8)                     if endpoints:
(12)                        if n < 2:
(16)                            scale = None  # no normalization
(12)                        else:
(16)                            scale = 1 / (n * (n - 1))
(8)                     elif n <= 2:
(12)                        scale = None  # no normalization b=0 for all nodes
(8)                     else:
(12)                        scale = 1 / ((n - 1) * (n - 2))
(4)                 else:  # rescale by 2 for undirected graphs
(8)                     if not directed:
(12)                        scale = 0.5
(8)                     else:
(12)                        scale = None
(4)                 if scale is not None:
(8)                     if k is not None:
(12)                        scale = scale * n / k
(8)                     for v in betweenness:
(12)                        betweenness[v] *= scale
(4)                 return betweenness
(0)             def _rescale_e(betweenness, n, normalized, directed=False, k=None):
(4)                 if normalized:
(8)                     if n <= 1:
(12)                        scale = None  # no normalization b=0 for all nodes
(8)                     else:
(12)                        scale = 1 / (n * (n - 1))
(4)                 else:  # rescale by 2 for undirected graphs
(8)                     if not directed:
(12)                        scale = 0.5
(8)                     else:
(12)                        scale = None
(4)                 if scale is not None:
(8)                     if k is not None:
(12)                        scale = scale * n / k
(8)                     for v in betweenness:
(12)                        betweenness[v] *= scale
(4)                 return betweenness
(0)             @not_implemented_for("graph")
(0)             def _add_edge_keys(G, betweenness, weight=None):
(4)                 r"""Adds the corrected betweenness centrality (BC) values for multigraphs.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph.
(4)                 betweenness : dictionary
(8)                     Dictionary mapping adjacent node tuples to betweenness centrality
values.
(4)                 weight : string or function
(8)                     See `_weight_function` for details. Defaults to `None`.
(4)                 Returns
(4)                 -------
(4)                 edges : dictionary
```

```
347: (8)                    The parameter `betweenness` including edges with keys and their
348: (8)                    betweenness centrality values.
349: (4)                The BC value is divided among edges of equal weight.
350: (4)                """
351: (4)                _weight = _weight_function(G, weight)
352: (4)                edge_bc = dict.fromkeys(G.edges, 0.0)
353: (4)                for u, v in betweenness:
354: (8)                    d = G[u][v]
355: (8)                    wt = _weight(u, v, d)
356: (8)                    keys = [k for k in d if _weight(u, v, {k: d[k]}) == wt]
357: (8)                    bc = betweenness[(u, v)] / len(keys)
358: (8)                    for k in keys:
359: (12)                       edge_bc[(u, v, k)] = bc
360: (4)                return edge_bc
```

File 133 - betweenness_subset.py:

```
1: (0)              """Betweenness centrality measures for subsets of nodes."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.centrality.betweenness import (
4: (4)                  _add_edge_keys,
5: (0)              )
6: (0)              from networkx.algorithms.centrality.betweenness import (
7: (4)                  _single_source_dijkstra_path_basic as dijkstra,
8: (0)              )
9: (0)              from networkx.algorithms.centrality.betweenness import (
10: (4)                 _single_source_shortest_path_basic as shortest_path,
11: (0)             )
12: (0)             __all__ = [
13: (4)                 "betweenness_centrality_subset",
14: (4)                 "edge_betweenness_centrality_subset",
15: (0)             ]
16: (0)             @nx._dispatchable(edge_attrs="weight")
17: (0)             def betweenness_centrality_subset(G, sources, targets, normalized=False,
weight=None):
18: (4)                 r"""Compute betweenness centrality for a subset of nodes.
19: (4)                 .. math::
20: (7)                    c_B(v) =\sum_{s\in S, t \in T} \frac{\sigma(s, t|v)}{\sigma(s, t)}
21: (4)                 where $S$ is the set of sources, $T$ is the set of targets,
22: (4)                 $\sigma(s, t)$ is the number of shortest $(s, t)$-paths,
23: (4)                 and $\sigma(s, t|v)$ is the number of those paths
24: (4)                 passing through some  node $v$ other than $s, t$.
25: (4)                 If $s = t$, $\sigma(s, t) = 1$,
26: (4)                 and if $v \in {s, t}$, $\sigma(s, t|v) = 0$ [2]_.
27: (4)                 Parameters
28: (4)                 ----------
29: (4)                 G : graph
30: (6)                   A NetworkX graph.
31: (4)                 sources: list of nodes
32: (6)                   Nodes to use as sources for shortest paths in betweenness
33: (4)                 targets: list of nodes
34: (6)                   Nodes to use as targets for shortest paths in betweenness
35: (4)                 normalized : bool, optional
36: (6)                   If True the betweenness values are normalized by $2/((n-1)(n-2))$
37: (6)                   for graphs, and $1/((n-1)(n-2))$ for directed graphs where $n$
38: (6)                   is the number of nodes in G.
39: (4)                 weight : None or string, optional (default=None)
40: (6)                   If None, all edge weights are considered equal.
41: (6)                   Otherwise holds the name of the edge attribute used as weight.
42: (6)                   Weights are used to calculate weighted shortest paths, so they are
43: (6)                   interpreted as distances.
44: (4)                 Returns
45: (4)                 -------
46: (4)                 nodes : dictionary
47: (7)                    Dictionary of nodes with betweenness centrality as the value.
48: (4)                 See Also
49: (4)                 --------
```

```
(4)                 edge_betweenness_centrality
(4)                 load_centrality
(4)                 Notes
(4)                 -----
(4)                 The basic algorithm is from [1]_.
(4)                 For weighted graphs the edge weights must be greater than zero.
(4)                 Zero edge weights can produce an infinite number of equal length
(4)                 paths between pairs of nodes.
(4)                 The normalization might seem a little strange but it is
(4)                 designed to make betweenness_centrality(G) be the same as
(4)                 betweenness_centrality_subset(G,sources=G.nodes(),targets=G.nodes()).
(4)                 The total number of paths between source and target is counted
(4)                 differently for directed and undirected graphs. Directed paths
(4)                 are easy to count. Undirected paths are tricky: should a path
(4)                 from "u" to "v" count as 1 undirected path or as 2 directed paths?
(4)                 For betweenness_centrality we report the number of undirected
(4)                 paths when G is undirected.
(4)                 For betweenness_centrality_subset the reporting is different.
(4)                 If the source and target subsets are the same, then we want
(4)                 to count undirected paths. But if the source and target subsets
(4)                 differ -- for example, if sources is {0} and targets is {1},
(4)                 then we are only counting the paths in one direction. They are
(4)                 undirected paths but we are counting them in a directed way.
(4)                 To count them as undirected paths, each should count as half a path.
(4)                 References
(4)                 ----------
(4)                 .. [1] Ulrik Brandes, A Faster Algorithm for Betweenness Centrality.
(7)                    Journal of Mathematical Sociology 25(2):163-177, 2001.
(7)                    https://doi.org/10.1080/0022250X.2001.9990249
(4)                 .. [2] Ulrik Brandes: On Variants of Shortest-Path Betweenness
(7)                    Centrality and their Generic Computation.
(7)                    Social Networks 30(2):136-145, 2008.
(7)                    https://doi.org/10.1016/j.socnet.2007.11.001
(4)                 """
(4)                 b = dict.fromkeys(G, 0.0)  # b[v]=0 for v in G
(4)                 for s in sources:
(8)                     if weight is None:  # use BFS
(12)                        S, P, sigma, _ = shortest_path(G, s)
(8)                     else:  # use Dijkstra's algorithm
(12)                        S, P, sigma, _ = dijkstra(G, s, weight)
(8)                     b = _accumulate_subset(b, S, P, sigma, s, targets)
(4)                 b = _rescale(b, len(G), normalized=normalized, directed=G.is_directed())
(4)                 return b
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def edge_betweenness_centrality_subset(
(4)                 G, sources, targets, normalized=False, weight=None
(0)             ):
(4)                 r"""Compute betweenness centrality for edges for a subset of nodes.
(4)                 .. math::
(7)                    c_B(v) =\sum_{s\in S,t \in T} \frac{\sigma(s, t|e)}{\sigma(s, t)}
(4)                 where $S$ is the set of sources, $T$ is the set of targets,
(4)                 $\sigma(s, t)$ is the number of shortest $(s, t)$-paths,
(4)                 and $\sigma(s, t|e)$ is the number of those paths
(4)                 passing through edge $e$ [2]_.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A networkx graph.
(4)                 sources: list of nodes
(6)                   Nodes to use as sources for shortest paths in betweenness
(4)                 targets: list of nodes
(6)                   Nodes to use as targets for shortest paths in betweenness
(4)                 normalized : bool, optional
(6)                   If True the betweenness values are normalized by `2/(n(n-1))`
(6)                   for graphs, and `1/(n(n-1))` for directed graphs where `n`
(6)                   is the number of nodes in G.
(4)                 weight : None or string, optional (default=None)
(6)                   If None, all edge weights are considered equal.
(6)                   Otherwise holds the name of the edge attribute used as weight.
```

```
(6)                 Weights are used to calculate weighted shortest paths, so they are
(6)                 interpreted as distances.
(4)             Returns
(4)             -------
(4)             edges : dictionary
(7)                Dictionary of edges with Betweenness centrality as the value.
(4)             See Also
(4)             --------
(4)             betweenness_centrality
(4)             edge_load
(4)             Notes
(4)             -----
(4)             The basic algorithm is from [1]_.
(4)             For weighted graphs the edge weights must be greater than zero.
(4)             Zero edge weights can produce an infinite number of equal length
(4)             paths between pairs of nodes.
(4)             The normalization might seem a little strange but it is the same
(4)             as in edge_betweenness_centrality() and is designed to make
(4)             edge_betweenness_centrality(G) be the same as
(4)             edge_betweenness_centrality_subset(G,sources=G.nodes(),targets=G.nodes()).
(4)             References
(4)             ----------
(4)             .. [1] Ulrik Brandes, A Faster Algorithm for Betweenness Centrality.
(7)                Journal of Mathematical Sociology 25(2):163-177, 2001.
(7)                https://doi.org/10.1080/0022250X.2001.9990249
(4)             .. [2] Ulrik Brandes: On Variants of Shortest-Path Betweenness
(7)                Centrality and their Generic Computation.
(7)                Social Networks 30(2):136-145, 2008.
(7)                https://doi.org/10.1016/j.socnet.2007.11.001
(4)             """
(4)             b = dict.fromkeys(G, 0.0)  # b[v]=0 for v in G
(4)             b.update(dict.fromkeys(G.edges(), 0.0))  # b[e] for e in G.edges()
(4)             for s in sources:
(8)                 if weight is None:  # use BFS
(12)                    S, P, sigma, _ = shortest_path(G, s)
(8)                 else:  # use Dijkstra's algorithm
(12)                    S, P, sigma, _ = dijkstra(G, s, weight)
(8)                 b = _accumulate_edges_subset(b, S, P, sigma, s, targets)
(4)             for n in G:  # remove nodes to only return edges
(8)                 del b[n]
(4)             b = _rescale_e(b, len(G), normalized=normalized, directed=G.is_directed())
(4)             if G.is_multigraph():
(8)                 b = _add_edge_keys(G, b, weight=weight)
(4)             return b
(0)         def _accumulate_subset(betweenness, S, P, sigma, s, targets):
(4)             delta = dict.fromkeys(S, 0.0)
(4)             target_set = set(targets) - {s}
(4)             while S:
(8)                 w = S.pop()
(8)                 if w in target_set:
(12)                    coeff = (delta[w] + 1.0) / sigma[w]
(8)                 else:
(12)                    coeff = delta[w] / sigma[w]
(8)                 for v in P[w]:
(12)                    delta[v] += sigma[v] * coeff
(8)                 if w != s:
(12)                    betweenness[w] += delta[w]
(4)             return betweenness
(0)         def _accumulate_edges_subset(betweenness, S, P, sigma, s, targets):
(4)             """edge_betweenness_centrality_subset helper."""
(4)             delta = dict.fromkeys(S, 0)
(4)             target_set = set(targets)
(4)             while S:
(8)                 w = S.pop()
(8)                 for v in P[w]:
(12)                    if w in target_set:
(16)                        c = (sigma[v] / sigma[w]) * (1.0 + delta[w])
(12)                    else:
(16)                        c = delta[w] / len(P[w])
```

```
188: (12)                        if (v, w) not in betweenness:
189: (16)                            betweenness[(w, v)] += c
190: (12)                        else:
191: (16)                            betweenness[(v, w)] += c
192: (12)                        delta[v] += c
193: (8)                    if w != s:
194: (12)                        betweenness[w] += delta[w]
195: (4)                return betweenness
196: (0)            def _rescale(betweenness, n, normalized, directed=False):
197: (4)                """betweenness_centrality_subset helper."""
198: (4)                if normalized:
199: (8)                    if n <= 2:
200: (12)                       scale = None  # no normalization b=0 for all nodes
201: (8)                    else:
202: (12)                       scale = 1.0 / ((n - 1) * (n - 2))
203: (4)                else:  # rescale by 2 for undirected graphs
204: (8)                    if not directed:
205: (12)                       scale = 0.5
206: (8)                    else:
207: (12)                       scale = None
208: (4)                if scale is not None:
209: (8)                    for v in betweenness:
210: (12)                       betweenness[v] *= scale
211: (4)                return betweenness
212: (0)            def _rescale_e(betweenness, n, normalized, directed=False):
213: (4)                """edge_betweenness_centrality_subset helper."""
214: (4)                if normalized:
215: (8)                    if n <= 1:
216: (12)                       scale = None  # no normalization b=0 for all nodes
217: (8)                    else:
218: (12)                       scale = 1.0 / (n * (n - 1))
219: (4)                else:  # rescale by 2 for undirected graphs
220: (8)                    if not directed:
221: (12)                       scale = 0.5
222: (8)                    else:
223: (12)                       scale = None
224: (4)                if scale is not None:
225: (8)                    for v in betweenness:
226: (12)                       betweenness[v] *= scale
227: (4)                return betweenness
```

File 134 - closeness.py:

```
1: (0)              """
2: (0)              Closeness centrality measures.
3: (0)              """
4: (0)              import functools
5: (0)              import networkx as nx
6: (0)              from networkx.exception import NetworkXError
7: (0)              from networkx.utils.decorators import not_implemented_for
8: (0)              __all__ = ["closeness_centrality", "incremental_closeness_centrality"]
9: (0)              @nx._dispatchable(edge_attrs="distance")
10: (0)             def closeness_centrality(G, u=None, distance=None, wf_improved=True):
11: (4)                 r"""Compute closeness centrality for nodes.
12: (4)                 Closeness centrality [1]_ of a node `u` is the reciprocal of the
13: (4)                 average shortest path distance to `u` over all `n-1` reachable nodes.
14: (4)                 .. math::
15: (8)                     C(u) = \frac{n - 1}{\sum_{v=1}^{n-1} d(v, u)},
16: (4)                 where `d(v, u)` is the shortest-path distance between `v` and `u`,
17: (4)                 and `n-1` is the number of nodes reachable from `u`. Notice that the
18: (4)                 closeness distance function computes the incoming distance to `u`
19: (4)                 for directed graphs. To use outward distance, act on `G.reverse()`.
20: (4)                 Notice that higher values of closeness indicate higher centrality.
21: (4)                 Wasserman and Faust propose an improved formula for graphs with
22: (4)                 more than one connected component. The result is "a ratio of the
23: (4)                 fraction of actors in the group who are reachable, to the average
24: (4)                 distance" from the reachable actors [2]_. You might think this
```

```
(4)                 scale factor is inverted but it is not. As is, nodes from small
(4)                 components receive a smaller closeness value. Letting `N` denote
(4)                 the number of nodes in the graph,
(4)                 .. math::
(8)                     C_{WF}(u) = \frac{n-1}{N-1} \frac{n - 1}{\sum_{v=1}^{n-1} d(v, u)},
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A NetworkX graph
(4)                 u : node, optional
(6)                   Return only the value for node u
(4)                 distance : edge attribute key, optional (default=None)
(6)                   Use the specified edge attribute as the edge distance in shortest
(6)                   path calculations.  If `None` (the default) all edges have a distance of
1.
(6)                   Absent edge attributes are assigned a distance of 1. Note that no check
(6)                   is performed to ensure that edges have the provided attribute.
(4)                 wf_improved : bool, optional (default=True)
(6)                   If True, scale by the fraction of nodes reachable. This gives the
(6)                   Wasserman and Faust improved formula. For single component graphs
(6)                   it is the same as the original formula.
(4)                 Returns
(4)                 -------
(4)                 nodes : dictionary
(6)                   Dictionary of nodes with closeness centrality as the value.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
(4)                 >>> nx.closeness_centrality(G)
(4)                 {0: 1.0, 1: 1.0, 2: 0.75, 3: 0.75}
(4)                 See Also
(4)                 --------
(4)                 betweenness_centrality, load_centrality, eigenvector_centrality,
(4)                 degree_centrality, incremental_closeness_centrality
(4)                 Notes
(4)                 -----
(4)                 The closeness centrality is normalized to `(n-1)/(|G|-1)` where
(4)                 `n` is the number of nodes in the connected part of graph
(4)                 containing the node.  If the graph is not completely connected,
(4)                 this algorithm computes the closeness centrality for each
(4)                 connected part separately scaled by that parts size.
(4)                 If the 'distance' keyword is set to an edge attribute key then the
(4)                 shortest-path length will be computed using Dijkstra's algorithm with
(4)                 that edge attribute as the edge weight.
(4)                 The closeness centrality uses *inward* distance to a node, not outward.
(4)                 If you want to use outword distances apply the function to `G.reverse()`
(4)                 In NetworkX 2.2 and earlier a bug caused Dijkstra's algorithm to use the
(4)                 outward distance rather than the inward distance. If you use a 'distance'
(4)                 keyword and a DiGraph, your results will change between v2.2 and v2.3.
(4)                 References
(4)                 ----------
(4)                 .. [1] Linton C. Freeman: Centrality in networks: I.
(7)                    Conceptual clarification. Social Networks 1:215-239, 1979.
(7)                    https://doi.org/10.1016/0378-8733(78)90021-7
(4)                 .. [2] pg. 201 of Wasserman, S. and Faust, K.,
(7)                    Social Network Analysis: Methods and Applications, 1994,
(7)                    Cambridge University Press.
(4)                 """
(4)                 if G.is_directed():
(8)                     G = G.reverse()  # create a reversed graph view
(4)                 if distance is not None:
(8)                     path_length = functools.partial(
(12)                        nx.single_source_dijkstra_path_length, weight=distance
(8)                     )
(4)                 else:
(8)                     path_length = nx.single_source_shortest_path_length
(4)                 if u is None:
(8)                     nodes = G.nodes
(4)                 else:
```

```
93: (8)                     nodes = [u]
94: (4)                 closeness_dict = {}
95: (4)                 for n in nodes:
96: (8)                     sp = path_length(G, n)
97: (8)                     totsp = sum(sp.values())
98: (8)                     len_G = len(G)
99: (8)                     _closeness_centrality = 0.0
100: (8)                    if totsp > 0.0 and len_G > 1:
101: (12)                       _closeness_centrality = (len(sp) - 1.0) / totsp
102: (12)                       if wf_improved:
103: (16)                           s = (len(sp) - 1.0) / (len_G - 1)
104: (16)                           _closeness_centrality *= s
105: (8)                    closeness_dict[n] = _closeness_centrality
106: (4)                if u is not None:
107: (8)                    return closeness_dict[u]
108: (4)                return closeness_dict
109: (0)            @not_implemented_for("directed")
110: (0)            @nx._dispatchable(mutates_input=True)
111: (0)            def incremental_closeness_centrality(
112: (4)                G, edge, prev_cc=None, insertion=True, wf_improved=True
113: (0)            ):
114: (4)                r"""Incremental closeness centrality for nodes.
115: (4)                Compute closeness centrality for nodes using level-based work filtering
116: (4)                as described in Incremental Algorithms for Closeness Centrality by
Sariyuce et al.
117: (4)                Level-based work filtering detects unnecessary updates to the closeness
118: (4)                centrality and filters them out.
119: (4)                ---
120: (4)                From "Incremental Algorithms for Closeness Centrality":
121: (4)                Theorem 1: Let :math:`G = (V, E)` be a graph and u and v be two vertices
in V
122: (4)                such that there is no edge (u, v) in E. Let :math:`G' = (V, E \cup uv)`
123: (4)                Then :math:`cc[s] = cc'[s]` if and only if :math:`\left|dG(s, u) - dG(s,
v)\right| \leq 1`.
124: (4)                Where :math:`dG(u, v)` denotes the length of the shortest path between
125: (4)                two vertices u, v in a graph G, cc[s] is the closeness centrality for a
126: (4)                vertex s in V, and cc'[s] is the closeness centrality for a
127: (4)                vertex s in V, with the (u, v) edge added.
128: (4)                ---
129: (4)                We use Theorem 1 to filter out updates when adding or removing an edge.
130: (4)                When adding an edge (u, v), we compute the shortest path lengths from all
131: (4)                other nodes to u and to v before the node is added. When removing an edge,
132: (4)                we compute the shortest path lengths after the edge is removed. Then we
133: (4)                apply Theorem 1 to use previously computed closeness centrality for nodes
134: (4)                where :math:`\left|dG(s, u) - dG(s, v)\right| \leq 1`. This works only for
135: (4)                undirected, unweighted graphs; the distance argument is not supported.
136: (4)                Closeness centrality [1]_ of a node `u` is the reciprocal of the
137: (4)                sum of the shortest path distances from `u` to all `n-1` other nodes.
138: (4)                Since the sum of distances depends on the number of nodes in the
139: (4)                graph, closeness is normalized by the sum of minimum possible
140: (4)                distances `n-1`.
141: (4)                .. math::
142: (8)                    C(u) = \frac{n - 1}{\sum_{v=1}^{n-1} d(v, u)},
143: (4)                where `d(v, u)` is the shortest-path distance between `v` and `u`,
144: (4)                and `n` is the number of nodes in the graph.
145: (4)                Notice that higher values of closeness indicate higher centrality.
146: (4)                Parameters
147: (4)                ----------
148: (4)                G : graph
149: (6)                  A NetworkX graph
150: (4)                edge : tuple
151: (6)                  The modified edge (u, v) in the graph.
152: (4)                prev_cc : dictionary
153: (6)                  The previous closeness centrality for all nodes in the graph.
154: (4)                insertion : bool, optional
155: (6)                  If True (default) the edge was inserted, otherwise it was deleted from
the graph.
156: (4)                wf_improved : bool, optional (default=True)
157: (6)                  If True, scale by the fraction of nodes reachable. This gives the
```

```
158: (6)                  Wasserman and Faust improved formula. For single component graphs
159: (6)                  it is the same as the original formula.
160: (4)                Returns
161: (4)                -------
162: (4)                nodes : dictionary
163: (6)                  Dictionary of nodes with closeness centrality as the value.
164: (4)                See Also
165: (4)                --------
166: (4)                betweenness_centrality, load_centrality, eigenvector_centrality,
167: (4)                degree_centrality, closeness_centrality
168: (4)                Notes
169: (4)                -----
170: (4)                The closeness centrality is normalized to `(n-1)/(|G|-1)` where
171: (4)                `n` is the number of nodes in the connected part of graph
172: (4)                containing the node.  If the graph is not completely connected,
173: (4)                this algorithm computes the closeness centrality for each
174: (4)                connected part separately.
175: (4)                References
176: (4)                ----------
177: (4)                .. [1] Freeman, L.C., 1979. Centrality in networks: I.
178: (7)                   Conceptual clarification.  Social Networks 1, 215--239.
179: (7)                   https://doi.org/10.1016/0378-8733(78)90021-7
180: (4)                .. [2] Sariyuce, A.E. ; Kaya, K. ; Saule, E. ; Catalyiirek, U.V.
Incremental
181: (7)                   Algorithms for Closeness Centrality. 2013 IEEE International Conference
on Big Data
182: (7)                   http://sariyuce.com/papers/bigdata13.pdf
183: (4)                """
184: (4)                if prev_cc is not None and set(prev_cc.keys()) != set(G.nodes()):
185: (8)                    raise NetworkXError("prev_cc and G do not have the same nodes")
186: (4)                (u, v) = edge
187: (4)                path_length = nx.single_source_shortest_path_length
188: (4)                if insertion:
189: (8)                    du = path_length(G, u)
190: (8)                    dv = path_length(G, v)
191: (8)                    G.add_edge(u, v)
192: (4)                else:
193: (8)                    G.remove_edge(u, v)
194: (8)                    du = path_length(G, u)
195: (8)                    dv = path_length(G, v)
196: (4)                if prev_cc is None:
197: (8)                    return nx.closeness_centrality(G)
198: (4)                nodes = G.nodes()
199: (4)                closeness_dict = {}
200: (4)                for n in nodes:
201: (8)                    if n in du and n in dv and abs(du[n] - dv[n]) <= 1:
202: (12)                       closeness_dict[n] = prev_cc[n]
203: (8)                    else:
204: (12)                       sp = path_length(G, n)
205: (12)                       totsp = sum(sp.values())
206: (12)                       len_G = len(G)
207: (12)                       _closeness_centrality = 0.0
208: (12)                       if totsp > 0.0 and len_G > 1:
209: (16)                           _closeness_centrality = (len(sp) - 1.0) / totsp
210: (16)                           if wf_improved:
211: (20)                               s = (len(sp) - 1.0) / (len_G - 1)
212: (20)                               _closeness_centrality *= s
213: (12)                       closeness_dict[n] = _closeness_centrality
214: (4)                if insertion:
215: (8)                    G.remove_edge(u, v)
216: (4)                else:
217: (8)                    G.add_edge(u, v)
218: (4)                return closeness_dict

----------------------------------------

File 135 - current_flow_betweenness.py:

1: (0)              """Current-flow betweenness centrality measures."""
```

```
import networkx as nx
from networkx.algorithms.centrality.flow_matrix import (
    CGInverseLaplacian,
    FullInverseLaplacian,
    SuperLUInverseLaplacian,
    flow_matrix_row,
)
from networkx.utils import (
    not_implemented_for,
    py_random_state,
    reverse_cuthill_mckee_ordering,
)
__all__ = [
    "current_flow_betweenness_centrality",
    "approximate_current_flow_betweenness_centrality",
    "edge_current_flow_betweenness_centrality",
]
@not_implemented_for("directed")
@py_random_state(7)
@nx._dispatchable(edge_attrs="weight")
def approximate_current_flow_betweenness_centrality(
    G,
    normalized=True,
    weight=None,
    dtype=float,
    solver="full",
    epsilon=0.5,
    kmax=10000,
    seed=None,
):
    r"""Compute the approximate current-flow betweenness centrality for nodes.
    Approximates the current-flow betweenness centrality within absolute
    error of epsilon with high probability [1]_.
    Parameters
    ----------
    G : graph
      A NetworkX graph
    normalized : bool, optional (default=True)
      If True the betweenness values are normalized by 2/[(n-1)(n-2)] where
      n is the number of nodes in G.
    weight : string or None, optional (default=None)
      Key for edge data used as the edge weight.
      If None, then use 1 as each edge weight.
      The weight reflects the capacity or the strength of the
      edge.
    dtype : data type (float)
      Default data type for internal matrices.
      Set to np.float32 for lower memory consumption.
    solver : string (default='full')
       Type of linear solver to use for computing the flow matrix.
       Options are "full" (uses most memory), "lu" (recommended), and
       "cg" (uses least memory).
    epsilon: float
        Absolute error tolerance.
    kmax: int
       Maximum number of sample node pairs to use for approximation.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    Returns
    -------
    nodes : dictionary
       Dictionary of nodes with betweenness centrality as the value.
    See Also
    --------
    current_flow_betweenness_centrality
    Notes
    -----
    The running time is $O((1/\epsilon^2)m{\sqrt k} \log n)$
```

```
 71: (4)                 and the space required is $O(m)$ for $n$ nodes and $m$ edges.
 72: (4)                 If the edges have a 'weight' attribute they will be used as
 73: (4)                 weights in this algorithm.  Unspecified weights are set to 1.
 74: (4)                 References
 75: (4)                 ----------
 76: (4)                 .. [1] Ulrik Brandes and Daniel Fleischer:
 77: (7)                    Centrality Measures Based on Current Flow.
 78: (7)                    Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05).
 79: (7)                    LNCS 3404, pp. 533-544. Springer-Verlag, 2005.
 80: (7)                    https://doi.org/10.1007/978-3-540-31856-9_44
 81: (4)                 """
 82: (4)                 import numpy as np
 83: (4)                 if not nx.is_connected(G):
 84: (8)                     raise nx.NetworkXError("Graph not connected.")
 85: (4)                 solvername = {
 86: (8)                     "full": FullInverseLaplacian,
 87: (8)                     "lu": SuperLUInverseLaplacian,
 88: (8)                     "cg": CGInverseLaplacian,
 89: (4)                 }
 90: (4)                 n = G.number_of_nodes()
 91: (4)                 ordering = list(reverse_cuthill_mckee_ordering(G))
 92: (4)                 H = nx.relabel_nodes(G, dict(zip(ordering, range(n))))
 93: (4)                 L = nx.laplacian_matrix(H, nodelist=range(n),
weight=weight).asformat("csc")
 94: (4)                 L = L.astype(dtype)
 95: (4)                 C = solvername[solver](L, dtype=dtype)  # initialize solver
 96: (4)                 betweenness = dict.fromkeys(H, 0.0)
 97: (4)                 nb = (n - 1.0) * (n - 2.0)  # normalization factor
 98: (4)                 cstar = n * (n - 1) / nb
 99: (4)                 l = 1  # parameter in approximation, adjustable
100: (4)                 k = l * int(np.ceil((cstar / epsilon) ** 2 * np.log(n)))
101: (4)                 if k > kmax:
102: (8)                     msg = f"Number random pairs k>kmax ({k}>{kmax}) "
103: (8)                     raise nx.NetworkXError(msg, "Increase kmax or epsilon")
104: (4)                 cstar2k = cstar / (2 * k)
105: (4)                 for _ in range(k):
106: (8)                     s, t = pair = seed.sample(range(n), 2)
107: (8)                     b = np.zeros(n, dtype=dtype)
108: (8)                     b[s] = 1
109: (8)                     b[t] = -1
110: (8)                     p = C.solve(b)
111: (8)                     for v in H:
112: (12)                        if v in pair:
113: (16)                            continue
114: (12)                        for nbr in H[v]:
115: (16)                            w = H[v][nbr].get(weight, 1.0)
116: (16)                            betweenness[v] += float(w * np.abs(p[v] - p[nbr]) * cstar2k)
117: (4)                 if normalized:
118: (8)                     factor = 1.0
119: (4)                 else:
120: (8)                     factor = nb / 2.0
121: (4)                 return {ordering[k]: v * factor for k, v in betweenness.items()}
122: (0)             @not_implemented_for("directed")
123: (0)             @nx._dispatchable(edge_attrs="weight")
124: (0)             def current_flow_betweenness_centrality(
125: (4)                 G, normalized=True, weight=None, dtype=float, solver="full"
126: (0)             ):
127: (4)                 r"""Compute current-flow betweenness centrality for nodes.
128: (4)                 Current-flow betweenness centrality uses an electrical current
129: (4)                 model for information spreading in contrast to betweenness
130: (4)                 centrality which uses shortest paths.
131: (4)                 Current-flow betweenness centrality is also known as
132: (4)                 random-walk betweenness centrality [2]_.
133: (4)                 Parameters
134: (4)                 ----------
135: (4)                 G : graph
136: (6)                   A NetworkX graph
137: (4)                 normalized : bool, optional (default=True)
138: (6)                   If True the betweenness values are normalized by 2/[(n-1)(n-2)] where
```

```
(6)                 n is the number of nodes in G.
(4)               weight : string or None, optional (default=None)
(6)                 Key for edge data used as the edge weight.
(6)                 If None, then use 1 as each edge weight.
(6)                 The weight reflects the capacity or the strength of the
(6)                 edge.
(4)               dtype : data type (float)
(6)                 Default data type for internal matrices.
(6)                 Set to np.float32 for lower memory consumption.
(4)               solver : string (default='full')
(7)                  Type of linear solver to use for computing the flow matrix.
(7)                  Options are "full" (uses most memory), "lu" (recommended), and
(7)                  "cg" (uses least memory).
(4)               Returns
(4)               -------
(4)               nodes : dictionary
(7)                  Dictionary of nodes with betweenness centrality as the value.
(4)               See Also
(4)               --------
(4)               approximate_current_flow_betweenness_centrality
(4)               betweenness_centrality
(4)               edge_betweenness_centrality
(4)               edge_current_flow_betweenness_centrality
(4)               Notes
(4)               -----
(4)               Current-flow betweenness can be computed in  $O(I(n-1)+mn \log n)$
(4)               time [1]_, where $I(n-1)$ is the time needed to compute the
(4)               inverse Laplacian.  For a full matrix this is $O(n^3)$ but using
(4)               sparse methods you can achieve $O(nm{\sqrt k})$ where $k$ is the
(4)               Laplacian matrix condition number.
(4)               The space required is $O(nw)$ where $w$ is the width of the sparse
(4)               Laplacian matrix.  Worse case is $w=n$ for $O(n^2)$.
(4)               If the edges have a 'weight' attribute they will be used as
(4)               weights in this algorithm.  Unspecified weights are set to 1.
(4)               References
(4)               ----------
(4)               .. [1] Centrality Measures Based on Current Flow.
(7)                  Ulrik Brandes and Daniel Fleischer,
(7)                  Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05).
(7)                  LNCS 3404, pp. 533-544. Springer-Verlag, 2005.
(7)                  https://doi.org/10.1007/978-3-540-31856-9_44
(4)               .. [2] A measure of betweenness centrality based on random walks,
(7)                  M. E. J. Newman, Social Networks 27, 39-54 (2005).
(4)               """
(4)               if not nx.is_connected(G):
(8)                   raise nx.NetworkXError("Graph not connected.")
(4)               N = G.number_of_nodes()
(4)               ordering = list(reverse_cuthill_mckee_ordering(G))
(4)               H = nx.relabel_nodes(G, dict(zip(ordering, range(N))))
(4)               betweenness = dict.fromkeys(H, 0.0)  # b[n]=0 for n in H
(4)               for row, (s, t) in flow_matrix_row(H, weight=weight, dtype=dtype,
solver=solver):
(8)                   pos = dict(zip(row.argsort()[::-1], range(N)))
(8)                   for i in range(N):
(12)                      betweenness[s] += (i - pos[i]) * row.item(i)
(12)                      betweenness[t] += (N - i - 1 - pos[i]) * row.item(i)
(4)               if normalized:
(8)                   nb = (N - 1.0) * (N - 2.0)  # normalization factor
(4)               else:
(8)                   nb = 2.0
(4)               return {ordering[n]: (b - n) * 2.0 / nb for n, b in betweenness.items()}
(0)           @not_implemented_for("directed")
(0)           @nx._dispatchable(edge_attrs="weight")
(0)           def edge_current_flow_betweenness_centrality(
(4)               G, normalized=True, weight=None, dtype=float, solver="full"
(0)           ):
(4)               r"""Compute current-flow betweenness centrality for edges.
(4)               Current-flow betweenness centrality uses an electrical current
(4)               model for information spreading in contrast to betweenness
```

```
(4)                 centrality which uses shortest paths.
(4)                 Current-flow betweenness centrality is also known as
(4)                 random-walk betweenness centrality [2]_.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A NetworkX graph
(4)                 normalized : bool, optional (default=True)
(6)                   If True the betweenness values are normalized by 2/[(n-1)(n-2)] where
(6)                   n is the number of nodes in G.
(4)                 weight : string or None, optional (default=None)
(6)                   Key for edge data used as the edge weight.
(6)                   If None, then use 1 as each edge weight.
(6)                   The weight reflects the capacity or the strength of the
(6)                   edge.
(4)                 dtype : data type (default=float)
(6)                   Default data type for internal matrices.
(6)                   Set to np.float32 for lower memory consumption.
(4)                 solver : string (default='full')
(7)                    Type of linear solver to use for computing the flow matrix.
(7)                    Options are "full" (uses most memory), "lu" (recommended), and
(7)                    "cg" (uses least memory).
(4)                 Returns
(4)                 -------
(4)                 nodes : dictionary
(7)                    Dictionary of edge tuples with betweenness centrality as the value.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     The algorithm does not support DiGraphs.
(8)                     If the input graph is an instance of DiGraph class, NetworkXError
(8)                     is raised.
(4)                 See Also
(4)                 --------
(4)                 betweenness_centrality
(4)                 edge_betweenness_centrality
(4)                 current_flow_betweenness_centrality
(4)                 Notes
(4)                 -----
(4)                 Current-flow betweenness can be computed in $O(I(n-1)+mn \log n)$
(4)                 time [1]_, where $I(n-1)$ is the time needed to compute the
(4)                 inverse Laplacian.  For a full matrix this is $O(n^3)$ but using
(4)                 sparse methods you can achieve $O(nm{\sqrt k})$ where $k$ is the
(4)                 Laplacian matrix condition number.
(4)                 The space required is $O(nw)$ where $w$ is the width of the sparse
(4)                 Laplacian matrix.  Worse case is $w=n$ for $O(n^2)$.
(4)                 If the edges have a 'weight' attribute they will be used as
(4)                 weights in this algorithm.  Unspecified weights are set to 1.
(4)                 References
(4)                 ----------
(4)                 .. [1] Centrality Measures Based on Current Flow.
(7)                    Ulrik Brandes and Daniel Fleischer,
(7)                    Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05).
(7)                    LNCS 3404, pp. 533-544. Springer-Verlag, 2005.
(7)                    https://doi.org/10.1007/978-3-540-31856-9_44
(4)                 .. [2] A measure of betweenness centrality based on random walks,
(7)                    M. E. J. Newman, Social Networks 27, 39-54 (2005).
(4)                 """
(4)                 if not nx.is_connected(G):
(8)                     raise nx.NetworkXError("Graph not connected.")
(4)                 N = G.number_of_nodes()
(4)                 ordering = list(reverse_cuthill_mckee_ordering(G))
(4)                 H = nx.relabel_nodes(G, dict(zip(ordering, range(N))))
(4)                 edges = (tuple(sorted((u, v))) for u, v in H.edges())
(4)                 betweenness = dict.fromkeys(edges, 0.0)
(4)                 if normalized:
(8)                     nb = (N - 1.0) * (N - 2.0)  # normalization factor
(4)                 else:
(8)                     nb = 2.0
```

```
276: (4)                for row, (e) in flow_matrix_row(H, weight=weight, dtype=dtype,
solver=solver):
277: (8)                    pos = dict(zip(row.argsort()[::-1], range(1, N + 1)))
278: (8)                    for i in range(N):
279: (12)                       betweenness[e] += (i + 1 - pos[i]) * row.item(i)
280: (12)                       betweenness[e] += (N - i - pos[i]) * row.item(i)
281: (8)                    betweenness[e] /= nb
282: (4)                return {(ordering[s], ordering[t]): b for (s, t), b in
betweenness.items()}

----------------------------------------

File 136 - current_flow_betweenness_subset.py:

1: (0)              """Current-flow betweenness centrality measures for subsets of nodes."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.centrality.flow_matrix import flow_matrix_row
4: (0)              from networkx.utils import not_implemented_for, reverse_cuthill_mckee_ordering
5: (0)              __all__ = [
6: (4)                  "current_flow_betweenness_centrality_subset",
7: (4)                  "edge_current_flow_betweenness_centrality_subset",
8: (0)              ]
9: (0)              @not_implemented_for("directed")
10: (0)             @nx._dispatchable(edge_attrs="weight")
11: (0)             def current_flow_betweenness_centrality_subset(
12: (4)                 G, sources, targets, normalized=True, weight=None, dtype=float,
solver="lu"
13: (0)             ):
14: (4)                 r"""Compute current-flow betweenness centrality for subsets of nodes.
15: (4)                 Current-flow betweenness centrality uses an electrical current
16: (4)                 model for information spreading in contrast to betweenness
17: (4)                 centrality which uses shortest paths.
18: (4)                 Current-flow betweenness centrality is also known as
19: (4)                 random-walk betweenness centrality [2]_.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : graph
23: (6)                   A NetworkX graph
24: (4)                 sources: list of nodes
25: (6)                   Nodes to use as sources for current
26: (4)                 targets: list of nodes
27: (6)                   Nodes to use as sinks for current
28: (4)                 normalized : bool, optional (default=True)
29: (6)                   If True the betweenness values are normalized by b=b/(n-1)(n-2) where
30: (6)                   n is the number of nodes in G.
31: (4)                 weight : string or None, optional (default=None)
32: (6)                   Key for edge data used as the edge weight.
33: (6)                   If None, then use 1 as each edge weight.
34: (6)                   The weight reflects the capacity or the strength of the
35: (6)                   edge.
36: (4)                 dtype: data type (float)
37: (6)                   Default data type for internal matrices.
38: (6)                   Set to np.float32 for lower memory consumption.
39: (4)                 solver: string (default='lu')
40: (7)                    Type of linear solver to use for computing the flow matrix.
41: (7)                    Options are "full" (uses most memory), "lu" (recommended), and
42: (7)                    "cg" (uses least memory).
43: (4)                 Returns
44: (4)                 -------
45: (4)                 nodes : dictionary
46: (7)                    Dictionary of nodes with betweenness centrality as the value.
47: (4)                 See Also
48: (4)                 --------
49: (4)                 approximate_current_flow_betweenness_centrality
50: (4)                 betweenness_centrality
51: (4)                 edge_betweenness_centrality
52: (4)                 edge_current_flow_betweenness_centrality
53: (4)                 Notes
54: (4)                 -----
```

```
 55: (4)                 Current-flow betweenness can be computed in $O(I(n-1)+mn \log n)$
 56: (4)                 time [1]_, where $I(n-1)$ is the time needed to compute the
 57: (4)                 inverse Laplacian.  For a full matrix this is $O(n^3)$ but using
 58: (4)                 sparse methods you can achieve $O(nm{\sqrt k})$ where $k$ is the
 59: (4)                 Laplacian matrix condition number.
 60: (4)                 The space required is $O(nw)$ where $w$ is the width of the sparse
 61: (4)                 Laplacian matrix.  Worse case is $w=n$ for $O(n^2)$.
 62: (4)                 If the edges have a 'weight' attribute they will be used as
 63: (4)                 weights in this algorithm.  Unspecified weights are set to 1.
 64: (4)                 References
 65: (4)                 ----------
 66: (4)                 .. [1] Centrality Measures Based on Current Flow.
 67: (7)                    Ulrik Brandes and Daniel Fleischer,
 68: (7)                    Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05).
 69: (7)                    LNCS 3404, pp. 533-544. Springer-Verlag, 2005.
 70: (7)                    https://doi.org/10.1007/978-3-540-31856-9_44
 71: (4)                 .. [2] A measure of betweenness centrality based on random walks,
 72: (7)                    M. E. J. Newman, Social Networks 27, 39-54 (2005).
 73: (4)                 """
 74: (4)                 import numpy as np
 75: (4)                 from networkx.utils import reverse_cuthill_mckee_ordering
 76: (4)                 if not nx.is_connected(G):
 77: (8)                     raise nx.NetworkXError("Graph not connected.")
 78: (4)                 N = G.number_of_nodes()
 79: (4)                 ordering = list(reverse_cuthill_mckee_ordering(G))
 80: (4)                 mapping = dict(zip(ordering, range(N)))
 81: (4)                 H = nx.relabel_nodes(G, mapping)
 82: (4)                 betweenness = dict.fromkeys(H, 0.0)  # b[n]=0 for n in H
 83: (4)                 for row, (s, t) in flow_matrix_row(H, weight=weight, dtype=dtype,
solver=solver):
 84: (8)                     for ss in sources:
 85: (12)                        i = mapping[ss]
 86: (12)                        for tt in targets:
 87: (16)                            j = mapping[tt]
 88: (16)                            betweenness[s] += 0.5 * abs(row.item(i) - row.item(j))
 89: (16)                            betweenness[t] += 0.5 * abs(row.item(i) - row.item(j))
 90: (4)                 if normalized:
 91: (8)                     nb = (N - 1.0) * (N - 2.0)  # normalization factor
 92: (4)                 else:
 93: (8)                     nb = 2.0
 94: (4)                 for node in H:
 95: (8)                     betweenness[node] = betweenness[node] / nb + 1.0 / (2 - N)
 96: (4)                 return {ordering[node]: value for node, value in betweenness.items()}
 97: (0)             @not_implemented_for("directed")
 98: (0)             @nx._dispatchable(edge_attrs="weight")
 99: (0)             def edge_current_flow_betweenness_centrality_subset(
100: (4)                 G, sources, targets, normalized=True, weight=None, dtype=float,
solver="lu"
101: (0)             ):
102: (4)                 r"""Compute current-flow betweenness centrality for edges using subsets
103: (4)                 of nodes.
104: (4)                 Current-flow betweenness centrality uses an electrical current
105: (4)                 model for information spreading in contrast to betweenness
106: (4)                 centrality which uses shortest paths.
107: (4)                 Current-flow betweenness centrality is also known as
108: (4)                 random-walk betweenness centrality [2]_.
109: (4)                 Parameters
110: (4)                 ----------
111: (4)                 G : graph
112: (6)                   A NetworkX graph
113: (4)                 sources: list of nodes
114: (6)                   Nodes to use as sources for current
115: (4)                 targets: list of nodes
116: (6)                   Nodes to use as sinks for current
117: (4)                 normalized : bool, optional (default=True)
118: (6)                   If True the betweenness values are normalized by b=b/(n-1)(n-2) where
119: (6)                   n is the number of nodes in G.
120: (4)                 weight : string or None, optional (default=None)
121: (6)                   Key for edge data used as the edge weight.
```

```
122: (6)                  If None, then use 1 as each edge weight.
123: (6)                  The weight reflects the capacity or the strength of the
124: (6)                  edge.
125: (4)                dtype: data type (float)
126: (6)                  Default data type for internal matrices.
127: (6)                  Set to np.float32 for lower memory consumption.
128: (4)                solver: string (default='lu')
129: (7)                   Type of linear solver to use for computing the flow matrix.
130: (7)                   Options are "full" (uses most memory), "lu" (recommended), and
131: (7)                   "cg" (uses least memory).
132: (4)                Returns
133: (4)                -------
134: (4)                nodes : dict
135: (7)                   Dictionary of edge tuples with betweenness centrality as the value.
136: (4)                See Also
137: (4)                --------
138: (4)                betweenness_centrality
139: (4)                edge_betweenness_centrality
140: (4)                current_flow_betweenness_centrality
141: (4)                Notes
142: (4)                -----
143: (4)                Current-flow betweenness can be computed in $O(I(n-1)+mn \log n)$
144: (4)                time [1]_, where $I(n-1)$ is the time needed to compute the
145: (4)                inverse Laplacian.  For a full matrix this is $O(n^3)$ but using
146: (4)                sparse methods you can achieve $O(nm{\sqrt k})$ where $k$ is the
147: (4)                Laplacian matrix condition number.
148: (4)                The space required is $O(nw)$ where $w$ is the width of the sparse
149: (4)                Laplacian matrix.  Worse case is $w=n$ for $O(n^2)$.
150: (4)                If the edges have a 'weight' attribute they will be used as
151: (4)                weights in this algorithm.  Unspecified weights are set to 1.
152: (4)                References
153: (4)                ----------
154: (4)                .. [1] Centrality Measures Based on Current Flow.
155: (7)                   Ulrik Brandes and Daniel Fleischer,
156: (7)                   Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05).
157: (7)                   LNCS 3404, pp. 533-544. Springer-Verlag, 2005.
158: (7)                   https://doi.org/10.1007/978-3-540-31856-9_44
159: (4)                .. [2] A measure of betweenness centrality based on random walks,
160: (7)                   M. E. J. Newman, Social Networks 27, 39-54 (2005).
161: (4)                """
162: (4)                import numpy as np
163: (4)                if not nx.is_connected(G):
164: (8)                    raise nx.NetworkXError("Graph not connected.")
165: (4)                N = G.number_of_nodes()
166: (4)                ordering = list(reverse_cuthill_mckee_ordering(G))
167: (4)                mapping = dict(zip(ordering, range(N)))
168: (4)                H = nx.relabel_nodes(G, mapping)
169: (4)                edges = (tuple(sorted((u, v))) for u, v in H.edges())
170: (4)                betweenness = dict.fromkeys(edges, 0.0)
171: (4)                if normalized:
172: (8)                    nb = (N - 1.0) * (N - 2.0)  # normalization factor
173: (4)                else:
174: (8)                    nb = 2.0
175: (4)                for row, (e) in flow_matrix_row(H, weight=weight, dtype=dtype,
solver=solver):
176: (8)                    for ss in sources:
177: (12)                       i = mapping[ss]
178: (12)                       for tt in targets:
179: (16)                           j = mapping[tt]
180: (16)                           betweenness[e] += 0.5 * abs(row.item(i) - row.item(j))
181: (8)                    betweenness[e] /= nb
182: (4)                return {(ordering[s], ordering[t]): value for (s, t), value in
betweenness.items()}

----------------------------------------

File 137 - degree_alg.py:

1: (0)              """Degree centrality measures."""
```

```
import networkx as nx
from networkx.utils.decorators import not_implemented_for
__all__ = ["degree_centrality", "in_degree_centrality", "out_degree_centrality"]
@nx._dispatchable
def degree_centrality(G):
    """Compute the degree centrality for nodes.
    The degree centrality for a node v is the fraction of nodes it
    is connected to.
    Parameters
    ----------
    G : graph
      A networkx graph
    Returns
    -------
    nodes : dictionary
       Dictionary of nodes with degree centrality as the value.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
    >>> nx.degree_centrality(G)
    {0: 1.0, 1: 1.0, 2: 0.6666666666666666, 3: 0.6666666666666666}
    See Also
    --------
    betweenness_centrality, load_centrality, eigenvector_centrality
    Notes
    -----
    The degree centrality values are normalized by dividing by the maximum
    possible degree in a simple graph n-1 where n is the number of nodes in G.
    For multigraphs or graphs with self loops the maximum degree might
    be higher than n-1 and values of degree centrality greater than 1
    are possible.
    """
    if len(G) <= 1:
        return {n: 1 for n in G}
    s = 1.0 / (len(G) - 1.0)
    centrality = {n: d * s for n, d in G.degree()}
    return centrality
@not_implemented_for("undirected")
@nx._dispatchable
def in_degree_centrality(G):
    """Compute the in-degree centrality for nodes.
    The in-degree centrality for a node v is the fraction of nodes its
    incoming edges are connected to.
    Parameters
    ----------
    G : graph
        A NetworkX graph
    Returns
    -------
    nodes : dictionary
        Dictionary of nodes with in-degree centrality as values.
    Raises
    ------
    NetworkXNotImplemented
        If G is undirected.
    Examples
    --------
    >>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
    >>> nx.in_degree_centrality(G)
    {0: 0.0, 1: 0.3333333333333333, 2: 0.6666666666666666, 3: 0.6666666666666666}
    See Also
    --------
    degree_centrality, out_degree_centrality
    Notes
    -----
    The degree centrality values are normalized by dividing by the maximum
    possible degree in a simple graph n-1 where n is the number of nodes in G.
```

```
69: (4)                 For multigraphs or graphs with self loops the maximum degree might
70: (4)                 be higher than n-1 and values of degree centrality greater than 1
71: (4)                 are possible.
72: (4)                 """
73: (4)                 if len(G) <= 1:
74: (8)                     return {n: 1 for n in G}
75: (4)                 s = 1.0 / (len(G) - 1.0)
76: (4)                 centrality = {n: d * s for n, d in G.in_degree()}
77: (4)                 return centrality
78: (0)             @not_implemented_for("undirected")
79: (0)             @nx._dispatchable
80: (0)             def out_degree_centrality(G):
81: (4)                 """Compute the out-degree centrality for nodes.
82: (4)                 The out-degree centrality for a node v is the fraction of nodes its
83: (4)                 outgoing edges are connected to.
84: (4)                 Parameters
85: (4)                 ----------
86: (4)                 G : graph
87: (8)                     A NetworkX graph
88: (4)                 Returns
89: (4)                 -------
90: (4)                 nodes : dictionary
91: (8)                     Dictionary of nodes with out-degree centrality as values.
92: (4)                 Raises
93: (4)                 ------
94: (4)                 NetworkXNotImplemented
95: (8)                     If G is undirected.
96: (4)                 Examples
97: (4)                 --------
98: (4)                 >>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
99: (4)                 >>> nx.out_degree_centrality(G)
100: (4)                {0: 1.0, 1: 0.6666666666666666, 2: 0.0, 3: 0.0}
101: (4)                See Also
102: (4)                --------
103: (4)                degree_centrality, in_degree_centrality
104: (4)                Notes
105: (4)                -----
106: (4)                The degree centrality values are normalized by dividing by the maximum
107: (4)                possible degree in a simple graph n-1 where n is the number of nodes in G.
108: (4)                For multigraphs or graphs with self loops the maximum degree might
109: (4)                be higher than n-1 and values of degree centrality greater than 1
110: (4)                are possible.
111: (4)                """
112: (4)                if len(G) <= 1:
113: (8)                    return {n: 1 for n in G}
114: (4)                s = 1.0 / (len(G) - 1.0)
115: (4)                centrality = {n: d * s for n, d in G.out_degree()}
116: (4)                return centrality

----------------------------------------

File 138 - current_flow_closeness.py:

1: (0)              """Current-flow closeness centrality measures."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.centrality.flow_matrix import (
4: (4)                  CGInverseLaplacian,
5: (4)                  FullInverseLaplacian,
6: (4)                  SuperLUInverseLaplacian,
7: (0)              )
8: (0)              from networkx.utils import not_implemented_for, reverse_cuthill_mckee_ordering
9: (0)              __all__ = ["current_flow_closeness_centrality", "information_centrality"]
10: (0)             @not_implemented_for("directed")
11: (0)             @nx._dispatchable(edge_attrs="weight")
12: (0)             def current_flow_closeness_centrality(G, weight=None, dtype=float,
solver="lu"):
13: (4)                 """Compute current-flow closeness centrality for nodes.
14: (4)                 Current-flow closeness centrality is variant of closeness
15: (4)                 centrality based on effective resistance between nodes in
```

```
16: (4)                 a network. This metric is also known as information centrality.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : graph
20: (6)                   A NetworkX graph.
21: (4)                 weight : None or string, optional (default=None)
22: (6)                   If None, all edge weights are considered equal.
23: (6)                   Otherwise holds the name of the edge attribute used as weight.
24: (6)                   The weight reflects the capacity or the strength of the
25: (6)                   edge.
26: (4)                 dtype: data type (default=float)
27: (6)                   Default data type for internal matrices.
28: (6)                   Set to np.float32 for lower memory consumption.
29: (4)                 solver: string (default='lu')
30: (7)                    Type of linear solver to use for computing the flow matrix.
31: (7)                    Options are "full" (uses most memory), "lu" (recommended), and
32: (7)                    "cg" (uses least memory).
33: (4)                 Returns
34: (4)                 -------
35: (4)                 nodes : dictionary
36: (7)                    Dictionary of nodes with current flow closeness centrality as the
value.
37: (4)                 See Also
38: (4)                 --------
39: (4)                 closeness_centrality
40: (4)                 Notes
41: (4)                 -----
42: (4)                 The algorithm is from Brandes [1]_.
43: (4)                 See also [2]_ for the original definition of information centrality.
44: (4)                 References
45: (4)                 ----------
46: (4)                 .. [1] Ulrik Brandes and Daniel Fleischer,
47: (7)                    Centrality Measures Based on Current Flow.
48: (7)                    Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05).
49: (7)                    LNCS 3404, pp. 533-544. Springer-Verlag, 2005.
50: (7)                    https://doi.org/10.1007/978-3-540-31856-9_44
51: (4)                 .. [2] Karen Stephenson and Marvin Zelen:
52: (7)                    Rethinking centrality: Methods and examples.
53: (7)                    Social Networks 11(1):1-37, 1989.
54: (7)                    https://doi.org/10.1016/0378-8733(89)90016-6
55: (4)                 """
56: (4)                 if not nx.is_connected(G):
57: (8)                     raise nx.NetworkXError("Graph not connected.")
58: (4)                 solvername = {
59: (8)                     "full": FullInverseLaplacian,
60: (8)                     "lu": SuperLUInverseLaplacian,
61: (8)                     "cg": CGInverseLaplacian,
62: (4)                 }
63: (4)                 N = G.number_of_nodes()
64: (4)                 ordering = list(reverse_cuthill_mckee_ordering(G))
65: (4)                 H = nx.relabel_nodes(G, dict(zip(ordering, range(N))))
66: (4)                 betweenness = dict.fromkeys(H, 0.0)  # b[n]=0 for n in H
67: (4)                 N = H.number_of_nodes()
68: (4)                 L = nx.laplacian_matrix(H, nodelist=range(N),
weight=weight).asformat("csc")
69: (4)                 L = L.astype(dtype)
70: (4)                 C2 = solvername[solver](L, width=1, dtype=dtype)  # initialize solver
71: (4)                 for v in H:
72: (8)                     col = C2.get_row(v)
73: (8)                     for w in H:
74: (12)                        betweenness[v] += col.item(v) - 2 * col.item(w)
75: (12)                        betweenness[w] += col.item(v)
76: (4)                 return {ordering[node]: 1 / value for node, value in betweenness.items()}
77: (0)             information_centrality = current_flow_closeness_centrality

----------------------------------------

File 139 - dispersion.py:
```

```
(0)  from itertools import combinations
(0)  import networkx as nx
(0)  __all__ = ["dispersion"]
(0)  @nx._dispatchable
(0)  def dispersion(G, u=None, v=None, normalized=True, alpha=1.0, b=0.0, c=0.0):
(4)      r"""Calculate dispersion between `u` and `v` in `G`.
(4)      A link between two actors (`u` and `v`) has a high dispersion when their
(4)      mutual ties (`s` and `t`) are not well connected with each other.
(4)      Parameters
(4)      ----------
(4)      G : graph
(8)          A NetworkX graph.
(4)      u : node, optional
(8)          The source for the dispersion score (e.g. ego node of the network).
(4)      v : node, optional
(8)          The target of the dispersion score if specified.
(4)      normalized : bool
(8)          If True (default) normalize by the embeddedness of the nodes (u and
v).
(4)      alpha, b, c : float
(8)          Parameters for the normalization procedure. When `normalized` is True,
(8)          the dispersion value is normalized by::
(12)             result = ((dispersion + b) ** alpha) / (embeddedness + c)
(8)          as long as the denominator is nonzero.
(4)      Returns
(4)      -------
(4)      nodes : dictionary
(8)          If u (v) is specified, returns a dictionary of nodes with dispersion
(8)          score for all "target" ("source") nodes. If neither u nor v is
(8)          specified, returns a dictionary of dictionaries for all nodes 'u' in
the
(8)          graph with a dispersion score for each node 'v'.
(4)      Notes
(4)      -----
(4)      This implementation follows Lars Backstrom and Jon Kleinberg [1]_. Typical
(4)      usage would be to run dispersion on the ego network $G_u$ if $u$ were
(4)      specified.  Running :func:`dispersion` with neither $u$ nor $v$ specified
(4)      can take some time to complete.
(4)      References
(4)      ----------
(4)      .. [1] Romantic Partnerships and the Dispersion of Social Ties:
(8)          A Network Analysis of Relationship Status on Facebook.
(8)          Lars Backstrom, Jon Kleinberg.
(8)          https://arxiv.org/pdf/1310.6753v1.pdf
(4)      """
(4)      def _dispersion(G_u, u, v):
(8)          """dispersion for all nodes 'v' in a ego network G_u of node 'u'"""
(8)          u_nbrs = set(G_u[u])
(8)          ST = {n for n in G_u[v] if n in u_nbrs}
(8)          set_uv = {u, v}
(8)          possib = combinations(ST, 2)
(8)          total = 0
(8)          for s, t in possib:
(12)             nbrs_s = u_nbrs.intersection(G_u[s]) - set_uv
(12)             if t not in nbrs_s:
(16)                 if nbrs_s.isdisjoint(G_u[t]):
(20)                     total += 1
(8)          embeddedness = len(ST)
(8)          dispersion_val = total
(8)          if normalized:
(12)             dispersion_val = (total + b) ** alpha
(12)             if embeddedness + c != 0:
(16)                 dispersion_val /= embeddedness + c
(8)          return dispersion_val
(4)      if u is None:
(8)          if v is None:
(12)             results = {n: {} for n in G}
(12)             for u in G:
(16)                 for v in G[u]:
```

```
68: (20)                                results[u][v] = _dispersion(G, u, v)
69: (8)                     else:
70: (12)                        results = dict.fromkeys(G[v], {})
71: (12)                        for u in G[v]:
72: (16)                            results[u] = _dispersion(G, v, u)
73: (4)                 else:
74: (8)                     if v is None:
75: (12)                        results = dict.fromkeys(G[u], {})
76: (12)                        for v in G[u]:
77: (16)                            results[v] = _dispersion(G, u, v)
78: (8)                     else:
79: (12)                        results = _dispersion(G, u, v)
80: (4)                 return results
```

File 140 - eigenvector.py:

```
1: (0)              """Functions for computing eigenvector centrality."""
2: (0)              import math
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["eigenvector_centrality", "eigenvector_centrality_numpy"]
6: (0)              @not_implemented_for("multigraph")
7: (0)              @nx._dispatchable(edge_attrs="weight")
8: (0)              def eigenvector_centrality(G, max_iter=100, tol=1.0e-6, nstart=None,
weight=None):
9: (4)                  r"""Compute the eigenvector centrality for the graph G.
10: (4)                 Eigenvector centrality computes the centrality for a node by adding
11: (4)                 the centrality of its predecessors. The centrality for node $i$ is the
12: (4)                 $i$-th element of a left eigenvector associated with the eigenvalue
$\lambda$
13: (4)                 of maximum modulus that is positive. Such an eigenvector $x$ is
14: (4)                 defined up to a multiplicative constant by the equation
15: (4)                 .. math::
16: (9)                      \lambda x^T = x^T A,
17: (4)                 where $A$ is the adjacency matrix of the graph G. By definition of
18: (4)                 row-column product, the equation above is equivalent to
19: (4)                 .. math::
20: (8)                     \lambda x_i = \sum_{j\to i}x_j.
21: (4)                 That is, adding the eigenvector centralities of the predecessors of
22: (4)                 $i$ one obtains the eigenvector centrality of $i$ multiplied by
23: (4)                 $\lambda$. In the case of undirected graphs, $x$ also solves the familiar
24: (4)                 right-eigenvector equation $Ax = \lambda x$.
25: (4)                 By virtue of the Perron–Frobenius theorem [1]_, if G is strongly
26: (4)                 connected there is a unique eigenvector $x$, and all its entries
27: (4)                 are strictly positive.
28: (4)                 If G is not strongly connected there might be several left
29: (4)                 eigenvectors associated with $\lambda$, and some of their elements
30: (4)                 might be zero.
31: (4)                 Parameters
32: (4)                 ----------
33: (4)                 G : graph
34: (6)                   A networkx graph.
35: (4)                 max_iter : integer, optional (default=100)
36: (6)                   Maximum number of power iterations.
37: (4)                 tol : float, optional (default=1.0e-6)
38: (6)                   Error tolerance (in Euclidean norm) used to check convergence in
39: (6)                   power iteration.
40: (4)                 nstart : dictionary, optional (default=None)
41: (6)                   Starting value of power iteration for each node. Must have a nonzero
42: (6)                   projection on the desired eigenvector for the power method to converge.
43: (6)                   If None, this implementation uses an all-ones vector, which is a safe
44: (6)                   choice.
45: (4)                 weight : None or string, optional (default=None)
46: (6)                   If None, all edge weights are considered equal. Otherwise holds the
47: (6)                   name of the edge attribute used as weight. In this measure the
48: (6)                   weight is interpreted as the connection strength.
49: (4)                 Returns
```

```
(4)                 -------
(4)                 nodes : dictionary
(7)                    Dictionary of nodes with eigenvector centrality as the value. The
(7)                    associated vector has unit Euclidean norm and the values are
(7)                    nonegative.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> centrality = nx.eigenvector_centrality(G)
(4)                 >>> sorted((v, f"{c:0.2f}") for v, c in centrality.items())
(4)                 [(0, '0.37'), (1, '0.60'), (2, '0.60'), (3, '0.37')]
(4)                 Raises
(4)                 ------
(4)                 NetworkXPointlessConcept
(8)                     If the graph G is the null graph.
(4)                 NetworkXError
(8)                     If each value in `nstart` is zero.
(4)                 PowerIterationFailedConvergence
(8)                     If the algorithm fails to converge to the specified tolerance
(8)                     within the specified number of iterations of the power iteration
(8)                     method.
(4)                 See Also
(4)                 --------
(4)                 eigenvector_centrality_numpy
(4)                 :func:`~networkx.algorithms.link_analysis.pagerank_alg.pagerank`
(4)                 :func:`~networkx.algorithms.link_analysis.hits_alg.hits`
(4)                 Notes
(4)                 -----
(4)                 Eigenvector centrality was introduced by Landau [2]_ for chess
(4)                 tournaments. It was later rediscovered by Wei [3]_ and then
(4)                 popularized by Kendall [4]_ in the context of sport ranking. Berge
(4)                 introduced a general definition for graphs based on social connections
(4)                 [5]_. Bonacich [6]_ reintroduced again eigenvector centrality and made
(4)                 it popular in link analysis.
(4)                 This function computes the left dominant eigenvector, which corresponds
(4)                 to adding the centrality of predecessors: this is the usual approach.
(4)                 To add the centrality of successors first reverse the graph with
(4)                 ``G.reverse()``.
(4)                 The implementation uses power iteration [7]_ to compute a dominant
(4)                 eigenvector starting from the provided vector `nstart`. Convergence is
(4)                 guaranteed as long as `nstart` has a nonzero projection on a dominant
(4)                 eigenvector, which certainly happens using the default value.
(4)                 The method stops when the change in the computed vector between two
(4)                 iterations is smaller than an error tolerance of ``G.number_of_nodes()
(4)                 * tol`` or after ``max_iter`` iterations, but in the second case it
(4)                 raises an exception.
(4)                 This implementation uses $(A + I)$ rather than the adjacency matrix
(4)                 $A$ because the change preserves eigenvectors, but it shifts the
(4)                 spectrum, thus guaranteeing convergence even for networks with
(4)                 negative eigenvalues of maximum modulus.
(4)                 References
(4)                 ----------
(4)                 .. [1] Abraham Berman and Robert J. Plemmons.
(7)                    "Nonnegative Matrices in the Mathematical Sciences."
(7)                    Classics in Applied Mathematics. SIAM, 1994.
(4)                 .. [2] Edmund Landau.
(7)                    "Zur relativen Wertbemessung der Turnierresultate."
(7)                    Deutsches Wochenschach, 11:366–369, 1895.
(4)                 .. [3] Teh-Hsing Wei.
(7)                    "The Algebraic Foundations of Ranking Theory."
(7)                    PhD thesis, University of Cambridge, 1952.
(4)                 .. [4] Maurice G. Kendall.
(7)                    "Further contributions to the theory of paired comparisons."
(7)                    Biometrics, 11(1):43–62, 1955.
(7)                    https://www.jstor.org/stable/3001479
(4)                 .. [5] Claude Berge
(7)                    "Théorie des graphes et ses applications."
(7)                    Dunod, Paris, France, 1958.
(4)                 .. [6] Phillip Bonacich.
```

```
(7)                 "Technique for analyzing overlapping memberships."
(7)                 Sociological Methodology, 4:176–185, 1972.
(7)                 https://www.jstor.org/stable/270732
(4)             .. [7] Power iteration:: https://en.wikipedia.org/wiki/Power_iteration
(4)             """
(4)             if len(G) == 0:
(8)                 raise nx.NetworkXPointlessConcept(
(12)                    "cannot compute centrality for the null graph"
(8)                 )
(4)             if nstart is None:
(8)                 nstart = {v: 1 for v in G}
(4)             if all(v == 0 for v in nstart.values()):
(8)                 raise nx.NetworkXError("initial vector cannot have all zero values")
(4)             nstart_sum = sum(nstart.values())
(4)             x = {k: v / nstart_sum for k, v in nstart.items()}
(4)             nnodes = G.number_of_nodes()
(4)             for _ in range(max_iter):
(8)                 xlast = x
(8)                 x = xlast.copy()  # Start with xlast times I to iterate with (A+I)
(8)                 for n in x:
(12)                    for nbr in G[n]:
(16)                        w = G[n][nbr].get(weight, 1) if weight else 1
(16)                        x[nbr] += xlast[n] * w
(8)                 norm = math.hypot(*x.values()) or 1
(8)                 x = {k: v / norm for k, v in x.items()}
(8)                 if sum(abs(x[n] - xlast[n]) for n in x) < nnodes * tol:
(12)                    return x
(4)             raise nx.PowerIterationFailedConvergence(max_iter)
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def eigenvector_centrality_numpy(G, weight=None, max_iter=50, tol=0):
(4)             r"""Compute the eigenvector centrality for the graph G.
(4)             Eigenvector centrality computes the centrality for a node by adding
(4)             the centrality of its predecessors. The centrality for node $i$ is the
(4)             $i$-th element of a left eigenvector associated with the eigenvalue
$\lambda$
(4)             of maximum modulus that is positive. Such an eigenvector $x$ is
(4)             defined up to a multiplicative constant by the equation
(4)             .. math::
(9)                  \lambda x^T = x^T A,
(4)             where $A$ is the adjacency matrix of the graph G. By definition of
(4)             row-column product, the equation above is equivalent to
(4)             .. math::
(8)                 \lambda x_i = \sum_{j\to i}x_j.
(4)             That is, adding the eigenvector centralities of the predecessors of
(4)             $i$ one obtains the eigenvector centrality of $i$ multiplied by
(4)             $\lambda$. In the case of undirected graphs, $x$ also solves the familiar
(4)             right-eigenvector equation $Ax = \lambda x$.
(4)             By virtue of the Perron–Frobenius theorem [1]_, if G is strongly
(4)             connected there is a unique eigenvector $x$, and all its entries
(4)             are strictly positive.
(4)             If G is not strongly connected there might be several left
(4)             eigenvectors associated with $\lambda$, and some of their elements
(4)             might be zero.
(4)             Parameters
(4)             ----------
(4)             G : graph
(6)               A networkx graph.
(4)             max_iter : integer, optional (default=50)
(6)               Maximum number of Arnoldi update iterations allowed.
(4)             tol : float, optional (default=0)
(6)               Relative accuracy for eigenvalues (stopping criterion).
(6)               The default value of 0 implies machine precision.
(4)             weight : None or string, optional (default=None)
(6)               If None, all edge weights are considered equal. Otherwise holds the
(6)               name of the edge attribute used as weight. In this measure the
(6)               weight is interpreted as the connection strength.
(4)             Returns
(4)             -------
(4)             nodes : dictionary
```

```
   Dictionary of nodes with eigenvector centrality as the value. The
   associated vector has unit Euclidean norm and the values are
   nonegative.
Examples
--------
>>> G = nx.path_graph(4)
>>> centrality = nx.eigenvector_centrality_numpy(G)
>>> print([f"{node} {centrality[node]:0.2f}" for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
Raises
------
NetworkXPointlessConcept
    If the graph G is the null graph.
ArpackNoConvergence
    When the requested convergence is not obtained. The currently
    converged eigenvalues and eigenvectors can be found as
    eigenvalues and eigenvectors attributes of the exception object.
See Also
--------
:func:`scipy.sparse.linalg.eigs`
eigenvector_centrality
:func:`~networkx.algorithms.link_analysis.pagerank_alg.pagerank`
:func:`~networkx.algorithms.link_analysis.hits_alg.hits`
Notes
-----
Eigenvector centrality was introduced by Landau [2]_ for chess
tournaments. It was later rediscovered by Wei [3]_ and then
popularized by Kendall [4]_ in the context of sport ranking. Berge
introduced a general definition for graphs based on social connections
[5]_. Bonacich [6]_ reintroduced again eigenvector centrality and made
it popular in link analysis.
This function computes the left dominant eigenvector, which corresponds
to adding the centrality of predecessors: this is the usual approach.
To add the centrality of successors first reverse the graph with
``G.reverse()``.
This implementation uses the
:func:`SciPy sparse eigenvalue solver<scipy.sparse.linalg.eigs>` (ARPACK)
to find the largest eigenvalue/eigenvector pair using Arnoldi iterations
[7]_.
References
----------
.. [1] Abraham Berman and Robert J. Plemmons.
   "Nonnegative Matrices in the Mathematical Sciences."
   Classics in Applied Mathematics. SIAM, 1994.
.. [2] Edmund Landau.
   "Zur relativen Wertbemessung der Turnierresultate."
   Deutsches Wochenschach, 11:366–369, 1895.
.. [3] Teh-Hsing Wei.
   "The Algebraic Foundations of Ranking Theory."
   PhD thesis, University of Cambridge, 1952.
.. [4] Maurice G. Kendall.
   "Further contributions to the theory of paired comparisons."
   Biometrics, 11(1):43–62, 1955.
   https://www.jstor.org/stable/3001479
.. [5] Claude Berge
   "Théorie des graphes et ses applications."
   Dunod, Paris, France, 1958.
.. [6] Phillip Bonacich.
   "Technique for analyzing overlapping memberships."
   Sociological Methodology, 4:176–185, 1972.
   https://www.jstor.org/stable/270732
.. [7] Arnoldi iteration:: https://en.wikipedia.org/wiki/Arnoldi_iteration
"""
import numpy as np
import scipy as sp
if len(G) == 0:
    raise nx.NetworkXPointlessConcept(
        "cannot compute centrality for the null graph"
    )
```

```
256: (4)                M = nx.to_scipy_sparse_array(G, nodelist=list(G), weight=weight,
dtype=float)
257: (4)                _, eigenvector = sp.sparse.linalg.eigs(
258: (8)                    M.T, k=1, which="LR", maxiter=max_iter, tol=tol
259: (4)                )
260: (4)                largest = eigenvector.flatten().real
261: (4)                norm = np.sign(largest.sum()) * sp.linalg.norm(largest)
262: (4)                return dict(zip(G, (largest / norm).tolist()))
```

File 141 - group.py:

```
1: (0)              """Group centrality measures."""
2: (0)              from copy import deepcopy
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.centrality.betweenness import (
5: (4)                  _accumulate_endpoints,
6: (4)                  _single_source_dijkstra_path_basic,
7: (4)                  _single_source_shortest_path_basic,
8: (0)              )
9: (0)              from networkx.utils.decorators import not_implemented_for
10: (0)             __all__ = [
11: (4)                 "group_betweenness_centrality",
12: (4)                 "group_closeness_centrality",
13: (4)                 "group_degree_centrality",
14: (4)                 "group_in_degree_centrality",
15: (4)                 "group_out_degree_centrality",
16: (4)                 "prominent_group",
17: (0)             ]
18: (0)             @nx._dispatchable(edge_attrs="weight")
19: (0)             def group_betweenness_centrality(G, C, normalized=True, weight=None,
endpoints=False):
20: (4)                 r"""Compute the group betweenness centrality for a group of nodes.
21: (4)                 Group betweenness centrality of a group of nodes $C$ is the sum of the
22: (4)                 fraction of all-pairs shortest paths that pass through any vertex in $C$
23: (4)                 .. math::
24: (7)                    c_B(v) =\sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}
25: (4)                 where $V$ is the set of nodes, $\sigma(s, t)$ is the number of
26: (4)                 shortest $(s, t)$-paths, and $\sigma(s, t|C)$ is the number of
27: (4)                 those paths passing through some node in group $C$. Note that
28: (4)                 $(s, t)$ are not members of the group ($V-C$ is the set of nodes
29: (4)                 in $V$ that are not in $C$).
30: (4)                 Parameters
31: (4)                 ----------
32: (4)                 G : graph
33: (6)                   A NetworkX graph.
34: (4)                 C : list or set or list of lists or list of sets
35: (6)                   A group or a list of groups containing nodes which belong to G, for
which group betweenness
36: (6)                   centrality is to be calculated.
37: (4)                 normalized : bool, optional (default=True)
38: (6)                   If True, group betweenness is normalized by `1/((|V|-|C|)(|V|-|C|-1))`
39: (6)                   where `|V|` is the number of nodes in G and `|C|` is the number of nodes
in C.
40: (4)                 weight : None or string, optional (default=None)
41: (6)                   If None, all edge weights are considered equal.
42: (6)                   Otherwise holds the name of the edge attribute used as weight.
43: (6)                   The weight of an edge is treated as the length or distance between the
two sides.
44: (4)                 endpoints : bool, optional (default=False)
45: (6)                   If True include the endpoints in the shortest path counts.
46: (4)                 Raises
47: (4)                 ------
48: (4)                 NodeNotFound
49: (7)                    If node(s) in C are not present in G.
50: (4)                 Returns
51: (4)                 -------
52: (4)                 betweenness : list of floats or float
```

```
53: (7)                    If C is a single group then return a float. If C is a list with
54: (7)                    several groups then return a list of group betweenness centralities.
55: (4)                 See Also
56: (4)                 --------
57: (4)                 betweenness_centrality
58: (4)                 Notes
59: (4)                 -----
60: (4)                 Group betweenness centrality is described in [1]_ and its importance
discussed in [3]_.
61: (4)                 The initial implementation of the algorithm is mentioned in [2]_. This
function uses
62: (4)                 an improved algorithm presented in [4]_.
63: (4)                 The number of nodes in the group must be a maximum of n - 2 where `n`
64: (4)                 is the total number of nodes in the graph.
65: (4)                 For weighted graphs the edge weights must be greater than zero.
66: (4)                 Zero edge weights can produce an infinite number of equal length
67: (4)                 paths between pairs of nodes.
68: (4)                 The total number of paths between source and target is counted
69: (4)                 differently for directed and undirected graphs. Directed paths
70: (4)                 between "u" and "v" are counted as two possible paths (one each
71: (4)                 direction) while undirected paths between "u" and "v" are counted
72: (4)                 as one path. Said another way, the sum in the expression above is
73: (4)                 over all ``s != t`` for directed graphs and for ``s < t`` for undirected
graphs.
74: (4)                 References
75: (4)                 ----------
76: (4)                 .. [1] M G Everett and S P Borgatti:
77: (7)                    The Centrality of Groups and Classes.
78: (7)                    Journal of Mathematical Sociology. 23(3): 181-201. 1999.
79: (7)                    http://www.analytictech.com/borgatti/group_centrality.htm
80: (4)                 .. [2] Ulrik Brandes:
81: (7)                    On Variants of Shortest-Path Betweenness
82: (7)                    Centrality and their Generic Computation.
83: (7)                    Social Networks 30(2):136-145, 2008.
84: (7)                    http://citeseerx.ist.psu.edu/viewdoc/download?
doi=10.1.1.72.9610&rep=rep1&type=pdf
85: (4)                 .. [3] Sourav Medya et. al.:
86: (7)                    Group Centrality Maximization via Network Design.
87: (7)                    SIAM International Conference on Data Mining, SDM 2018, 126–134.
88: (7)                    https://sites.cs.ucsb.edu/~arlei/pubs/sdm18.pdf
89: (4)                 .. [4] Rami Puzis, Yuval Elovici, and Shlomi Dolev.
90: (7)                    "Fast algorithm for successive computation of group betweenness
centrality."
91: (7)                    https://journals.aps.org/pre/pdf/10.1103/PhysRevE.76.056709
92: (4)                 """
93: (4)                 GBC = []  # initialize betweenness
94: (4)                 list_of_groups = True
95: (4)                 if any(el in G for el in C):
96: (8)                     C = [C]
97: (8)                     list_of_groups = False
98: (4)                 set_v = {node for group in C for node in group}
99: (4)                 if set_v - G.nodes:  # element(s) of C not in G
100: (8)                    raise nx.NodeNotFound(f"The node(s) {set_v - G.nodes} are in C but not
in G.")
101: (4)                PB, sigma, D = _group_preprocessing(G, set_v, weight)
102: (4)                for group in C:
103: (8)                    group = set(group)  # set of nodes in group
104: (8)                    GBC_group = 0
105: (8)                    sigma_m = deepcopy(sigma)
106: (8)                    PB_m = deepcopy(PB)
107: (8)                    sigma_m_v = deepcopy(sigma_m)
108: (8)                    PB_m_v = deepcopy(PB_m)
109: (8)                    for v in group:
110: (12)                       GBC_group += PB_m[v][v]
111: (12)                       for x in group:
112: (16)                           for y in group:
113: (20)                               dxvy = 0
114: (20)                               dxyv = 0
115: (20)                               dvxy = 0
```

```
116: (20)                                if not (
117: (24)                                    sigma_m[x][y] == 0 or sigma_m[x][v] == 0 or sigma_m[v]
[y] == 0
118: (20)                                ):
119: (24)                                    if D[x][v] == D[x][y] + D[y][v]:
120: (28)                                        dxyv = sigma_m[x][y] * sigma_m[y][v] / sigma_m[x]
[v]
121: (24)                                    if D[x][y] == D[x][v] + D[v][y]:
122: (28)                                        dxvy = sigma_m[x][v] * sigma_m[v][y] / sigma_m[x]
[y]
123: (24)                                    if D[v][y] == D[v][x] + D[x][y]:
124: (28)                                        dvxy = sigma_m[v][x] * sigma[x][y] / sigma[v][y]
125: (20)                                sigma_m_v[x][y] = sigma_m[x][y] * (1 - dxvy)
126: (20)                                PB_m_v[x][y] = PB_m[x][y] - PB_m[x][y] * dxvy
127: (20)                                if y != v:
128: (24)                                    PB_m_v[x][y] -= PB_m[x][v] * dxyv
129: (20)                                if x != v:
130: (24)                                    PB_m_v[x][y] -= PB_m[v][y] * dvxy
131: (12)                        sigma_m, sigma_m_v = sigma_m_v, sigma_m
132: (12)                        PB_m, PB_m_v = PB_m_v, PB_m
133: (8)                    v, c = len(G), len(group)
134: (8)                    if not endpoints:
135: (12)                        scale = 0
136: (12)                        if nx.is_directed(G):
137: (16)                            if nx.is_strongly_connected(G):
138: (20)                                scale = c * (2 * v - c - 1)
139: (12)                        elif nx.is_connected(G):
140: (16)                            scale = c * (2 * v - c - 1)
141: (12)                        if scale == 0:
142: (16)                            for group_node1 in group:
143: (20)                                for node in D[group_node1]:
144: (24)                                    if node != group_node1:
145: (28)                                        if node in group:
146: (32)                                            scale += 1
147: (28)                                        else:
148: (32)                                            scale += 2
149: (12)                        GBC_group -= scale
150: (8)                    if normalized:
151: (12)                        scale = 1 / ((v - c) * (v - c - 1))
152: (12)                        GBC_group *= scale
153: (8)                    elif not G.is_directed():
154: (12)                        GBC_group /= 2
155: (8)                    GBC.append(GBC_group)
156: (4)                if list_of_groups:
157: (8)                    return GBC
158: (4)                return GBC[0]
159: (0)            def _group_preprocessing(G, set_v, weight):
160: (4)                sigma = {}
161: (4)                delta = {}
162: (4)                D = {}
163: (4)                betweenness = dict.fromkeys(G, 0)
164: (4)                for s in G:
165: (8)                    if weight is None:  # use BFS
166: (12)                       S, P, sigma[s], D[s] = _single_source_shortest_path_basic(G, s)
167: (8)                    else:  # use Dijkstra's algorithm
168: (12)                       S, P, sigma[s], D[s] = _single_source_dijkstra_path_basic(G, s,
weight)
169: (8)                    betweenness, delta[s] = _accumulate_endpoints(betweenness, S, P,
sigma[s], s)
170: (8)                    for i in delta[s]:  # add the paths from s to i and rescale sigma
171: (12)                       if s != i:
172: (16)                           delta[s][i] += 1
173: (12)                       if weight is not None:
174: (16)                           sigma[s][i] = sigma[s][i] / 2
175: (4)                PB = dict.fromkeys(G)
176: (4)                for group_node1 in set_v:
177: (8)                    PB[group_node1] = dict.fromkeys(G, 0.0)
178: (8)                    for group_node2 in set_v:
179: (12)                       if group_node2 not in D[group_node1]:
```

```
(16)                            continue
(12)                        for node in G:
(16)                            if group_node2 in D[node] and group_node1 in D[node]:
(20)                                if (
(24)                                    D[node][group_node2]
(24)                                    == D[node][group_node1] + D[group_node1][group_node2]
(20)                                ):
(24)                                    PB[group_node1][group_node2] += (
(28)                                        delta[node][group_node2]
(28)                                        * sigma[node][group_node1]
(28)                                        * sigma[group_node1][group_node2]
(28)                                        / sigma[node][group_node2]
(24)                                    )
(4)                 return PB, sigma, D
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def prominent_group(
(4)                 G, k, weight=None, C=None, endpoints=False, normalized=True, greedy=False
(0)             ):
(4)                 r"""Find the prominent group of size $k$ in graph $G$. The prominence of
the
(4)                 group is evaluated by the group betweenness centrality.
(4)                 Group betweenness centrality of a group of nodes $C$ is the sum of the
(4)                 fraction of all-pairs shortest paths that pass through any vertex in $C$
(4)                 .. math::
(7)                    c_B(v) =\sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}
(4)                 where $V$ is the set of nodes, $\sigma(s, t)$ is the number of
(4)                 shortest $(s, t)$-paths, and $\sigma(s, t|C)$ is the number of
(4)                 those paths passing through some node in group $C$. Note that
(4)                 $(s, t)$ are not members of the group ($V-C$ is the set of nodes
(4)                 in $V$ that are not in $C$).
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph.
(4)                 k : int
(7)                    The number of nodes in the group.
(4)                 normalized : bool, optional (default=True)
(7)                    If True, group betweenness is normalized by ``1/((|V|-|C|)(|V|-
|C|-1))``
(7)                    where ``|V|`` is the number of nodes in G and ``|C|`` is the number of
(7)                    nodes in C.
(4)                 weight : None or string, optional (default=None)
(7)                    If None, all edge weights are considered equal.
(7)                    Otherwise holds the name of the edge attribute used as weight.
(7)                    The weight of an edge is treated as the length or distance between the
two sides.
(4)                 endpoints : bool, optional (default=False)
(7)                    If True include the endpoints in the shortest path counts.
(4)                 C : list or set, optional (default=None)
(7)                    list of nodes which won't be candidates of the prominent group.
(4)                 greedy : bool, optional (default=False)
(7)                    Using a naive greedy algorithm in order to find non-optimal prominent
(7)                    group. For scale free networks the results are negligibly below the
optimal
(7)                    results.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(7)                    If node(s) in C are not present in G.
(4)                 Returns
(4)                 -------
(4)                 max_GBC : float
(7)                    The group betweenness centrality of the prominent group.
(4)                 max_group : list
(8)                     The list of nodes in the prominent group.
(4)                 See Also
(4)                 --------
(4)                 betweenness_centrality, group_betweenness_centrality
(4)                 Notes
```

```
(4)                 -----
(4)                 Group betweenness centrality is described in [1]_ and its importance discussed in [3]_.
(4)                 The algorithm is described in [2]_ and is based on techniques mentioned in [4]_.
(4)                 The number of nodes in the group must be a maximum of ``n - 2`` where ``n``
(4)                 is the total number of nodes in the graph.
(4)                 For weighted graphs the edge weights must be greater than zero.
(4)                 Zero edge weights can produce an infinite number of equal length
(4)                 paths between pairs of nodes.
(4)                 The total number of paths between source and target is counted
(4)                 differently for directed and undirected graphs. Directed paths
(4)                 between "u" and "v" are counted as two possible paths (one each
(4)                 direction) while undirected paths between "u" and "v" are counted
(4)                 as one path. Said another way, the sum in the expression above is
(4)                 over all ``s != t`` for directed graphs and for ``s < t`` for undirected graphs.
(4)                 References
(4)                 ----------
(4)                 .. [1] M G Everett and S P Borgatti:
(7)                    The Centrality of Groups and Classes.
(7)                    Journal of Mathematical Sociology. 23(3): 181-201. 1999.
(7)                    http://www.analytictech.com/borgatti/group_centrality.htm
(4)                 .. [2] Rami Puzis, Yuval Elovici, and Shlomi Dolev:
(7)                    "Finding the Most Prominent Group in Complex Networks"
(7)                    AI communications 20(4): 287-296, 2007.
(7)                    https://www.researchgate.net/profile/Rami_Puzis2/publication/220308855
(4)                 .. [3] Sourav Medya et. al.:
(7)                    Group Centrality Maximization via Network Design.
(7)                    SIAM International Conference on Data Mining, SDM 2018, 126–134.
(7)                    https://sites.cs.ucsb.edu/~arlei/pubs/sdm18.pdf
(4)                 .. [4] Rami Puzis, Yuval Elovici, and Shlomi Dolev.
(7)                    "Fast algorithm for successive computation of group betweenness centrality."
(7)                    https://journals.aps.org/pre/pdf/10.1103/PhysRevE.76.056709
(4)                 """
(4)                 import numpy as np
(4)                 import pandas as pd
(4)                 if C is not None:
(8)                     C = set(C)
(8)                     if C - G.nodes:  # element(s) of C not in G
(12)                        raise nx.NodeNotFound(f"The node(s) {C - G.nodes} are in C but not in G.")
(8)                     nodes = list(G.nodes - C)
(4)                 else:
(8)                     nodes = list(G.nodes)
(4)                 DF_tree = nx.Graph()
(4)                 DF_tree.__networkx_cache__ = None  # Disable caching
(4)                 PB, sigma, D = _group_preprocessing(G, nodes, weight)
(4)                 betweenness = pd.DataFrame.from_dict(PB)
(4)                 if C is not None:
(8)                     for node in C:
(12)                        betweenness.drop(index=node, inplace=True)
(12)                        betweenness.drop(columns=node, inplace=True)
(4)                 CL = [node for _, node in sorted(zip(np.diag(betweenness), nodes), reverse=True)]
(4)                 max_GBC = 0
(4)                 max_group = []
(4)                 DF_tree.add_node(
(8)                     1,
(8)                     CL=CL,
(8)                     betweenness=betweenness,
(8)                     GBC=0,
(8)                     GM=[],
(8)                     sigma=sigma,
(8)                     cont=dict(zip(nodes, np.diag(betweenness))),
(4)                 )
(4)                 DF_tree.nodes[1]["heu"] = 0
```

```
307: (4)                for i in range(k):
308: (8)                    DF_tree.nodes[1]["heu"] += DF_tree.nodes[1]["cont"][DF_tree.nodes[1]
["CL"][i]]
309: (4)                max_GBC, DF_tree, max_group = _dfbnb(
310: (8)                    G, k, DF_tree, max_GBC, 1, D, max_group, nodes, greedy
311: (4)                )
312: (4)                v = len(G)
313: (4)                if not endpoints:
314: (8)                    scale = 0
315: (8)                    if nx.is_directed(G):
316: (12)                       if nx.is_strongly_connected(G):
317: (16)                           scale = k * (2 * v - k - 1)
318: (8)                    elif nx.is_connected(G):
319: (12)                       scale = k * (2 * v - k - 1)
320: (8)                    if scale == 0:
321: (12)                       for group_node1 in max_group:
322: (16)                           for node in D[group_node1]:
323: (20)                               if node != group_node1:
324: (24)                                   if node in max_group:
325: (28)                                       scale += 1
326: (24)                                   else:
327: (28)                                       scale += 2
328: (8)                    max_GBC -= scale
329: (4)                if normalized:
330: (8)                    scale = 1 / ((v - k) * (v - k - 1))
331: (8)                    max_GBC *= scale
332: (4)                elif not G.is_directed():
333: (8)                    max_GBC /= 2
334: (4)                max_GBC = float("%.2f" % max_GBC)
335: (4)                return max_GBC, max_group
336: (0)            def _dfbnb(G, k, DF_tree, max_GBC, root, D, max_group, nodes, greedy):
337: (4)                if len(DF_tree.nodes[root]["GM"]) == k and DF_tree.nodes[root]["GBC"] >
max_GBC:
338: (8)                    return DF_tree.nodes[root]["GBC"], DF_tree, DF_tree.nodes[root]["GM"]
339: (4)                if (
340: (8)                    len(DF_tree.nodes[root]["GM"]) == k
341: (8)                    or len(DF_tree.nodes[root]["CL"]) <= k - len(DF_tree.nodes[root]
["GM"])
342: (8)                    or DF_tree.nodes[root]["GBC"] + DF_tree.nodes[root]["heu"] <= max_GBC
343: (4)                ):
344: (8)                    return max_GBC, DF_tree, max_group
345: (4)                node_p, node_m, DF_tree = _heuristic(k, root, DF_tree, D, nodes, greedy)
346: (4)                if greedy:
347: (8)                    max_GBC, DF_tree, max_group = _dfbnb(
348: (12)                       G, k, DF_tree, max_GBC, node_p, D, max_group, nodes, greedy
349: (8)                    )
350: (4)                elif (
351: (8)                    DF_tree.nodes[node_p]["GBC"] + DF_tree.nodes[node_p]["heu"]
352: (8)                    > DF_tree.nodes[node_m]["GBC"] + DF_tree.nodes[node_m]["heu"]
353: (4)                ):
354: (8)                    max_GBC, DF_tree, max_group = _dfbnb(
355: (12)                       G, k, DF_tree, max_GBC, node_p, D, max_group, nodes, greedy
356: (8)                    )
357: (8)                    max_GBC, DF_tree, max_group = _dfbnb(
358: (12)                       G, k, DF_tree, max_GBC, node_m, D, max_group, nodes, greedy
359: (8)                    )
360: (4)                else:
361: (8)                    max_GBC, DF_tree, max_group = _dfbnb(
362: (12)                       G, k, DF_tree, max_GBC, node_m, D, max_group, nodes, greedy
363: (8)                    )
364: (8)                    max_GBC, DF_tree, max_group = _dfbnb(
365: (12)                       G, k, DF_tree, max_GBC, node_p, D, max_group, nodes, greedy
366: (8)                    )
367: (4)                return max_GBC, DF_tree, max_group
368: (0)            def _heuristic(k, root, DF_tree, D, nodes, greedy):
369: (4)                import numpy as np
370: (4)                node_p = DF_tree.number_of_nodes() + 1
371: (4)                node_m = DF_tree.number_of_nodes() + 2
372: (4)                added_node = DF_tree.nodes[root]["CL"][0]
```

```
(4)                 DF_tree.add_nodes_from([(node_p, deepcopy(DF_tree.nodes[root]))])
(4)                 DF_tree.nodes[node_p]["GM"].append(added_node)
(4)                 DF_tree.nodes[node_p]["GBC"] += DF_tree.nodes[node_p]["cont"][added_node]
(4)                 root_node = DF_tree.nodes[root]
(4)                 for x in nodes:
(8)                     for y in nodes:
(12)                        dxvy = 0
(12)                        dxyv = 0
(12)                        dvxy = 0
(12)                        if not (
(16)                            root_node["sigma"][x][y] == 0
(16)                            or root_node["sigma"][x][added_node] == 0
(16)                            or root_node["sigma"][added_node][y] == 0
(12)                        ):
(16)                            if D[x][added_node] == D[x][y] + D[y][added_node]:
(20)                                dxyv = (
(24)                                    root_node["sigma"][x][y]
(24)                                    * root_node["sigma"][y][added_node]
(24)                                    / root_node["sigma"][x][added_node]
(20)                                )
(16)                            if D[x][y] == D[x][added_node] + D[added_node][y]:
(20)                                dxvy = (
(24)                                    root_node["sigma"][x][added_node]
(24)                                    * root_node["sigma"][added_node][y]
(24)                                    / root_node["sigma"][x][y]
(20)                                )
(16)                            if D[added_node][y] == D[added_node][x] + D[x][y]:
(20)                                dvxy = (
(24)                                    root_node["sigma"][added_node][x]
(24)                                    * root_node["sigma"][x][y]
(24)                                    / root_node["sigma"][added_node][y]
(20)                                )
(12)                        DF_tree.nodes[node_p]["sigma"][x][y] = root_node["sigma"][x][y] *
(1 - dxvy)
(12)                        DF_tree.nodes[node_p]["betweenness"].loc[y, x] = (
(16)                            root_node["betweenness"][x][y] - root_node["betweenness"][x]
[y] * dxvy
(12)                        )
(12)                        if y != added_node:
(16)                            DF_tree.nodes[node_p]["betweenness"].loc[y, x] -= (
(20)                                root_node["betweenness"][x][added_node] * dxyv
(16)                            )
(12)                        if x != added_node:
(16)                            DF_tree.nodes[node_p]["betweenness"].loc[y, x] -= (
(20)                                root_node["betweenness"][added_node][y] * dvxy
(16)                            )
(4)                 DF_tree.nodes[node_p]["CL"] = [
(8)                     node
(8)                     for _, node in sorted(
(12)                        zip(np.diag(DF_tree.nodes[node_p]["betweenness"]), nodes),
reverse=True
(8)                     )
(8)                     if node not in DF_tree.nodes[node_p]["GM"]
(4)                 ]
(4)                 DF_tree.nodes[node_p]["cont"] = dict(
(8)                     zip(nodes, np.diag(DF_tree.nodes[node_p]["betweenness"]))
(4)                 )
(4)                 DF_tree.nodes[node_p]["heu"] = 0
(4)                 for i in range(k - len(DF_tree.nodes[node_p]["GM"])):
(8)                     DF_tree.nodes[node_p]["heu"] += DF_tree.nodes[node_p]["cont"][
(12)                        DF_tree.nodes[node_p]["CL"][i]
(8)                     ]
(4)                 if not greedy:
(8)                     DF_tree.add_nodes_from([(node_m, deepcopy(DF_tree.nodes[root]))])
(8)                     DF_tree.nodes[node_m]["CL"].pop(0)
(8)                     DF_tree.nodes[node_m]["cont"].pop(added_node)
(8)                     DF_tree.nodes[node_m]["heu"] = 0
(8)                     for i in range(k - len(DF_tree.nodes[node_m]["GM"])):
(12)                        DF_tree.nodes[node_m]["heu"] += DF_tree.nodes[node_m]["cont"][
```

```
(16)                            DF_tree.nodes[node_m]["CL"][i]
(12)                        ]
(4)                 else:
(8)                     node_m = None
(4)                 return node_p, node_m, DF_tree
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def group_closeness_centrality(G, S, weight=None):
(4)                 r"""Compute the group closeness centrality for a group of nodes.
(4)                 Group closeness centrality of a group of nodes $S$ is a measure
(4)                 of how close the group is to the other nodes in the graph.
(4)                 .. math::
(7)                    c_{close}(S) = \frac{|V-S|}{\sum_{v \in V-S} d_{S, v}}
(7)                    d_{S, v} = min_{u \in S} (d_{u, v})
(4)                 where $V$ is the set of nodes, $d_{S, v}$ is the distance of
(4)                 the group $S$ from $v$ defined as above. ($V-S$ is the set of nodes
(4)                 in $V$ that are not in $S$).
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph.
(4)                 S : list or set
(7)                    S is a group of nodes which belong to G, for which group closeness
(7)                    centrality is to be calculated.
(4)                 weight : None or string, optional (default=None)
(7)                    If None, all edge weights are considered equal.
(7)                    Otherwise holds the name of the edge attribute used as weight.
(7)                    The weight of an edge is treated as the length or distance between the
two sides.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(7)                    If node(s) in S are not present in G.
(4)                 Returns
(4)                 -------
(4)                 closeness : float
(7)                    Group closeness centrality of the group S.
(4)                 See Also
(4)                 --------
(4)                 closeness_centrality
(4)                 Notes
(4)                 -----
(4)                 The measure was introduced in [1]_.
(4)                 The formula implemented here is described in [2]_.
(4)                 Higher values of closeness indicate greater centrality.
(4)                 It is assumed that 1 / 0 is 0 (required in the case of directed graphs,
(4)                 or when a shortest path length is 0).
(4)                 The number of nodes in the group must be a maximum of n - 1 where `n`
(4)                 is the total number of nodes in the graph.
(4)                 For directed graphs, the incoming distance is utilized here. To use the
(4)                 outward distance, act on `G.reverse()`.
(4)                 For weighted graphs the edge weights must be greater than zero.
(4)                 Zero edge weights can produce an infinite number of equal length
(4)                 paths between pairs of nodes.
(4)                 References
(4)                 ----------
(4)                 .. [1] M G Everett and S P Borgatti:
(7)                    The Centrality of Groups and Classes.
(7)                    Journal of Mathematical Sociology. 23(3): 181-201. 1999.
(7)                    http://www.analytictech.com/borgatti/group_centrality.htm
(4)                 .. [2] J. Zhao et. al.:
(7)                    Measuring and Maximizing Group Closeness Centrality over
(7)                    Disk Resident Graphs.
(7)                    WWWConference Proceedings, 2014. 689-694.
(7)                    https://doi.org/10.1145/2567948.2579356
(4)                 """
(4)                 if G.is_directed():
(8)                     G = G.reverse()  # reverse view
(4)                 closeness = 0  # initialize to 0
(4)                 V = set(G)  # set of nodes in G
```

```
(4)                 S = set(S)  # set of nodes in group S
(4)                 V_S = V - S  # set of nodes in V but not S
(4)                 shortest_path_lengths = nx.multi_source_dijkstra_path_length(G, S,
weight=weight)
(4)                 for v in V_S:
(8)                     try:
(12)                        closeness += shortest_path_lengths[v]
(8)                     except KeyError:  # no path exists
(12)                        closeness += 0
(4)                 try:
(8)                     closeness = len(V_S) / closeness
(4)                 except ZeroDivisionError:  # 1 / 0 assumed as 0
(8)                     closeness = 0
(4)                 return closeness
(0)             @nx._dispatchable
(0)             def group_degree_centrality(G, S):
(4)                 """Compute the group degree centrality for a group of nodes.
(4)                 Group degree centrality of a group of nodes $S$ is the fraction
(4)                 of non-group members connected to group members.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph.
(4)                 S : list or set
(7)                    S is a group of nodes which belong to G, for which group degree
(7)                    centrality is to be calculated.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(7)                    If node(s) in S are not in G.
(4)                 Returns
(4)                 -------
(4)                 centrality : float
(7)                    Group degree centrality of the group S.
(4)                 See Also
(4)                 --------
(4)                 degree_centrality
(4)                 group_in_degree_centrality
(4)                 group_out_degree_centrality
(4)                 Notes
(4)                 -----
(4)                 The measure was introduced in [1]_.
(4)                 The number of nodes in the group must be a maximum of n - 1 where `n`
(4)                 is the total number of nodes in the graph.
(4)                 References
(4)                 ----------
(4)                 .. [1] M G Everett and S P Borgatti:
(7)                    The Centrality of Groups and Classes.
(7)                    Journal of Mathematical Sociology. 23(3): 181-201. 1999.
(7)                    http://www.analytictech.com/borgatti/group_centrality.htm
(4)                 """
(4)                 centrality = len(set().union(*[set(G.neighbors(i)) for i in S]) - set(S))
(4)                 centrality /= len(G.nodes()) - len(S)
(4)                 return centrality
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable
(0)             def group_in_degree_centrality(G, S):
(4)                 """Compute the group in-degree centrality for a group of nodes.
(4)                 Group in-degree centrality of a group of nodes $S$ is the fraction
(4)                 of non-group members connected to group members by incoming edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph.
(4)                 S : list or set
(7)                    S is a group of nodes which belong to G, for which group in-degree
(7)                    centrality is to be calculated.
(4)                 Returns
(4)                 -------
```

```
575: (4)                centrality : float
576: (7)                   Group in-degree centrality of the group S.
577: (4)                Raises
578: (4)                ------
579: (4)                NetworkXNotImplemented
580: (7)                   If G is undirected.
581: (4)                NodeNotFound
582: (7)                   If node(s) in S are not in G.
583: (4)                See Also
584: (4)                --------
585: (4)                degree_centrality
586: (4)                group_degree_centrality
587: (4)                group_out_degree_centrality
588: (4)                Notes
589: (4)                -----
590: (4)                The number of nodes in the group must be a maximum of n - 1 where `n`
591: (4)                is the total number of nodes in the graph.
592: (4)                `G.neighbors(i)` gives nodes with an outward edge from i, in a DiGraph,
593: (4)                so for group in-degree centrality, the reverse graph is used.
594: (4)                """
595: (4)                return group_degree_centrality(G.reverse(), S)
596: (0)            @not_implemented_for("undirected")
597: (0)            @nx._dispatchable
598: (0)            def group_out_degree_centrality(G, S):
599: (4)                """Compute the group out-degree centrality for a group of nodes.
600: (4)                Group out-degree centrality of a group of nodes $S$ is the fraction
601: (4)                of non-group members connected to group members by outgoing edges.
602: (4)                Parameters
603: (4)                ----------
604: (4)                G : graph
605: (7)                   A NetworkX graph.
606: (4)                S : list or set
607: (7)                   S is a group of nodes which belong to G, for which group in-degree
608: (7)                   centrality is to be calculated.
609: (4)                Returns
610: (4)                -------
611: (4)                centrality : float
612: (7)                   Group out-degree centrality of the group S.
613: (4)                Raises
614: (4)                ------
615: (4)                NetworkXNotImplemented
616: (7)                   If G is undirected.
617: (4)                NodeNotFound
618: (7)                   If node(s) in S are not in G.
619: (4)                See Also
620: (4)                --------
621: (4)                degree_centrality
622: (4)                group_degree_centrality
623: (4)                group_in_degree_centrality
624: (4)                Notes
625: (4)                -----
626: (4)                The number of nodes in the group must be a maximum of n - 1 where `n`
627: (4)                is the total number of nodes in the graph.
628: (4)                `G.neighbors(i)` gives nodes with an outward edge from i, in a DiGraph,
629: (4)                so for group out-degree centrality, the graph itself is used.
630: (4)                """
631: (4)                return group_degree_centrality(G, S)

----------------------------------------

File 142 - flow_matrix.py:

1: (0)              import networkx as nx
2: (0)              @nx._dispatchable(edge_attrs="weight")
3: (0)              def flow_matrix_row(G, weight=None, dtype=float, solver="lu"):
4: (4)                  import numpy as np
5: (4)                  solvername = {
6: (8)                      "full": FullInverseLaplacian,
7: (8)                      "lu": SuperLUInverseLaplacian,
```

```
 8: (8)                     "cg": CGInverseLaplacian,
 9: (4)                 }
 10: (4)                n = G.number_of_nodes()
 11: (4)                L = nx.laplacian_matrix(G, nodelist=range(n),
weight=weight).asformat("csc")
 12: (4)                L = L.astype(dtype)
 13: (4)                C = solvername[solver](L, dtype=dtype)  # initialize solver
 14: (4)                w = C.w  # w is the Laplacian matrix width
 15: (4)                for u, v in sorted(sorted((u, v)) for u, v in G.edges()):
 16: (8)                    B = np.zeros(w, dtype=dtype)
 17: (8)                    c = G[u][v].get(weight, 1.0)
 18: (8)                    B[u % w] = c
 19: (8)                    B[v % w] = -c
 20: (8)                    row = B @ C.get_rows(u, v)
 21: (8)                    yield row, (u, v)
 22: (0)            class InverseLaplacian:
 23: (4)                def __init__(self, L, width=None, dtype=None):
 24: (8)                    global np
 25: (8)                    import numpy as np
 26: (8)                    (n, n) = L.shape
 27: (8)                    self.dtype = dtype
 28: (8)                    self.n = n
 29: (8)                    if width is None:
 30: (12)                       self.w = self.width(L)
 31: (8)                    else:
 32: (12)                       self.w = width
 33: (8)                    self.C = np.zeros((self.w, n), dtype=dtype)
 34: (8)                    self.L1 = L[1:, 1:]
 35: (8)                    self.init_solver(L)
 36: (4)                def init_solver(self, L):
 37: (8)                    pass
 38: (4)                def solve(self, r):
 39: (8)                    raise nx.NetworkXError("Implement solver")
 40: (4)                def solve_inverse(self, r):
 41: (8)                    raise nx.NetworkXError("Implement solver")
 42: (4)                def get_rows(self, r1, r2):
 43: (8)                    for r in range(r1, r2 + 1):
 44: (12)                       self.C[r % self.w, 1:] = self.solve_inverse(r)
 45: (8)                    return self.C
 46: (4)                def get_row(self, r):
 47: (8)                    self.C[r % self.w, 1:] = self.solve_inverse(r)
 48: (8)                    return self.C[r % self.w]
 49: (4)                def width(self, L):
 50: (8)                    m = 0
 51: (8)                    for i, row in enumerate(L):
 52: (12)                       w = 0
 53: (12)                       x, y = np.nonzero(row)
 54: (12)                       if len(y) > 0:
 55: (16)                           v = y - i
 56: (16)                           w = v.max() - v.min() + 1
 57: (16)                           m = max(w, m)
 58: (8)                    return m
 59: (0)            class FullInverseLaplacian(InverseLaplacian):
 60: (4)                def init_solver(self, L):
 61: (8)                    self.IL = np.zeros(L.shape, dtype=self.dtype)
 62: (8)                    self.IL[1:, 1:] = np.linalg.inv(self.L1.todense())
 63: (4)                def solve(self, rhs):
 64: (8)                    s = np.zeros(rhs.shape, dtype=self.dtype)
 65: (8)                    s = self.IL @ rhs
 66: (8)                    return s
 67: (4)                def solve_inverse(self, r):
 68: (8)                    return self.IL[r, 1:]
 69: (0)            class SuperLUInverseLaplacian(InverseLaplacian):
 70: (4)                def init_solver(self, L):
 71: (8)                    import scipy as sp
 72: (8)                    self.lusolve = sp.sparse.linalg.factorized(self.L1.tocsc())
 73: (4)                def solve_inverse(self, r):
 74: (8)                    rhs = np.zeros(self.n, dtype=self.dtype)
 75: (8)                    rhs[r] = 1
```

```
76: (8)                     return self.lusolve(rhs[1:])
77: (4)                 def solve(self, rhs):
78: (8)                     s = np.zeros(rhs.shape, dtype=self.dtype)
79: (8)                     s[1:] = self.lusolve(rhs[1:])
80: (8)                     return s
81: (0)             class CGInverseLaplacian(InverseLaplacian):
82: (4)                 def init_solver(self, L):
83: (8)                     global sp
84: (8)                     import scipy as sp
85: (8)                     ilu = sp.sparse.linalg.spilu(self.L1.tocsc())
86: (8)                     n = self.n - 1
87: (8)                     self.M = sp.sparse.linalg.LinearOperator(shape=(n, n),
matvec=ilu.solve)
88: (4)                 def solve(self, rhs):
89: (8)                     s = np.zeros(rhs.shape, dtype=self.dtype)
90: (8)                     s[1:] = sp.sparse.linalg.cg(self.L1, rhs[1:], M=self.M, atol=0)[0]
91: (8)                     return s
92: (4)                 def solve_inverse(self, r):
93: (8)                     rhs = np.zeros(self.n, self.dtype)
94: (8)                     rhs[r] = 1
95: (8)                     return sp.sparse.linalg.cg(self.L1, rhs[1:], M=self.M, atol=0)[0]
```

File 143 - katz.py:

```
1: (0)              """Katz centrality."""
2: (0)              import math
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["katz_centrality", "katz_centrality_numpy"]
6: (0)              @not_implemented_for("multigraph")
7: (0)              @nx._dispatchable(edge_attrs="weight")
8: (0)              def katz_centrality(
9: (4)                  G,
10: (4)                 alpha=0.1,
11: (4)                 beta=1.0,
12: (4)                 max_iter=1000,
13: (4)                 tol=1.0e-6,
14: (4)                 nstart=None,
15: (4)                 normalized=True,
16: (4)                 weight=None,
17: (0)             ):
18: (4)                 r"""Compute the Katz centrality for the nodes of the graph G.
19: (4)                 Katz centrality computes the centrality for a node based on the centrality
20: (4)                 of its neighbors. It is a generalization of the eigenvector centrality.
The
21: (4)                 Katz centrality for node $i$ is
22: (4)                 .. math::
23: (8)                     x_i = \alpha \sum_{j} A_{ij} x_j + \beta,
24: (4)                 where $A$ is the adjacency matrix of graph G with eigenvalues $\lambda$.
25: (4)                 The parameter $\beta$ controls the initial centrality and
26: (4)                 .. math::
27: (8)                     \alpha < \frac{1}{\lambda_{\max}}.
28: (4)                 Katz centrality computes the relative influence of a node within a
29: (4)                 network by measuring the number of the immediate neighbors (first
30: (4)                 degree nodes) and also all other nodes in the network that connect
31: (4)                 to the node under consideration through these immediate neighbors.
32: (4)                 Extra weight can be provided to immediate neighbors through the
33: (4)                 parameter $\beta$.  Connections made with distant neighbors
34: (4)                 are, however, penalized by an attenuation factor $\alpha$ which
35: (4)                 should be strictly less than the inverse largest eigenvalue of the
36: (4)                 adjacency matrix in order for the Katz centrality to be computed
37: (4)                 correctly. More information is provided in [1]_.
38: (4)                 Parameters
39: (4)                 ----------
40: (4)                 G : graph
41: (6)                   A NetworkX graph.
42: (4)                 alpha : float, optional (default=0.1)
```

```
43: (6)                 Attenuation factor
44: (4)               beta : scalar or dictionary, optional (default=1.0)
45: (6)                 Weight attributed to the immediate neighborhood. If not a scalar, the
46: (6)                 dictionary must have a value for every node.
47: (4)               max_iter : integer, optional (default=1000)
48: (6)                 Maximum number of iterations in power method.
49: (4)               tol : float, optional (default=1.0e-6)
50: (6)                 Error tolerance used to check convergence in power method iteration.
51: (4)               nstart : dictionary, optional
52: (6)                 Starting value of Katz iteration for each node.
53: (4)               normalized : bool, optional (default=True)
54: (6)                 If True normalize the resulting values.
55: (4)               weight : None or string, optional (default=None)
56: (6)                 If None, all edge weights are considered equal.
57: (6)                 Otherwise holds the name of the edge attribute used as weight.
58: (6)                 In this measure the weight is interpreted as the connection strength.
59: (4)               Returns
60: (4)               -------
61: (4)               nodes : dictionary
62: (7)                  Dictionary of nodes with Katz centrality as the value.
63: (4)               Raises
64: (4)               ------
65: (4)               NetworkXError
66: (7)                  If the parameter `beta` is not a scalar but lacks a value for at least
67: (7)                  one node
68: (4)               PowerIterationFailedConvergence
69: (8)                   If the algorithm fails to converge to the specified tolerance
70: (8)                   within the specified number of iterations of the power iteration
71: (8)                   method.
72: (4)               Examples
73: (4)               --------
74: (4)               >>> import math
75: (4)               >>> G = nx.path_graph(4)
76: (4)               >>> phi = (1 + math.sqrt(5)) / 2.0  # largest eigenvalue of adj matrix
77: (4)               >>> centrality = nx.katz_centrality(G, 1 / phi - 0.01)
78: (4)               >>> for n, c in sorted(centrality.items()):
79: (4)               ...     print(f"{n} {c:.2f}")
80: (4)               0 0.37
81: (4)               1 0.60
82: (4)               2 0.60
83: (4)               3 0.37
84: (4)               See Also
85: (4)               --------
86: (4)               katz_centrality_numpy
87: (4)               eigenvector_centrality
88: (4)               eigenvector_centrality_numpy
89: (4)               :func:`~networkx.algorithms.link_analysis.pagerank_alg.pagerank`
90: (4)               :func:`~networkx.algorithms.link_analysis.hits_alg.hits`
91: (4)               Notes
92: (4)               -----
93: (4)               Katz centrality was introduced by [2]_.
94: (4)               This algorithm it uses the power method to find the eigenvector
95: (4)               corresponding to the largest eigenvalue of the adjacency matrix of ``G``.
96: (4)               The parameter ``alpha`` should be strictly less than the inverse of
largest
97: (4)               eigenvalue of the adjacency matrix for the algorithm to converge.
98: (4)               You can use ``max(nx.adjacency_spectrum(G))`` to get $\lambda_{\max}$ the
largest
99: (4)               eigenvalue of the adjacency matrix.
100: (4)              The iteration will stop after ``max_iter`` iterations or an error
tolerance of
101: (4)              ``number_of_nodes(G) * tol`` has been reached.
102: (4)              For strongly connected graphs, as $\alpha \to 1/\lambda_{\max}$, and
$\beta > 0$,
103: (4)              Katz centrality approaches the results for eigenvector centrality.
104: (4)              For directed graphs this finds "left" eigenvectors which corresponds
105: (4)              to the in-edges in the graph. For out-edges Katz centrality,
106: (4)              first reverse the graph with ``G.reverse()``.
107: (4)              References
```

```
(4)                 ----------
(4)                 .. [1] Mark E. J. Newman:
(7)                    Networks: An Introduction.
(7)                    Oxford University Press, USA, 2010, p. 720.
(4)                 .. [2] Leo Katz:
(7)                    A New Status Index Derived from Sociometric Index.
(7)                    Psychometrika 18(1):39–43, 1953
(7)                    https://link.springer.com/content/pdf/10.1007/BF02289026.pdf
(4)                 """
(4)                 if len(G) == 0:
(8)                     return {}
(4)                 nnodes = G.number_of_nodes()
(4)                 if nstart is None:
(8)                     x = {n: 0 for n in G}
(4)                 else:
(8)                     x = nstart
(4)                 try:
(8)                     b = dict.fromkeys(G, float(beta))
(4)                 except (TypeError, ValueError, AttributeError) as err:
(8)                     b = beta
(8)                     if set(beta) != set(G):
(12)                        raise nx.NetworkXError(
(16)                            "beta dictionary must have a value for every node"
(12)                        ) from err
(4)                 for _ in range(max_iter):
(8)                     xlast = x
(8)                     x = dict.fromkeys(xlast, 0)
(8)                     for n in x:
(12)                        for nbr in G[n]:
(16)                            x[nbr] += xlast[n] * G[n][nbr].get(weight, 1)
(8)                     for n in x:
(12)                        x[n] = alpha * x[n] + b[n]
(8)                     error = sum(abs(x[n] - xlast[n]) for n in x)
(8)                     if error < nnodes * tol:
(12)                        if normalized:
(16)                            try:
(20)                                s = 1.0 / math.hypot(*x.values())
(16)                            except ZeroDivisionError:
(20)                                s = 1.0
(12)                        else:
(16)                            s = 1
(12)                        for n in x:
(16)                            x[n] *= s
(12)                        return x
(4)                 raise nx.PowerIterationFailedConvergence(max_iter)
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def katz_centrality_numpy(G, alpha=0.1, beta=1.0, normalized=True,
weight=None):
(4)                 r"""Compute the Katz centrality for the graph G.
(4)                 Katz centrality computes the centrality for a node based on the centrality
(4)                 of its neighbors. It is a generalization of the eigenvector centrality.
The
(4)                 Katz centrality for node $i$ is
(4)                 .. math::
(8)                     x_i = \alpha \sum_{j} A_{ij} x_j + \beta,
(4)                 where $A$ is the adjacency matrix of graph G with eigenvalues $\lambda$.
(4)                 The parameter $\beta$ controls the initial centrality and
(4)                 .. math::
(8)                     \alpha < \frac{1}{\lambda_{\max}}.
(4)                 Katz centrality computes the relative influence of a node within a
(4)                 network by measuring the number of the immediate neighbors (first
(4)                 degree nodes) and also all other nodes in the network that connect
(4)                 to the node under consideration through these immediate neighbors.
(4)                 Extra weight can be provided to immediate neighbors through the
(4)                 parameter $\beta$.  Connections made with distant neighbors
(4)                 are, however, penalized by an attenuation factor $\alpha$ which
(4)                 should be strictly less than the inverse largest eigenvalue of the
(4)                 adjacency matrix in order for the Katz centrality to be computed
```

```
(4)                 correctly. More information is provided in [1]_.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A NetworkX graph
(4)                 alpha : float
(6)                   Attenuation factor
(4)                 beta : scalar or dictionary, optional (default=1.0)
(6)                   Weight attributed to the immediate neighborhood. If not a scalar the
(6)                   dictionary must have an value for every node.
(4)                 normalized : bool
(6)                   If True normalize the resulting values.
(4)                 weight : None or string, optional
(6)                   If None, all edge weights are considered equal.
(6)                   Otherwise holds the name of the edge attribute used as weight.
(6)                   In this measure the weight is interpreted as the connection strength.
(4)                 Returns
(4)                 -------
(4)                 nodes : dictionary
(7)                    Dictionary of nodes with Katz centrality as the value.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(7)                    If the parameter `beta` is not a scalar but lacks a value for at least
(7)                    one node
(4)                 Examples
(4)                 --------
(4)                 >>> import math
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> phi = (1 + math.sqrt(5)) / 2.0  # largest eigenvalue of adj matrix
(4)                 >>> centrality = nx.katz_centrality_numpy(G, 1 / phi)
(4)                 >>> for n, c in sorted(centrality.items()):
(4)                 ...     print(f"{n} {c:.2f}")
(4)                 0 0.37
(4)                 1 0.60
(4)                 2 0.60
(4)                 3 0.37
(4)                 See Also
(4)                 --------
(4)                 katz_centrality
(4)                 eigenvector_centrality_numpy
(4)                 eigenvector_centrality
(4)                 :func:`~networkx.algorithms.link_analysis.pagerank_alg.pagerank`
(4)                 :func:`~networkx.algorithms.link_analysis.hits_alg.hits`
(4)                 Notes
(4)                 -----
(4)                 Katz centrality was introduced by [2]_.
(4)                 This algorithm uses a direct linear solver to solve the above equation.
(4)                 The parameter ``alpha`` should be strictly less than the inverse of
largest
(4)                 eigenvalue of the adjacency matrix for there to be a solution.
(4)                 You can use ``max(nx.adjacency_spectrum(G))`` to get $\lambda_{\max}$ the
largest
(4)                 eigenvalue of the adjacency matrix.
(4)                 For strongly connected graphs, as $\alpha \to 1/\lambda_{\max}$, and
$\beta > 0$,
(4)                 Katz centrality approaches the results for eigenvector centrality.
(4)                 For directed graphs this finds "left" eigenvectors which corresponds
(4)                 to the in-edges in the graph. For out-edges Katz centrality,
(4)                 first reverse the graph with ``G.reverse()``.
(4)                 References
(4)                 ----------
(4)                 .. [1] Mark E. J. Newman:
(7)                    Networks: An Introduction.
(7)                    Oxford University Press, USA, 2010, p. 173.
(4)                 .. [2] Leo Katz:
(7)                    A New Status Index Derived from Sociometric Index.
(7)                    Psychometrika 18(1):39–43, 1953
(7)                    https://link.springer.com/content/pdf/10.1007/BF02289026.pdf
```

```
241: (4)                """
242: (4)                import numpy as np
243: (4)                if len(G) == 0:
244: (8)                    return {}
245: (4)                try:
246: (8)                    nodelist = beta.keys()
247: (8)                    if set(nodelist) != set(G):
248: (12)                       raise nx.NetworkXError("beta dictionary must have a value for
every node")
249: (8)                    b = np.array(list(beta.values()), dtype=float)
250: (4)                except AttributeError:
251: (8)                    nodelist = list(G)
252: (8)                    try:
253: (12)                       b = np.ones((len(nodelist), 1)) * beta
254: (8)                    except (TypeError, ValueError, AttributeError) as err:
255: (12)                       raise nx.NetworkXError("beta must be a number") from err
256: (4)                A = nx.adjacency_matrix(G, nodelist=nodelist, weight=weight).todense().T
257: (4)                n = A.shape[0]
258: (4)                centrality = np.linalg.solve(np.eye(n, n) - (alpha * A), b).squeeze()
259: (4)                norm = np.sign(sum(centrality)) * np.linalg.norm(centrality) if normalized
else 1
260: (4)                return dict(zip(nodelist, (centrality / norm).tolist()))

----------------------------------------

File 144 - harmonic.py:

1: (0)              """Functions for computing the harmonic centrality of a graph."""
2: (0)              from functools import partial
3: (0)              import networkx as nx
4: (0)              __all__ = ["harmonic_centrality"]
5: (0)              @nx._dispatchable(edge_attrs="distance")
6: (0)              def harmonic_centrality(G, nbunch=None, distance=None, sources=None):
7: (4)                  r"""Compute harmonic centrality for nodes.
8: (4)                  Harmonic centrality [1]_ of a node `u` is the sum of the reciprocal
9: (4)                  of the shortest path distances from all other nodes to `u`
10: (4)                 .. math::
11: (8)                     C(u) = \sum_{v \neq u} \frac{1}{d(v, u)}
12: (4)                 where `d(v, u)` is the shortest-path distance between `v` and `u`.
13: (4)                 If `sources` is given as an argument, the returned harmonic centrality
14: (4)                 values are calculated as the sum of the reciprocals of the shortest
15: (4)                 path distances from the nodes specified in `sources` to `u` instead
16: (4)                 of from all nodes to `u`.
17: (4)                 Notice that higher values indicate higher centrality.
18: (4)                 Parameters
19: (4)                 ----------
20: (4)                 G : graph
21: (6)                   A NetworkX graph
22: (4)                 nbunch : container (default: all nodes in G)
23: (6)                   Container of nodes for which harmonic centrality values are calculated.
24: (4)                 sources : container (default: all nodes in G)
25: (6)                   Container of nodes `v` over which reciprocal distances are computed.
26: (6)                   Nodes not in `G` are silently ignored.
27: (4)                 distance : edge attribute key, optional (default=None)
28: (6)                   Use the specified edge attribute as the edge distance in shortest
29: (6)                   path calculations.  If `None`, then each edge will have distance equal
to 1.
30: (4)                 Returns
31: (4)                 -------
32: (4)                 nodes : dictionary
33: (6)                   Dictionary of nodes with harmonic centrality as the value.
34: (4)                 See Also
35: (4)                 --------
36: (4)                 betweenness_centrality, load_centrality, eigenvector_centrality,
37: (4)                 degree_centrality, closeness_centrality
38: (4)                 Notes
39: (4)                 -----
40: (4)                 If the 'distance' keyword is set to an edge attribute key then the
41: (4)                 shortest-path length will be computed using Dijkstra's algorithm with
```

```
42: (4)                 that edge attribute as the edge weight.
43: (4)                 References
44: (4)                 ----------
45: (4)                 .. [1] Boldi, Paolo, and Sebastiano Vigna. "Axioms for centrality."
46: (11)                       Internet Mathematics 10.3-4 (2014): 222-262.
47: (4)                 """
48: (4)                 nbunch = set(G.nbunch_iter(nbunch)) if nbunch is not None else
set(G.nodes)
49: (4)                 sources = set(G.nbunch_iter(sources)) if sources is not None else G.nodes
50: (4)                 spl = partial(nx.shortest_path_length, G, weight=distance)
51: (4)                 centrality = {u: 0 for u in nbunch}
52: (4)                 for v in sources:
53: (8)                     dist = spl(v)
54: (8)                     for u in nbunch.intersection(dist):
55: (12)                        d = dist[u]
56: (12)                        if d == 0:  # handle u == v and edges with 0 weight
57: (16)                            continue
58: (12)                        centrality[u] += 1 / d
59: (4)                 return centrality

----------------------------------------

File 145 - load.py:

1: (0)              """Load centrality."""
2: (0)              from operator import itemgetter
3: (0)              import networkx as nx
4: (0)              __all__ = ["load_centrality", "edge_load_centrality"]
5: (0)              @nx._dispatchable(edge_attrs="weight")
6: (0)              def newman_betweenness_centrality(G, v=None, cutoff=None, normalized=True,
weight=None):
7: (4)                  """Compute load centrality for nodes.
8: (4)                  The load centrality of a node is the fraction of all shortest
9: (4)                  paths that pass through that node.
10: (4)                 Parameters
11: (4)                 ----------
12: (4)                 G : graph
13: (6)                   A networkx graph.
14: (4)                 normalized : bool, optional (default=True)
15: (6)                   If True the betweenness values are normalized by b=b/(n-1)(n-2) where
16: (6)                   n is the number of nodes in G.
17: (4)                 weight : None or string, optional (default=None)
18: (6)                   If None, edge weights are ignored.
19: (6)                   Otherwise holds the name of the edge attribute used as weight.
20: (6)                   The weight of an edge is treated as the length or distance between the
two sides.
21: (4)                 cutoff : bool, optional (default=None)
22: (6)                   If specified, only consider paths of length <= cutoff.
23: (4)                 Returns
24: (4)                 -------
25: (4)                 nodes : dictionary
26: (7)                    Dictionary of nodes with centrality as the value.
27: (4)                 See Also
28: (4)                 --------
29: (4)                 betweenness_centrality
30: (4)                 Notes
31: (4)                 -----
32: (4)                 Load centrality is slightly different than betweenness. It was originally
33: (4)                 introduced by [2]_. For this load algorithm see [1]_.
34: (4)                 References
35: (4)                 ----------
36: (4)                 .. [1] Mark E. J. Newman:
37: (7)                    Scientific collaboration networks. II.
38: (7)                    Shortest paths, weighted networks, and centrality.
39: (7)                    Physical Review E 64, 016132, 2001.
40: (7)                    http://journals.aps.org/pre/abstract/10.1103/PhysRevE.64.016132
41: (4)                 .. [2] Kwang-Il Goh, Byungnam Kahng and Doochul Kim
42: (7)                    Universal behavior of Load Distribution in Scale-Free Networks.
43: (7)                    Physical Review Letters 87(27):1–4, 2001.
```

```
 44: (7)                    https://doi.org/10.1103/PhysRevLett.87.278701
 45: (4)                 """
 46: (4)                 if v is not None:  # only one node
 47: (8)                     betweenness = 0.0
 48: (8)                     for source in G:
 49: (12)                        ubetween = _node_betweenness(G, source, cutoff, False, weight)
 50: (12)                        betweenness += ubetween[v] if v in ubetween else 0
 51: (8)                     if normalized:
 52: (12)                        order = G.order()
 53: (12)                        if order <= 2:
 54: (16)                            return betweenness  # no normalization b=0 for all nodes
 55: (12)                        betweenness *= 1.0 / ((order - 1) * (order - 2))
 56: (4)                 else:
 57: (8)                     betweenness = {}.fromkeys(G, 0.0)
 58: (8)                     for source in betweenness:
 59: (12)                        ubetween = _node_betweenness(G, source, cutoff, False, weight)
 60: (12)                        for vk in ubetween:
 61: (16)                            betweenness[vk] += ubetween[vk]
 62: (8)                     if normalized:
 63: (12)                        order = G.order()
 64: (12)                        if order <= 2:
 65: (16)                            return betweenness  # no normalization b=0 for all nodes
 66: (12)                        scale = 1.0 / ((order - 1) * (order - 2))
 67: (12)                        for v in betweenness:
 68: (16)                            betweenness[v] *= scale
 69: (4)                 return betweenness  # all nodes
 70: (0)             def _node_betweenness(G, source, cutoff=False, normalized=True, weight=None):
 71: (4)                 """Node betweenness_centrality helper:
 72: (4)                 See betweenness_centrality for what you probably want.
 73: (4)                 This actually computes "load" and not betweenness.
 74: (4)                 See https://networkx.lanl.gov/ticket/103
 75: (4)                 This calculates the load of each node for paths from a single source.
 76: (4)                 (The fraction of number of shortests paths from source that go
 77: (4)                 through each node.)
 78: (4)                 To get the load for a node you need to do all-pairs shortest paths.
 79: (4)                 If weight is not None then use Dijkstra for finding shortest paths.
 80: (4)                 """
 81: (4)                 if weight is None:
 82: (8)                     (pred, length) = nx.predecessor(G, source, cutoff=cutoff,
return_seen=True)
 83: (4)                 else:
 84: (8)                     (pred, length) = nx.dijkstra_predecessor_and_distance(G, source,
cutoff, weight)
 85: (4)                 onodes = [(l, vert) for (vert, l) in length.items()]
 86: (4)                 onodes.sort()
 87: (4)                 onodes[:] = [vert for (l, vert) in onodes if l > 0]
 88: (4)                 between = {}.fromkeys(length, 1.0)
 89: (4)                 while onodes:
 90: (8)                     v = onodes.pop()
 91: (8)                     if v in pred:
 92: (12)                        num_paths = len(pred[v])  # Discount betweenness if more than
 93: (12)                        for x in pred[v]:  # one shortest path.
 94: (16)                            if x == source:  # stop if hit source because all remaining v
 95: (20)                                break  # also have pred[v]==[source]
 96: (16)                            between[x] += between[v] / num_paths
 97: (4)                 for v in between:
 98: (8)                     between[v] -= 1
 99: (4)                 if normalized:
 100: (8)                    l = len(between)
 101: (8)                    if l > 2:
 102: (12)                       scale = 1 / ((l - 1) * (l - 2))
 103: (12)                       for v in between:
 104: (16)                           between[v] *= scale
 105: (4)                return between
 106: (0)            load_centrality = newman_betweenness_centrality
 107: (0)            @nx._dispatchable
 108: (0)            def edge_load_centrality(G, cutoff=False):
 109: (4)                """Compute edge load.
 110: (4)                WARNING: This concept of edge load has not been analysed
```

```
111: (4)                or discussed outside of NetworkX that we know of.
112: (4)                It is based loosely on load_centrality in the sense that
113: (4)                it counts the number of shortest paths which cross each edge.
114: (4)                This function is for demonstration and testing purposes.
115: (4)                Parameters
116: (4)                ----------
117: (4)                G : graph
118: (8)                    A networkx graph
119: (4)                cutoff : bool, optional (default=False)
120: (8)                    If specified, only consider paths of length <= cutoff.
121: (4)                Returns
122: (4)                -------
123: (4)                A dict keyed by edge 2-tuple to the number of shortest paths
124: (4)                which use that edge. Where more than one path is shortest
125: (4)                the count is divided equally among paths.
126: (4)                """
127: (4)                betweenness = {}
128: (4)                for u, v in G.edges():
129: (8)                    betweenness[(u, v)] = 0.0
130: (8)                    betweenness[(v, u)] = 0.0
131: (4)                for source in G:
132: (8)                    ubetween = _edge_betweenness(G, source, cutoff=cutoff)
133: (8)                    for e, ubetweenv in ubetween.items():
134: (12)                       betweenness[e] += ubetweenv  # cumulative total
135: (4)                return betweenness
136: (0)            def _edge_betweenness(G, source, nodes=None, cutoff=False):
137: (4)                """Edge betweenness helper."""
138: (4)                (pred, length) = nx.predecessor(G, source, cutoff=cutoff,
return_seen=True)
139: (4)                onodes = [n for n, d in sorted(length.items(), key=itemgetter(1))]
140: (4)                between = {}
141: (4)                for u, v in G.edges(nodes):
142: (8)                    between[(u, v)] = 1.0
143: (8)                    between[(v, u)] = 1.0
144: (4)                while onodes:  # work through all paths
145: (8)                    v = onodes.pop()
146: (8)                    if v in pred:
147: (12)                       num_paths = len(pred[v])
148: (12)                       for w in pred[v]:
149: (16)                           if w in pred:
150: (20)                               num_paths = len(pred[w])
151: (20)                               for x in pred[w]:
152: (24)                                   between[(w, x)] += between[(v, w)] / num_paths
153: (24)                                   between[(x, w)] += between[(w, v)] / num_paths
154: (4)                return between

----------------------------------------

File 146 - laplacian.py:

1: (0)              """
2: (0)              Laplacian centrality measures.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = ["laplacian_centrality"]
6: (0)              @nx._dispatchable(edge_attrs="weight")
7: (0)              def laplacian_centrality(
8: (4)                  G, normalized=True, nodelist=None, weight="weight", walk_type=None,
alpha=0.95
9: (0)              ):
10: (4)                 r"""Compute the Laplacian centrality for nodes in the graph `G`.
11: (4)                 The Laplacian Centrality of a node ``i`` is measured by the drop in the
12: (4)                 Laplacian Energy after deleting node ``i`` from the graph. The Laplacian
Energy
13: (4)                 is the sum of the squared eigenvalues of a graph's Laplacian matrix.
14: (4)                 .. math::
15: (8)                     C_L(u_i,G) = \frac{(\Delta E)_i}{E_L (G)} = \frac{E_L (G)-E_L (G_i)}
{E_L (G)}
16: (8)                     E_L (G) = \sum_{i=0}^n \lambda_i^2
```

```
Where $E_L (G)$ is the Laplacian energy of graph `G`,
E_L (G_i) is the Laplacian energy of graph `G` after deleting node ``i``
and $\lambda_i$ are the eigenvalues of `G`'s Laplacian matrix.
This formula shows the normalized value. Without normalization,
the numerator on the right side is returned.
Parameters
----------
G : graph
    A networkx graph
normalized : bool (default = True)
    If True the centrality score is scaled so the sum over all nodes is 1.
    If False the centrality score for each node is the drop in Laplacian
    energy when that node is removed.
nodelist : list, optional (default = None)
    The rows and columns are ordered according to the nodes in nodelist.
    If nodelist is None, then the ordering is produced by G.nodes().
weight: string or None, optional (default=`weight`)
    Optional parameter `weight` to compute the Laplacian matrix.
    The edge data key used to compute each value in the matrix.
    If None, then each edge has weight 1.
walk_type : string or None, optional (default=None)
    Optional parameter `walk_type` used when calling
    :func:`directed_laplacian_matrix <networkx.directed_laplacian_matrix>`.
    One of ``"random"``, ``"lazy"``, or ``"pagerank"``. If ``walk_type=None``
    (the default), then a value is selected according to the properties of `G`:
    - ``walk_type="random"`` if `G` is strongly connected and aperiodic
    - ``walk_type="lazy"`` if `G` is strongly connected but not aperiodic
    - ``walk_type="pagerank"`` for all other cases.
alpha : real (default = 0.95)
    Optional parameter `alpha` used when calling
    :func:`directed_laplacian_matrix <networkx.directed_laplacian_matrix>`.
    (1 - alpha) is the teleportation probability used with pagerank.
Returns
-------
nodes : dictionary
   Dictionary of nodes with Laplacian centrality as the value.
Examples
--------
>>> G = nx.Graph()
>>> edges = [(0, 1, 4), (0, 2, 2), (2, 1, 1), (1, 3, 2), (1, 4, 2), (4, 5, 1)]
>>> G.add_weighted_edges_from(edges)
>>> sorted((v, f"{c:0.2f}") for v, c in laplacian_centrality(G).items())
[(0, '0.70'), (1, '0.90'), (2, '0.28'), (3, '0.22'), (4, '0.26'), (5, '0.04')]
Notes
-----
The algorithm is implemented based on [1]_ with an extension to directed graphs
using the ``directed_laplacian_matrix`` function.
Raises
------
NetworkXPointlessConcept
    If the graph `G` is the null graph.
ZeroDivisionError
    If the graph `G` has no edges (is empty) and normalization is requested.
References
----------
.. [1] Qi, X., Fuller, E., Wu, Q., Wu, Y., and Zhang, C.-Q. (2012).
   Laplacian centrality: A new centrality measure for weighted networks.
   Information Sciences, 194:240-253.
   https://math.wvu.edu/~cqzhang/Publication-files/my-paper/INS-2012-Laplacian-W.pdf
See Also
```

```
77: (4)                 --------
78: (4)                 :func:`~networkx.linalg.laplacianmatrix.directed_laplacian_matrix`
79: (4)                 :func:`~networkx.linalg.laplacianmatrix.laplacian_matrix`
80: (4)                 """
81: (4)                 import numpy as np
82: (4)                 import scipy as sp
83: (4)                 if len(G) == 0:
84: (8)                     raise nx.NetworkXPointlessConcept("null graph has no centrality
defined")
85: (4)                 if G.size(weight=weight) == 0:
86: (8)                     if normalized:
87: (12)                        raise ZeroDivisionError("graph with no edges has zero full
energy")
88: (8)                     return {n: 0 for n in G}
89: (4)                 if nodelist is not None:
90: (8)                     nodeset = set(G.nbunch_iter(nodelist))
91: (8)                     if len(nodeset) != len(nodelist):
92: (12)                        raise nx.NetworkXError("nodelist has duplicate nodes or nodes not
in G")
93: (8)                     nodes = nodelist + [n for n in G if n not in nodeset]
94: (4)                 else:
95: (8)                     nodelist = nodes = list(G)
96: (4)                 if G.is_directed():
97: (8)                     lap_matrix = nx.directed_laplacian_matrix(G, nodes, weight, walk_type,
alpha)
98: (4)                 else:
99: (8)                     lap_matrix = nx.laplacian_matrix(G, nodes, weight).toarray()
100: (4)                full_energy = np.power(sp.linalg.eigh(lap_matrix, eigvals_only=True),
2).sum()
101: (4)                laplace_centralities_dict = {}
102: (4)                for i, node in enumerate(nodelist):
103: (8)                    all_but_i = list(np.arange(lap_matrix.shape[0]))
104: (8)                    all_but_i.remove(i)
105: (8)                    A_2 = lap_matrix[all_but_i, :][:, all_but_i]
106: (8)                    new_diag = lap_matrix.diagonal() - abs(lap_matrix[:, i])
107: (8)                    np.fill_diagonal(A_2, new_diag[all_but_i])
108: (8)                    if len(all_but_i) > 0:  # catches degenerate case of single node
109: (12)                       new_energy = np.power(sp.linalg.eigh(A_2, eigvals_only=True),
2).sum()
110: (8)                    else:
111: (12)                       new_energy = 0.0
112: (8)                    lapl_cent = full_energy - new_energy
113: (8)                    if normalized:
114: (12)                       lapl_cent = lapl_cent / full_energy
115: (8)                    laplace_centralities_dict[node] = float(lapl_cent)
116: (4)                return laplace_centralities_dict
```

File 147 - percolation.py:

```
1: (0)              """Percolation centrality measures."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.centrality.betweenness import (
4: (4)                  _single_source_dijkstra_path_basic as dijkstra,
5: (0)              )
6: (0)              from networkx.algorithms.centrality.betweenness import (
7: (4)                  _single_source_shortest_path_basic as shortest_path,
8: (0)              )
9: (0)              __all__ = ["percolation_centrality"]
10: (0)             @nx._dispatchable(node_attrs="attribute", edge_attrs="weight")
11: (0)             def percolation_centrality(G, attribute="percolation", states=None,
weight=None):
12: (4)                 r"""Compute the percolation centrality for nodes.
13: (4)                 Percolation centrality of a node $v$, at a given time, is defined
14: (4)                 as the proportion of ‘percolated paths’ that go through that node.
15: (4)                 This measure quantifies relative impact of nodes based on their
16: (4)                 topological connectivity, as well as their percolation states.
17: (4)                 Percolation states of nodes are used to depict network percolation
```

```
scenarios (such as during infection transmission in a social network
of individuals, spreading of computer viruses on computer networks, or
transmission of disease over a network of towns) over time. In this
measure usually the percolation state is expressed as a decimal
between 0.0 and 1.0.
When all nodes are in the same percolated state this measure is
equivalent to betweenness centrality.
Parameters
----------
G : graph
  A NetworkX graph.
attribute : None or string, optional (default='percolation')
  Name of the node attribute to use for percolation state, used
  if `states` is None. If a node does not set the attribute the
  state of that node will be set to the default value of 1.
  If all nodes do not have the attribute all nodes will be set to
  1 and the centrality measure will be equivalent to betweenness
centrality.
states : None or dict, optional (default=None)
  Specify percolation states for the nodes, nodes as keys states
  as values.
weight : None or string, optional (default=None)
  If None, all edge weights are considered equal.
  Otherwise holds the name of the edge attribute used as weight.
  The weight of an edge is treated as the length or distance between the
two sides.
Returns
-------
nodes : dictionary
   Dictionary of nodes with percolation centrality as the value.
See Also
--------
betweenness_centrality
Notes
-----
The algorithm is from Mahendra Piraveenan, Mikhail Prokopenko, and
Liaquat Hossain [1]_
Pair dependencies are calculated and accumulated using [2]_
For weighted graphs the edge weights must be greater than zero.
Zero edge weights can produce an infinite number of equal length
paths between pairs of nodes.
References
----------
.. [1] Mahendra Piraveenan, Mikhail Prokopenko, Liaquat Hossain
   Percolation Centrality: Quantifying Graph-Theoretic Impact of Nodes
   during Percolation in Networks
   http://journals.plos.org/plosone/article?
id=10.1371/journal.pone.0053095
.. [2] Ulrik Brandes:
   A Faster Algorithm for Betweenness Centrality.
   Journal of Mathematical Sociology 25(2):163-177, 2001.
   https://doi.org/10.1080/0022250X.2001.9990249
"""
percolation = dict.fromkeys(G, 0.0)  # b[v]=0 for v in G
nodes = G
if states is None:
    states = nx.get_node_attributes(nodes, attribute, default=1)
p_sigma_x_t = 0.0
for v in states.values():
    p_sigma_x_t += v
for s in nodes:
    if weight is None:  # use BFS
        S, P, sigma, _ = shortest_path(G, s)
    else:  # use Dijkstra's algorithm
        S, P, sigma, _ = dijkstra(G, s, weight)
    percolation = _accumulate_percolation(
        percolation, S, P, sigma, s, states, p_sigma_x_t
    )
n = len(G)
```

```
84: (4)                 for v in percolation:
85: (8)                     percolation[v] *= 1 / (n - 2)
86: (4)                 return percolation
87: (0)             def _accumulate_percolation(percolation, S, P, sigma, s, states, p_sigma_x_t):
88: (4)                 delta = dict.fromkeys(S, 0)
89: (4)                 while S:
90: (8)                     w = S.pop()
91: (8)                     coeff = (1 + delta[w]) / sigma[w]
92: (8)                     for v in P[w]:
93: (12)                        delta[v] += sigma[v] * coeff
94: (8)                     if w != s:
95: (12)                        pw_s_w = states[s] / (p_sigma_x_t - states[w])
96: (12)                        percolation[w] += delta[w] * pw_s_w
97: (4)                 return percolation
```

File 148 - reaching.py:

```
1: (0)              """Functions for computing reaching centrality of a node or a graph."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import pairwise
4: (0)              __all__ = ["global_reaching_centrality", "local_reaching_centrality"]
5: (0)              def _average_weight(G, path, weight=None):
6: (4)                  """Returns the average weight of an edge in a weighted path.
7: (4)                  Parameters
8: (4)                  ----------
9: (4)                  G : graph
10: (6)                   A networkx graph.
11: (4)                 path: list
12: (6)                   A list of vertices that define the path.
13: (4)                 weight : None or string, optional (default=None)
14: (6)                   If None, edge weights are ignored.  Then the average weight of an edge
15: (6)                   is assumed to be the multiplicative inverse of the length of the path.
16: (6)                   Otherwise holds the name of the edge attribute used as weight.
17: (4)                 """
18: (4)                 path_length = len(path) - 1
19: (4)                 if path_length <= 0:
20: (8)                     return 0
21: (4)                 if weight is None:
22: (8)                     return 1 / path_length
23: (4)                 total_weight = sum(G.edges[i, j][weight] for i, j in pairwise(path))
24: (4)                 return total_weight / path_length
25: (0)             @nx._dispatchable(edge_attrs="weight")
26: (0)             def global_reaching_centrality(G, weight=None, normalized=True):
27: (4)                 """Returns the global reaching centrality of a directed graph.
28: (4)                 The *global reaching centrality* of a weighted directed graph is the
29: (4)                 average over all nodes of the difference between the local reaching
30: (4)                 centrality of the node and the greatest local reaching centrality of
31: (4)                 any node in the graph [1]_. For more information on the local
32: (4)                 reaching centrality, see :func:`local_reaching_centrality`.
33: (4)                 Informally, the local reaching centrality is the proportion of the
34: (4)                 graph that is reachable from the neighbors of the node.
35: (4)                 Parameters
36: (4)                 ----------
37: (4)                 G : DiGraph
38: (8)                     A networkx DiGraph.
39: (4)                 weight : None or string, optional (default=None)
40: (8)                     Attribute to use for edge weights. If ``None``, each edge weight
41: (8)                     is assumed to be one. A higher weight implies a stronger
42: (8)                     connection between nodes and a *shorter* path length.
43: (4)                 normalized : bool, optional (default=True)
44: (8)                     Whether to normalize the edge weights by the total sum of edge
45: (8)                     weights.
46: (4)                 Returns
47: (4)                 -------
48: (4)                 h : float
49: (8)                     The global reaching centrality of the graph.
50: (4)                 Examples
```

```
    --------
    >>> G = nx.DiGraph()
    >>> G.add_edge(1, 2)
    >>> G.add_edge(1, 3)
    >>> nx.global_reaching_centrality(G)
    1.0
    >>> G.add_edge(3, 2)
    >>> nx.global_reaching_centrality(G)
    0.75
    See also
    --------
    local_reaching_centrality
    References
    ----------
    .. [1] Mones, Enys, Lilla Vicsek, and Tamás Vicsek.
           "Hierarchy Measure for Complex Networks."
           *PLoS ONE* 7.3 (2012): e33799.
           https://doi.org/10.1371/journal.pone.0033799
    """
    if nx.is_negatively_weighted(G, weight=weight):
        raise nx.NetworkXError("edge weights must be positive")
    total_weight = G.size(weight=weight)
    if total_weight <= 0:
        raise nx.NetworkXError("Size of G must be positive")
    if weight is not None:
        def as_distance(u, v, d):
            return total_weight / d.get(weight, 1)
        shortest_paths = nx.shortest_path(G, weight=as_distance)
    else:
        shortest_paths = nx.shortest_path(G)
    centrality = local_reaching_centrality
    lrc = [
        centrality(G, node, paths=paths, weight=weight, normalized=normalized)
        for node, paths in shortest_paths.items()
    ]
    max_lrc = max(lrc)
    return sum(max_lrc - c for c in lrc) / (len(G) - 1)
@nx._dispatchable(edge_attrs="weight")
def local_reaching_centrality(G, v, paths=None, weight=None, normalized=True):
    """Returns the local reaching centrality of a node in a directed
    graph.
    The *local reaching centrality* of a node in a directed graph is the
    proportion of other nodes reachable from that node [1]_.
    Parameters
    ----------
    G : DiGraph
        A NetworkX DiGraph.
    v : node
        A node in the directed graph `G`.
    paths : dictionary (default=None)
        If this is not `None` it must be a dictionary representation
        of single-source shortest paths, as computed by, for example,
        :func:`networkx.shortest_path` with source node `v`. Use this
        keyword argument if you intend to invoke this function many
        times but don't want the paths to be recomputed each time.
    weight : None or string, optional (default=None)
        Attribute to use for edge weights.  If `None`, each edge weight
        is assumed to be one. A higher weight implies a stronger
        connection between nodes and a *shorter* path length.
    normalized : bool, optional (default=True)
        Whether to normalize the edge weights by the total sum of edge
        weights.
    Returns
    -------
    h : float
        The local reaching centrality of the node ``v`` in the graph
        ``G``.
    Examples
    --------
```

```
120: (4)                >>> G = nx.DiGraph()
121: (4)                >>> G.add_edges_from([(1, 2), (1, 3)])
122: (4)                >>> nx.local_reaching_centrality(G, 3)
123: (4)                0.0
124: (4)                >>> G.add_edge(3, 2)
125: (4)                >>> nx.local_reaching_centrality(G, 3)
126: (4)                0.5
127: (4)                See also
128: (4)                --------
129: (4)                global_reaching_centrality
130: (4)                References
131: (4)                ----------
132: (4)                .. [1] Mones, Enys, Lilla Vicsek, and Tamás Vicsek.
133: (11)                       "Hierarchy Measure for Complex Networks."
134: (11)                       *PLoS ONE* 7.3 (2012): e33799.
135: (11)                       https://doi.org/10.1371/journal.pone.0033799
136: (4)                """
137: (4)                if paths is None:
138: (8)                    if nx.is_negatively_weighted(G, weight=weight):
139: (12)                       raise nx.NetworkXError("edge weights must be positive")
140: (8)                    total_weight = G.size(weight=weight)
141: (8)                    if total_weight <= 0:
142: (12)                       raise nx.NetworkXError("Size of G must be positive")
143: (8)                    if weight is not None:
144: (12)                       def as_distance(u, v, d):
145: (16)                           return total_weight / d.get(weight, 1)
146: (12)                       paths = nx.shortest_path(G, source=v, weight=as_distance)
147: (8)                    else:
148: (12)                       paths = nx.shortest_path(G, source=v)
149: (4)                if weight is None and G.is_directed():
150: (8)                    return (len(paths) - 1) / (len(G) - 1)
151: (4)                if normalized and weight is not None:
152: (8)                    norm = G.size(weight=weight) / G.size()
153: (4)                else:
154: (8)                    norm = 1
155: (4)                avgw = (_average_weight(G, path, weight=weight) for path in
paths.values())
156: (4)                sum_avg_weight = sum(avgw) / norm
157: (4)                return sum_avg_weight / (len(G) - 1)

----------------------------------------

File 149 - second_order.py:

1: (0)              """Copyright (c) 2015 – Thomson Licensing, SAS
2: (0)              Redistribution and use in source and binary forms, with or without
3: (0)              modification, are permitted (subject to the limitations in the
4: (0)              disclaimer below) provided that the following conditions are met:
5: (0)              * Redistributions of source code must retain the above copyright
6: (0)              notice, this list of conditions and the following disclaimer.
7: (0)              * Redistributions in binary form must reproduce the above copyright
8: (0)              notice, this list of conditions and the following disclaimer in the
9: (0)              documentation and/or other materials provided with the distribution.
10: (0)             * Neither the name of Thomson Licensing, or Technicolor, nor the names
11: (0)             of its contributors may be used to endorse or promote products derived
12: (0)             from this software without specific prior written permission.
13: (0)             NO EXPRESS OR IMPLIED LICENSES TO ANY PARTY'S PATENT RIGHTS ARE
14: (0)             GRANTED BY THIS LICENSE.  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT
15: (0)             HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
16: (0)             WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
17: (0)             MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18: (0)             DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
19: (0)             LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
20: (0)             CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
21: (0)             SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
22: (0)             BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
23: (0)             WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
24: (0)             OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
25: (0)             IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
(0)             """
(0)             import networkx as nx
(0)             from networkx.utils import not_implemented_for
(0)             __all__ = ["second_order_centrality"]
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def second_order_centrality(G, weight="weight"):
(4)                 """Compute the second order centrality for nodes of G.
(4)                 The second order centrality of a given node is the standard deviation of
(4)                 the return times to that node of a perpetual random walk on G:
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A NetworkX connected and undirected graph.
(4)                 weight : string or None, optional (default="weight")
(8)                     The name of an edge attribute that holds the numerical value
(8)                     used as a weight. If None then each edge has weight 1.
(4)                 Returns
(4)                 -------
(4)                 nodes : dictionary
(7)                    Dictionary keyed by node with second order centrality as the value.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.star_graph(10)
(4)                 >>> soc = nx.second_order_centrality(G)
(4)                 >>> print(sorted(soc.items(), key=lambda x: x[1])[0][0])  # pick first id
(4)                 0
(4)                 Raises
(4)                 ------
(4)                 NetworkXException
(8)                     If the graph G is empty, non connected or has negative weights.
(4)                 See Also
(4)                 --------
(4)                 betweenness_centrality
(4)                 Notes
(4)                 -----
(4)                 Lower values of second order centrality indicate higher centrality.
(4)                 The algorithm is from Kermarrec, Le Merrer, Sericola and Trédan [1]_.
(4)                 This code implements the analytical version of the algorithm, i.e.,
(4)                 there is no simulation of a random walk process involved. The random walk
(4)                 is here unbiased (corresponding to eq 6 of the paper [1]_), thus the
(4)                 centrality values are the standard deviations for random walk return times
(4)                 on the transformed input graph G (equal in-degree at each nodes by adding
(4)                 self-loops).
(4)                 Complexity of this implementation, made to run locally on a single
machine,
(4)                 is O(n^3), with n the size of G, which makes it viable only for small
(4)                 graphs.
(4)                 References
(4)                 ----------
(4)                 .. [1] Anne-Marie Kermarrec, Erwan Le Merrer, Bruno Sericola, Gilles
Trédan
(7)                    "Second order centrality: Distributed assessment of nodes criticity in
(7)                    complex networks", Elsevier Computer Communications 34(5):619-628,
2011.
(4)                 """
(4)                 import numpy as np
(4)                 n = len(G)
(4)                 if n == 0:
(8)                     raise nx.NetworkXException("Empty graph.")
(4)                 if not nx.is_connected(G):
(8)                     raise nx.NetworkXException("Non connected graph.")
(4)                 if any(d.get(weight, 0) < 0 for u, v, d in G.edges(data=True)):
(8)                     raise nx.NetworkXException("Graph has negative edge weights.")
(4)                 G = nx.DiGraph(G)
(4)                 in_deg = dict(G.in_degree(weight=weight))
(4)                 d_max = max(in_deg.values())
(4)                 for i, deg in in_deg.items():
(8)                     if deg < d_max:
```

```
92: (12)                        G.add_edge(i, i, weight=d_max - deg)
93: (4)                 P = nx.to_numpy_array(G)
94: (4)                 P /= P.sum(axis=1)[:, np.newaxis]  # to transition probability matrix
95: (4)                 def _Qj(P, j):
96: (8)                     P = P.copy()
97: (8)                     P[:, j] = 0
98: (8)                     return P
99: (4)                 M = np.empty([n, n])
100: (4)                for i in range(n):
101: (8)                    M[:, i] = np.linalg.solve(
102: (12)                       np.identity(n) - _Qj(P, i), np.ones([n, 1])[:, 0]
103: (8)                    )  # eq 3
104: (4)                return dict(
105: (8)                    zip(
106: (12)                       G.nodes,
107: (12)                       (float(np.sqrt(2 * np.sum(M[:, i]) - n * (n + 1))) for i in
range(n)),
108: (8)                    )
109: (4)                )  # eq 6
```

File 150 - subgraph_alg.py:

```
1: (0)              """
2: (0)              Subraph centrality and communicability betweenness.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = [
7: (4)                  "subgraph_centrality_exp",
8: (4)                  "subgraph_centrality",
9: (4)                  "communicability_betweenness_centrality",
10: (4)                 "estrada_index",
11: (0)             ]
12: (0)             @not_implemented_for("directed")
13: (0)             @not_implemented_for("multigraph")
14: (0)             @nx._dispatchable
15: (0)             def subgraph_centrality_exp(G):
16: (4)                 r"""Returns the subgraph centrality for each node of G.
17: (4)                 Subgraph centrality  of a node `n` is the sum of weighted closed
18: (4)                 walks of all lengths starting and ending at node `n`. The weights
19: (4)                 decrease with path length. Each closed walk is associated with a
20: (4)                 connected subgraph ([1]_).
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G: graph
24: (4)                 Returns
25: (4)                 -------
26: (4)                 nodes:dictionary
27: (8)                     Dictionary of nodes with subgraph centrality as the value.
28: (4)                 Raises
29: (4)                 ------
30: (4)                 NetworkXError
31: (8)                     If the graph is not undirected and simple.
32: (4)                 See Also
33: (4)                 --------
34: (4)                 subgraph_centrality:
35: (8)                     Alternative algorithm of the subgraph centrality for each node of G.
36: (4)                 Notes
37: (4)                 -----
38: (4)                 This version of the algorithm exponentiates the adjacency matrix.
39: (4)                 The subgraph centrality of a node `u` in G can be found using
40: (4)                 the matrix exponential of the adjacency matrix of G [1]_,
41: (4)                 .. math::
42: (8)                     SC(u)=(e^A)_{uu} .
43: (4)                 References
44: (4)                 ----------
45: (4)                 .. [1] Ernesto Estrada, Juan A. Rodriguez-Velazquez,
```

```
46: (7)                    "Subgraph centrality in complex networks",
47: (7)                    Physical Review E 71, 056103 (2005).
48: (7)                    https://arxiv.org/abs/cond-mat/0504730
49: (4)                Examples
50: (4)                --------
51: (4)                (Example from [1]_)
52: (4)                >>> G = nx.Graph(
53: (4)                ...     [
54: (4)                ...         (1, 2),
55: (4)                ...         (1, 5),
56: (4)                ...         (1, 8),
57: (4)                ...         (2, 3),
58: (4)                ...         (2, 8),
59: (4)                ...         (3, 4),
60: (4)                ...         (3, 6),
61: (4)                ...         (4, 5),
62: (4)                ...         (4, 7),
63: (4)                ...         (5, 6),
64: (4)                ...         (6, 7),
65: (4)                ...         (7, 8),
66: (4)                ...     ]
67: (4)                ... )
68: (4)                >>> sc = nx.subgraph_centrality_exp(G)
69: (4)                >>> print([f"{node} {sc[node]:0.2f}" for node in sorted(sc)])
70: (4)                ['1 3.90', '2 3.90', '3 3.64', '4 3.71', '5 3.64', '6 3.71', '7 3.64', '8
3.90']
71: (4)                """
72: (4)                import scipy as sp
73: (4)                nodelist = list(G)  # ordering of nodes in matrix
74: (4)                A = nx.to_numpy_array(G, nodelist)
75: (4)                A[A != 0.0] = 1
76: (4)                expA = sp.linalg.expm(A)
77: (4)                sc = dict(zip(nodelist, map(float, expA.diagonal())))
78: (4)                return sc
79: (0)            @not_implemented_for("directed")
80: (0)            @not_implemented_for("multigraph")
81: (0)            @nx._dispatchable
82: (0)            def subgraph_centrality(G):
83: (4)                r"""Returns subgraph centrality for each node in G.
84: (4)                Subgraph centrality  of a node `n` is the sum of weighted closed
85: (4)                walks of all lengths starting and ending at node `n`. The weights
86: (4)                decrease with path length. Each closed walk is associated with a
87: (4)                connected subgraph ([1]_).
88: (4)                Parameters
89: (4)                ----------
90: (4)                G: graph
91: (4)                Returns
92: (4)                -------
93: (4)                nodes : dictionary
94: (7)                   Dictionary of nodes with subgraph centrality as the value.
95: (4)                Raises
96: (4)                ------
97: (4)                NetworkXError
98: (7)                   If the graph is not undirected and simple.
99: (4)                See Also
100: (4)               --------
101: (4)               subgraph_centrality_exp:
102: (8)                   Alternative algorithm of the subgraph centrality for each node of G.
103: (4)               Notes
104: (4)               -----
105: (4)               This version of the algorithm computes eigenvalues and eigenvectors
106: (4)               of the adjacency matrix.
107: (4)               Subgraph centrality of a node `u` in G can be found using
108: (4)               a spectral decomposition of the adjacency matrix [1]_,
109: (4)               .. math::
110: (7)                  SC(u)=\sum_{j=1}^{N}(v_{j}^{u})^2 e^{\lambda_{j}},
111: (4)               where `v_j` is an eigenvector of the adjacency matrix `A` of G
112: (4)               corresponding to the eigenvalue `\lambda_j`.
113: (4)               Examples
```

```
114: (4)                --------
115: (4)                (Example from [1]_)
116: (4)                >>> G = nx.Graph(
117: (4)                ...     [
118: (4)                ...         (1, 2),
119: (4)                ...         (1, 5),
120: (4)                ...         (1, 8),
121: (4)                ...         (2, 3),
122: (4)                ...         (2, 8),
123: (4)                ...         (3, 4),
124: (4)                ...         (3, 6),
125: (4)                ...         (4, 5),
126: (4)                ...         (4, 7),
127: (4)                ...         (5, 6),
128: (4)                ...         (6, 7),
129: (4)                ...         (7, 8),
130: (4)                ...     ]
131: (4)                ... )
132: (4)                >>> sc = nx.subgraph_centrality(G)
133: (4)                >>> print([f"{node} {sc[node]:0.2f}" for node in sorted(sc)])
134: (4)                ['1 3.90', '2 3.90', '3 3.64', '4 3.71', '5 3.64', '6 3.71', '7 3.64', '8
3.90']
135: (4)                References
136: (4)                ----------
137: (4)                .. [1] Ernesto Estrada, Juan A. Rodriguez-Velazquez,
138: (7)                   "Subgraph centrality in complex networks",
139: (7)                   Physical Review E 71, 056103 (2005).
140: (7)                   https://arxiv.org/abs/cond-mat/0504730
141: (4)                """
142: (4)                import numpy as np
143: (4)                nodelist = list(G)  # ordering of nodes in matrix
144: (4)                A = nx.to_numpy_array(G, nodelist)
145: (4)                A[np.nonzero(A)] = 1
146: (4)                w, v = np.linalg.eigh(A)
147: (4)                vsquare = np.array(v) ** 2
148: (4)                expw = np.exp(w)
149: (4)                xg = vsquare @ expw
150: (4)                sc = dict(zip(nodelist, map(float, xg)))
151: (4)                return sc
152: (0)            @not_implemented_for("directed")
153: (0)            @not_implemented_for("multigraph")
154: (0)            @nx._dispatchable
155: (0)            def communicability_betweenness_centrality(G):
156: (4)                r"""Returns subgraph communicability for all pairs of nodes in G.
157: (4)                Communicability betweenness measure makes use of the number of walks
158: (4)                connecting every pair of nodes as the basis of a betweenness centrality
159: (4)                measure.
160: (4)                Parameters
161: (4)                ----------
162: (4)                G: graph
163: (4)                Returns
164: (4)                -------
165: (4)                nodes : dictionary
166: (8)                    Dictionary of nodes with communicability betweenness as the value.
167: (4)                Raises
168: (4)                ------
169: (4)                NetworkXError
170: (8)                    If the graph is not undirected and simple.
171: (4)                Notes
172: (4)                -----
173: (4)                Let `G=(V,E)` be a simple undirected graph with `n` nodes and `m` edges,
174: (4)                and `A` denote the adjacency matrix of `G`.
175: (4)                Let `G(r)=(V,E(r))` be the graph resulting from
176: (4)                removing all edges connected to node `r` but not the node itself.
177: (4)                The adjacency matrix for `G(r)` is `A+E(r)`,  where `E(r)` has nonzeros
178: (4)                only in row and column `r`.
179: (4)                The subraph betweenness of a node `r`  is [1]_
180: (4)                .. math::
181: (9)                     \omega_{r} = \frac{1}{C}\sum_{p}\sum_{q}\frac{G_{prq}}{G_{pq}},
```

```
 182: (9)                     p\neq q, q\neq r,
 183: (4)                 where
 184: (4)                 `G_{prq}=(e^{A}_{pq} - (e^{A+E(r)})_{pq}`  is the number of walks
 185: (4)                 involving node r,
 186: (4)                 `G_{pq}=(e^{A})_{pq}` is the number of closed walks starting
 187: (4)                 at node `p` and ending at node `q`,
 188: (4)                 and `C=(n-1)^{2}-(n-1)` is a normalization factor equal to the
 189: (4)                 number of terms in the sum.
 190: (4)                 The resulting `\omega_{r}` takes values between zero and one.
 191: (4)                 The lower bound cannot be attained for a connected
 192: (4)                 graph, and the upper bound is attained in the star graph.
 193: (4)                 References
 194: (4)                 ----------
 195: (4)                 .. [1] Ernesto Estrada, Desmond J. Higham, Naomichi Hatano,
 196: (7)                    "Communicability Betweenness in Complex Networks"
 197: (7)                    Physica A 388 (2009) 764-774.
 198: (7)                    https://arxiv.org/abs/0905.4102
 199: (4)                 Examples
 200: (4)                 --------
 201: (4)                 >>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3),
(3, 6)])
 202: (4)                 >>> cbc = nx.communicability_betweenness_centrality(G)
 203: (4)                 >>> print([f"{node} {cbc[node]:0.2f}" for node in sorted(cbc)])
 204: (4)                 ['0 0.03', '1 0.45', '2 0.51', '3 0.45', '4 0.40', '5 0.19', '6 0.03']
 205: (4)                 """
 206: (4)                 import numpy as np
 207: (4)                 import scipy as sp
 208: (4)                 nodelist = list(G)  # ordering of nodes in matrix
 209: (4)                 n = len(nodelist)
 210: (4)                 A = nx.to_numpy_array(G, nodelist)
 211: (4)                 A[np.nonzero(A)] = 1
 212: (4)                 expA = sp.linalg.expm(A)
 213: (4)                 mapping = dict(zip(nodelist, range(n)))
 214: (4)                 cbc = {}
 215: (4)                 for v in G:
 216: (8)                     i = mapping[v]
 217: (8)                     row = A[i, :].copy()
 218: (8)                     col = A[:, i].copy()
 219: (8)                     A[i, :] = 0
 220: (8)                     A[:, i] = 0
 221: (8)                     B = (expA - sp.linalg.expm(A)) / expA
 222: (8)                     B[i, :] = 0
 223: (8)                     B[:, i] = 0
 224: (8)                     B -= np.diag(np.diag(B))
 225: (8)                     cbc[v] = float(B.sum())
 226: (8)                     A[i, :] = row
 227: (8)                     A[:, i] = col
 228: (4)                 order = len(cbc)
 229: (4)                 if order > 2:
 230: (8)                     scale = 1.0 / ((order - 1.0) ** 2 - (order - 1.0))
 231: (8)                     cbc = {node: value * scale for node, value in cbc.items()}
 232: (4)                 return cbc
 233: (0)             @nx._dispatchable
 234: (0)             def estrada_index(G):
 235: (4)                 r"""Returns the Estrada index of a the graph G.
 236: (4)                 The Estrada Index is a topological index of folding or 3D "compactness"
([1]_).
 237: (4)                 Parameters
 238: (4)                 ----------
 239: (4)                 G: graph
 240: (4)                 Returns
 241: (4)                 -------
 242: (4)                 estrada index: float
 243: (4)                 Raises
 244: (4)                 ------
 245: (4)                 NetworkXError
 246: (8)                     If the graph is not undirected and simple.
 247: (4)                 Notes
 248: (4)                 -----
```

```
249: (4)                Let `G=(V,E)` be a simple undirected graph with `n` nodes  and let
250: (4)                `\lambda_{1}\leq\lambda_{2}\leq\cdots\lambda_{n}`
251: (4)                be a non-increasing ordering of the eigenvalues of its adjacency
252: (4)                matrix `A`. The Estrada index is ([1]_, [2]_)
253: (4)                .. math::
254: (8)                    EE(G)=\sum_{j=1}^n e^{\lambda _j}.
255: (4)                References
256: (4)                ----------
257: (4)                .. [1] E. Estrada, "Characterization of 3D molecular structure",
258: (7)                   Chem. Phys. Lett. 319, 713 (2000).
259: (7)                   https://doi.org/10.1016/S0009-2614(00)00158-5
260: (4)                .. [2] José Antonio de la Peñaa, Ivan Gutman, Juan Rada,
261: (7)                   "Estimating the Estrada index",
262: (7)                   Linear Algebra and its Applications. 427, 1 (2007).
263: (7)                   https://doi.org/10.1016/j.laa.2007.06.020
264: (4)                Examples
265: (4)                --------
266: (4)                >>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3),
(3, 6)])
267: (4)                >>> ei = nx.estrada_index(G)
268: (4)                >>> print(f"{ei:0.5}")
269: (4)                20.55
270: (4)                """
271: (4)                return sum(subgraph_centrality(G).values())

----------------------------------------

File 151 - trophic.py:

1: (0)              """Trophic levels"""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for
4: (0)              __all__ = ["trophic_levels", "trophic_differences",
"trophic_incoherence_parameter"]
5: (0)              @not_implemented_for("undirected")
6: (0)              @nx._dispatchable(edge_attrs="weight")
7: (0)              def trophic_levels(G, weight="weight"):
8: (4)                  r"""Compute the trophic levels of nodes.
9: (4)                  The trophic level of a node $i$ is
10: (4)                 .. math::
11: (8)                     s_i = 1 + \frac{1}{k^{in}_i} \sum_{j} a_{ij} s_j
12: (4)                 where $k^{in}_i$ is the in-degree of i
13: (4)                 .. math::
14: (8)                     k^{in}_i = \sum_{j} a_{ij}
15: (4)                 and nodes with $k^{in}_i = 0$ have $s_i = 1$ by convention.
16: (4)                 These are calculated using the method outlined in Levine [1]_.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : DiGraph
20: (8)                     A directed networkx graph
21: (4)                 Returns
22: (4)                 -------
23: (4)                 nodes : dict
24: (8)                     Dictionary of nodes with trophic level as the value.
25: (4)                 References
26: (4)                 ----------
27: (4)                 .. [1] Stephen Levine (1980) J. theor. Biol. 83, 195-207
28: (4)                 """
29: (4)                 import numpy as np
30: (4)                 a = nx.adjacency_matrix(G, weight=weight).T.toarray()
31: (4)                 rowsum = np.sum(a, axis=1)
32: (4)                 p = a[rowsum != 0][:, rowsum != 0]
33: (4)                 p = p / rowsum[rowsum != 0][:, np.newaxis]
34: (4)                 nn = p.shape[0]
35: (4)                 i = np.eye(nn)
36: (4)                 try:
37: (8)                     n = np.linalg.inv(i - p)
38: (4)                 except np.linalg.LinAlgError as err:
39: (8)                     msg = (
```

```
40: (12)                        "Trophic levels are only defined for graphs where every "
41: (12)                        + "node has a path from a basal node (basal nodes are nodes "
42: (12)                        + "with no incoming edges)."
43: (8)                     )
44: (8)                     raise nx.NetworkXError(msg) from err
45: (4)                 y = n.sum(axis=1) + 1
46: (4)                 levels = {}
47: (4)                 zero_node_ids = (node_id for node_id, degree in G.in_degree if degree ==
0)
48: (4)                 for node_id in zero_node_ids:
49: (8)                     levels[node_id] = 1
50: (4)                 nonzero_node_ids = (node_id for node_id, degree in G.in_degree if degree
!= 0)
51: (4)                 for i, node_id in enumerate(nonzero_node_ids):
52: (8)                     levels[node_id] = y.item(i)
53: (4)                 return levels
54: (0)             @not_implemented_for("undirected")
55: (0)             @nx._dispatchable(edge_attrs="weight")
56: (0)             def trophic_differences(G, weight="weight"):
57: (4)                 r"""Compute the trophic differences of the edges of a directed graph.
58: (4)                 The trophic difference $x_ij$ for each edge is defined in Johnson et al.
59: (4)                 [1]_ as:
60: (4)                 .. math::
61: (8)                     x_ij = s_j - s_i
62: (4)                 Where $s_i$ is the trophic level of node $i$.
63: (4)                 Parameters
64: (4)                 ----------
65: (4)                 G : DiGraph
66: (8)                     A directed networkx graph
67: (4)                 Returns
68: (4)                 -------
69: (4)                 diffs : dict
70: (8)                     Dictionary of edges with trophic differences as the value.
71: (4)                 References
72: (4)                 ----------
73: (4)                 .. [1] Samuel Johnson, Virginia Dominguez-Garcia, Luca Donetti, Miguel A.
74: (8)                     Munoz (2014) PNAS "Trophic coherence determines food-web stability"
75: (4)                 """
76: (4)                 levels = trophic_levels(G, weight=weight)
77: (4)                 diffs = {}
78: (4)                 for u, v in G.edges:
79: (8)                     diffs[(u, v)] = levels[v] - levels[u]
80: (4)                 return diffs
81: (0)             @not_implemented_for("undirected")
82: (0)             @nx._dispatchable(edge_attrs="weight")
83: (0)             def trophic_incoherence_parameter(G, weight="weight", cannibalism=False):
84: (4)                 r"""Compute the trophic incoherence parameter of a graph.
85: (4)                 Trophic coherence is defined as the homogeneity of the distribution of
86: (4)                 trophic distances: the more similar, the more coherent. This is measured
by
87: (4)                 the standard deviation of the trophic differences and referred to as the
88: (4)                 trophic incoherence parameter $q$ by [1].
89: (4)                 Parameters
90: (4)                 ----------
91: (4)                 G : DiGraph
92: (8)                     A directed networkx graph
93: (4)                 cannibalism: Boolean
94: (8)                     If set to False, self edges are not considered in the calculation
95: (4)                 Returns
96: (4)                 -------
97: (4)                 trophic_incoherence_parameter : float
98: (8)                     The trophic coherence of a graph
99: (4)                 References
100: (4)                ----------
101: (4)                .. [1] Samuel Johnson, Virginia Dominguez-Garcia, Luca Donetti, Miguel A.
102: (8)                    Munoz (2014) PNAS "Trophic coherence determines food-web stability"
103: (4)                """
104: (4)                import numpy as np
105: (4)                if cannibalism:
```

```
106: (8)                    diffs = trophic_differences(G, weight=weight)
107: (4)                else:
108: (8)                    self_loops = list(nx.selfloop_edges(G))
109: (8)                    if self_loops:
110: (12)                       G_2 = G.copy()
111: (12)                       G_2.remove_edges_from(self_loops)
112: (8)                    else:
113: (12)                       G_2 = G
114: (8)                    diffs = trophic_differences(G_2, weight=weight)
115: (4)                return float(np.std(list(diffs.values())))

----------------------------------------

File 152 - voterank_alg.py:

1: (0)              """Algorithm to select influential nodes in a graph using VoteRank."""
2: (0)              import networkx as nx
3: (0)              __all__ = ["voterank"]
4: (0)              @nx._dispatchable
5: (0)              def voterank(G, number_of_nodes=None):
6: (4)                  """Select a list of influential nodes in a graph using VoteRank algorithm
7: (4)                  VoteRank [1]_ computes a ranking of the nodes in a graph G based on a
8: (4)                  voting scheme. With VoteRank, all nodes vote for each of its in-neighbors
9: (4)                  and the node with the highest votes is elected iteratively. The voting
10: (4)                 ability of out-neighbors of elected nodes is decreased in subsequent
turns.
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 G : graph
14: (8)                     A NetworkX graph.
15: (4)                 number_of_nodes : integer, optional
16: (8)                     Number of ranked nodes to extract (default all nodes).
17: (4)                 Returns
18: (4)                 -------
19: (4)                 voterank : list
20: (8)                     Ordered list of computed seeds.
21: (8)                     Only nodes with positive number of votes are returned.
22: (4)                 Examples
23: (4)                 --------
24: (4)                 >>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 4)])
25: (4)                 >>> nx.voterank(G)
26: (4)                 [0, 1]
27: (4)                 The algorithm can be used both for undirected and directed graphs.
28: (4)                 However, the directed version is different in two ways:
29: (4)                 (i) nodes only vote for their in-neighbors and
30: (4)                 (ii) only the voting ability of elected node and its out-neighbors are
updated:
31: (4)                 >>> G = nx.DiGraph([(0, 1), (2, 1), (2, 3), (3, 4)])
32: (4)                 >>> nx.voterank(G)
33: (4)                 [2, 3]
34: (4)                 Notes
35: (4)                 -----
36: (4)                 Each edge is treated independently in case of multigraphs.
37: (4)                 References
38: (4)                 ----------
39: (4)                 .. [1] Zhang, J.-X. et al. (2016).
40: (8)                     Identifying a set of influential spreaders in complex networks.
41: (8)                     Sci. Rep. 6, 27823; doi: 10.1038/srep27823.
42: (4)                 """
43: (4)                 influential_nodes = []
44: (4)                 vote_rank = {}
45: (4)                 if len(G) == 0:
46: (8)                     return influential_nodes
47: (4)                 if number_of_nodes is None or number_of_nodes > len(G):
48: (8)                     number_of_nodes = len(G)
49: (4)                 if G.is_directed():
50: (8)                     avgDegree = sum(deg for _, deg in G.out_degree()) / len(G)
51: (4)                 else:
52: (8)                     avgDegree = sum(deg for _, deg in G.degree()) / len(G)
```

```
53: (4)                 for n in G.nodes():
54: (8)                     vote_rank[n] = [0, 1]
55: (4)                 for _ in range(number_of_nodes):
56: (8)                     for n in G.nodes():
57: (12)                        vote_rank[n][0] = 0
58: (8)                     for n, nbr in G.edges():
59: (12)                        vote_rank[n][0] += vote_rank[nbr][1]
60: (12)                        if not G.is_directed():
61: (16)                            vote_rank[nbr][0] += vote_rank[n][1]
62: (8)                     for n in influential_nodes:
63: (12)                        vote_rank[n][0] = 0
64: (8)                     n = max(G.nodes, key=lambda x: vote_rank[x][0])
65: (8)                     if vote_rank[n][0] == 0:
66: (12)                        return influential_nodes
67: (8)                     influential_nodes.append(n)
68: (8)                     vote_rank[n] = [0, 0]
69: (8)                     for _, nbr in G.edges(n):
70: (12)                        vote_rank[nbr][1] -= 1 / avgDegree
71: (12)                        vote_rank[nbr][1] = max(vote_rank[nbr][1], 0)
72: (4)                 return influential_nodes
```

File 153 - __init__.py:

```
1: (0)
```

File 154 - test_betweenness_centrality.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def weighted_G():
4: (4)                  G = nx.Graph()
5: (4)                  G.add_edge(0, 1, weight=3)
6: (4)                  G.add_edge(0, 2, weight=2)
7: (4)                  G.add_edge(0, 3, weight=6)
8: (4)                  G.add_edge(0, 4, weight=4)
9: (4)                  G.add_edge(1, 3, weight=5)
10: (4)                 G.add_edge(1, 5, weight=5)
11: (4)                 G.add_edge(2, 4, weight=1)
12: (4)                 G.add_edge(3, 4, weight=2)
13: (4)                 G.add_edge(3, 5, weight=1)
14: (4)                 G.add_edge(4, 5, weight=4)
15: (4)                 return G
16: (0)             class TestBetweennessCentrality:
17: (4)                 def test_K5(self):
18: (8)                     """Betweenness centrality: K5"""
19: (8)                     G = nx.complete_graph(5)
20: (8)                     b = nx.betweenness_centrality(G, weight=None, normalized=False)
21: (8)                     b_answer = {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0}
22: (8)                     for n in sorted(G):
23: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
24: (4)                 def test_K5_endpoints(self):
25: (8)                     """Betweenness centrality: K5 endpoints"""
26: (8)                     G = nx.complete_graph(5)
27: (8)                     b = nx.betweenness_centrality(G, weight=None, normalized=False,
endpoints=True)
28: (8)                     b_answer = {0: 4.0, 1: 4.0, 2: 4.0, 3: 4.0, 4: 4.0}
29: (8)                     for n in sorted(G):
30: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
31: (8)                     b = nx.betweenness_centrality(G, weight=None, normalized=True,
endpoints=True)
32: (8)                     b_answer = {0: 0.4, 1: 0.4, 2: 0.4, 3: 0.4, 4: 0.4}
33: (8)                     for n in sorted(G):
34: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
35: (4)                 def test_P3_normalized(self):
36: (8)                     """Betweenness centrality: P3 normalized"""
```

```
(8)                     G = nx.path_graph(3)
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=True)
(8)                     b_answer = {0: 0.0, 1: 1.0, 2: 0.0}
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_P3(self):
(8)                     """Betweenness centrality: P3"""
(8)                     G = nx.path_graph(3)
(8)                     b_answer = {0: 0.0, 1: 1.0, 2: 0.0}
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_sample_from_P3(self):
(8)                     """Betweenness centrality: P3 sample"""
(8)                     G = nx.path_graph(3)
(8)                     b_answer = {0: 0.0, 1: 1.0, 2: 0.0}
(8)                     b = nx.betweenness_centrality(G, k=3, weight=None, normalized=False,
seed=1)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(8)                     b = nx.betweenness_centrality(G, k=2, weight=None, normalized=False,
seed=1)
(8)                     b_approx1 = {0: 0.0, 1: 1.5, 2: 0.0}
(8)                     b_approx2 = {0: 0.0, 1: 0.75, 2: 0.0}
(8)                     for n in sorted(G):
(12)                        assert b[n] in (b_approx1[n], b_approx2[n])
(4)                 def test_P3_endpoints(self):
(8)                     """Betweenness centrality: P3 endpoints"""
(8)                     G = nx.path_graph(3)
(8)                     b_answer = {0: 2.0, 1: 3.0, 2: 2.0}
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=False,
endpoints=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(8)                     b_answer = {0: 2 / 3, 1: 1.0, 2: 2 / 3}
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=True,
endpoints=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_krackhardt_kite_graph(self):
(8)                     """Betweenness centrality: Krackhardt kite graph"""
(8)                     G = nx.krackhardt_kite_graph()
(8)                     b_answer = {
(12)                        0: 1.667,
(12)                        1: 1.667,
(12)                        2: 0.000,
(12)                        3: 7.333,
(12)                        4: 0.000,
(12)                        5: 16.667,
(12)                        6: 16.667,
(12)                        7: 28.000,
(12)                        8: 16.000,
(12)                        9: 0.000,
(8)                     }
(8)                     for b in b_answer:
(12)                        b_answer[b] /= 2
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_krackhardt_kite_graph_normalized(self):
(8)                     """Betweenness centrality: Krackhardt kite graph normalized"""
(8)                     G = nx.krackhardt_kite_graph()
(8)                     b_answer = {
(12)                        0: 0.023,
(12)                        1: 0.023,
(12)                        2: 0.000,
(12)                        3: 0.102,
(12)                        4: 0.000,
(12)                        5: 0.231,
```

```
(12)                        6: 0.231,
(12)                        7: 0.389,
(12)                        8: 0.222,
(12)                        9: 0.000,
(8)                     }
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_florentine_families_graph(self):
(8)                     """Betweenness centrality: Florentine families graph"""
(8)                     G = nx.florentine_families_graph()
(8)                     b_answer = {
(12)                        "Acciaiuoli": 0.000,
(12)                        "Albizzi": 0.212,
(12)                        "Barbadori": 0.093,
(12)                        "Bischeri": 0.104,
(12)                        "Castellani": 0.055,
(12)                        "Ginori": 0.000,
(12)                        "Guadagni": 0.255,
(12)                        "Lamberteschi": 0.000,
(12)                        "Medici": 0.522,
(12)                        "Pazzi": 0.000,
(12)                        "Peruzzi": 0.022,
(12)                        "Ridolfi": 0.114,
(12)                        "Salviati": 0.143,
(12)                        "Strozzi": 0.103,
(12)                        "Tornabuoni": 0.092,
(8)                     }
(8)                     b = nx.betweenness_centrality(G, weight=None, normalized=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_les_miserables_graph(self):
(8)                     """Betweenness centrality: Les Miserables graph"""
(8)                     G = nx.les_miserables_graph()
(8)                     b_answer = {
(12)                        "Napoleon": 0.000,
(12)                        "Myriel": 0.177,
(12)                        "MlleBaptistine": 0.000,
(12)                        "MmeMagloire": 0.000,
(12)                        "CountessDeLo": 0.000,
(12)                        "Geborand": 0.000,
(12)                        "Champtercier": 0.000,
(12)                        "Cravatte": 0.000,
(12)                        "Count": 0.000,
(12)                        "OldMan": 0.000,
(12)                        "Valjean": 0.570,
(12)                        "Labarre": 0.000,
(12)                        "Marguerite": 0.000,
(12)                        "MmeDeR": 0.000,
(12)                        "Isabeau": 0.000,
(12)                        "Gervais": 0.000,
(12)                        "Listolier": 0.000,
(12)                        "Tholomyes": 0.041,
(12)                        "Fameuil": 0.000,
(12)                        "Blacheville": 0.000,
(12)                        "Favourite": 0.000,
(12)                        "Dahlia": 0.000,
(12)                        "Zephine": 0.000,
(12)                        "Fantine": 0.130,
(12)                        "MmeThenardier": 0.029,
(12)                        "Thenardier": 0.075,
(12)                        "Cosette": 0.024,
(12)                        "Javert": 0.054,
(12)                        "Fauchelevent": 0.026,
(12)                        "Bamatabois": 0.008,
(12)                        "Perpetue": 0.000,
(12)                        "Simplice": 0.009,
(12)                        "Scaufflaire": 0.000,
(12)                        "Woman1": 0.000,
```

```
171: (12)                        "Judge": 0.000,
172: (12)                        "Champmathieu": 0.000,
173: (12)                        "Brevet": 0.000,
174: (12)                        "Chenildieu": 0.000,
175: (12)                        "Cochepaille": 0.000,
176: (12)                        "Pontmercy": 0.007,
177: (12)                        "Boulatruelle": 0.000,
178: (12)                        "Eponine": 0.011,
179: (12)                        "Anzelma": 0.000,
180: (12)                        "Woman2": 0.000,
181: (12)                        "MotherInnocent": 0.000,
182: (12)                        "Gribier": 0.000,
183: (12)                        "MmeBurgon": 0.026,
184: (12)                        "Jondrette": 0.000,
185: (12)                        "Gavroche": 0.165,
186: (12)                        "Gillenormand": 0.020,
187: (12)                        "Magnon": 0.000,
188: (12)                        "MlleGillenormand": 0.048,
189: (12)                        "MmePontmercy": 0.000,
190: (12)                        "MlleVaubois": 0.000,
191: (12)                        "LtGillenormand": 0.000,
192: (12)                        "Marius": 0.132,
193: (12)                        "BaronessT": 0.000,
194: (12)                        "Mabeuf": 0.028,
195: (12)                        "Enjolras": 0.043,
196: (12)                        "Combeferre": 0.001,
197: (12)                        "Prouvaire": 0.000,
198: (12)                        "Feuilly": 0.001,
199: (12)                        "Courfeyrac": 0.005,
200: (12)                        "Bahorel": 0.002,
201: (12)                        "Bossuet": 0.031,
202: (12)                        "Joly": 0.002,
203: (12)                        "Grantaire": 0.000,
204: (12)                        "MotherPlutarch": 0.000,
205: (12)                        "Gueulemer": 0.005,
206: (12)                        "Babet": 0.005,
207: (12)                        "Claquesous": 0.005,
208: (12)                        "Montparnasse": 0.004,
209: (12)                        "Toussaint": 0.000,
210: (12)                        "Child1": 0.000,
211: (12)                        "Child2": 0.000,
212: (12)                        "Brujon": 0.000,
213: (12)                        "MmeHucheloup": 0.000,
214: (8)                    }
215: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=True)
216: (8)                    for n in sorted(G):
217: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
218: (4)                def test_ladder_graph(self):
219: (8)                    """Betweenness centrality: Ladder graph"""
220: (8)                    G = nx.Graph()  # ladder_graph(3)
221: (8)                    G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (2, 4), (4, 5), (3,
5)])
222: (8)                    b_answer = {0: 1.667, 1: 1.667, 2: 6.667, 3: 6.667, 4: 1.667, 5:
1.667}
223: (8)                    for b in b_answer:
224: (12)                       b_answer[b] /= 2
225: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=False)
226: (8)                    for n in sorted(G):
227: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
228: (4)                def test_disconnected_path(self):
229: (8)                    """Betweenness centrality: disconnected path"""
230: (8)                    G = nx.Graph()
231: (8)                    nx.add_path(G, [0, 1, 2])
232: (8)                    nx.add_path(G, [3, 4, 5, 6])
233: (8)                    b_answer = {0: 0, 1: 1, 2: 0, 3: 0, 4: 2, 5: 2, 6: 0}
234: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=False)
235: (8)                    for n in sorted(G):
236: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
237: (4)                def test_disconnected_path_endpoints(self):
```

```
238: (8)                    """Betweenness centrality: disconnected path endpoints"""
239: (8)                    G = nx.Graph()
240: (8)                    nx.add_path(G, [0, 1, 2])
241: (8)                    nx.add_path(G, [3, 4, 5, 6])
242: (8)                    b_answer = {0: 2, 1: 3, 2: 2, 3: 3, 4: 5, 5: 5, 6: 3}
243: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=False,
endpoints=True)
244: (8)                    for n in sorted(G):
245: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
246: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=True,
endpoints=True)
247: (8)                    for n in sorted(G):
248: (12)                       assert b[n] == pytest.approx(b_answer[n] / 21, abs=1e-7)
249: (4)                def test_directed_path(self):
250: (8)                    """Betweenness centrality: directed path"""
251: (8)                    G = nx.DiGraph()
252: (8)                    nx.add_path(G, [0, 1, 2])
253: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=False)
254: (8)                    b_answer = {0: 0.0, 1: 1.0, 2: 0.0}
255: (8)                    for n in sorted(G):
256: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
257: (4)                def test_directed_path_normalized(self):
258: (8)                    """Betweenness centrality: directed path normalized"""
259: (8)                    G = nx.DiGraph()
260: (8)                    nx.add_path(G, [0, 1, 2])
261: (8)                    b = nx.betweenness_centrality(G, weight=None, normalized=True)
262: (8)                    b_answer = {0: 0.0, 1: 0.5, 2: 0.0}
263: (8)                    for n in sorted(G):
264: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
265: (0)            class TestWeightedBetweennessCentrality:
266: (4)                def test_K5(self):
267: (8)                    """Weighted betweenness centrality: K5"""
268: (8)                    G = nx.complete_graph(5)
269: (8)                    b = nx.betweenness_centrality(G, weight="weight", normalized=False)
270: (8)                    b_answer = {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0}
271: (8)                    for n in sorted(G):
272: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
273: (4)                def test_P3_normalized(self):
274: (8)                    """Weighted betweenness centrality: P3 normalized"""
275: (8)                    G = nx.path_graph(3)
276: (8)                    b = nx.betweenness_centrality(G, weight="weight", normalized=True)
277: (8)                    b_answer = {0: 0.0, 1: 1.0, 2: 0.0}
278: (8)                    for n in sorted(G):
279: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
280: (4)                def test_P3(self):
281: (8)                    """Weighted betweenness centrality: P3"""
282: (8)                    G = nx.path_graph(3)
283: (8)                    b_answer = {0: 0.0, 1: 1.0, 2: 0.0}
284: (8)                    b = nx.betweenness_centrality(G, weight="weight", normalized=False)
285: (8)                    for n in sorted(G):
286: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
287: (4)                def test_krackhardt_kite_graph(self):
288: (8)                    """Weighted betweenness centrality: Krackhardt kite graph"""
289: (8)                    G = nx.krackhardt_kite_graph()
290: (8)                    b_answer = {
291: (12)                       0: 1.667,
292: (12)                       1: 1.667,
293: (12)                       2: 0.000,
294: (12)                       3: 7.333,
295: (12)                       4: 0.000,
296: (12)                       5: 16.667,
297: (12)                       6: 16.667,
298: (12)                       7: 28.000,
299: (12)                       8: 16.000,
300: (12)                       9: 0.000,
301: (8)                    }
302: (8)                    for b in b_answer:
303: (12)                       b_answer[b] /= 2
304: (8)                    b = nx.betweenness_centrality(G, weight="weight", normalized=False)
```

```
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_krackhardt_kite_graph_normalized(self):
(8)                     """Weighted betweenness centrality:
(8)                     Krackhardt kite graph normalized
(8)                     """
(8)                     G = nx.krackhardt_kite_graph()
(8)                     b_answer = {
(12)                        0: 0.023,
(12)                        1: 0.023,
(12)                        2: 0.000,
(12)                        3: 0.102,
(12)                        4: 0.000,
(12)                        5: 0.231,
(12)                        6: 0.231,
(12)                        7: 0.389,
(12)                        8: 0.222,
(12)                        9: 0.000,
(8)                     }
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_florentine_families_graph(self):
(8)                     """Weighted betweenness centrality:
(8)                     Florentine families graph"""
(8)                     G = nx.florentine_families_graph()
(8)                     b_answer = {
(12)                        "Acciaiuoli": 0.000,
(12)                        "Albizzi": 0.212,
(12)                        "Barbadori": 0.093,
(12)                        "Bischeri": 0.104,
(12)                        "Castellani": 0.055,
(12)                        "Ginori": 0.000,
(12)                        "Guadagni": 0.255,
(12)                        "Lamberteschi": 0.000,
(12)                        "Medici": 0.522,
(12)                        "Pazzi": 0.000,
(12)                        "Peruzzi": 0.022,
(12)                        "Ridolfi": 0.114,
(12)                        "Salviati": 0.143,
(12)                        "Strozzi": 0.103,
(12)                        "Tornabuoni": 0.092,
(8)                     }
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_les_miserables_graph(self):
(8)                     """Weighted betweenness centrality: Les Miserables graph"""
(8)                     G = nx.les_miserables_graph()
(8)                     b_answer = {
(12)                        "Napoleon": 0.000,
(12)                        "Myriel": 0.177,
(12)                        "MlleBaptistine": 0.000,
(12)                        "MmeMagloire": 0.000,
(12)                        "CountessDeLo": 0.000,
(12)                        "Geborand": 0.000,
(12)                        "Champtercier": 0.000,
(12)                        "Cravatte": 0.000,
(12)                        "Count": 0.000,
(12)                        "OldMan": 0.000,
(12)                        "Valjean": 0.454,
(12)                        "Labarre": 0.000,
(12)                        "Marguerite": 0.009,
(12)                        "MmeDeR": 0.000,
(12)                        "Isabeau": 0.000,
(12)                        "Gervais": 0.000,
(12)                        "Listolier": 0.000,
(12)                        "Tholomyes": 0.066,
(12)                        "Fameuil": 0.000,
```

```
(12)                        "Blacheville": 0.000,
(12)                        "Favourite": 0.000,
(12)                        "Dahlia": 0.000,
(12)                        "Zephine": 0.000,
(12)                        "Fantine": 0.114,
(12)                        "MmeThenardier": 0.046,
(12)                        "Thenardier": 0.129,
(12)                        "Cosette": 0.075,
(12)                        "Javert": 0.193,
(12)                        "Fauchelevent": 0.026,
(12)                        "Bamatabois": 0.080,
(12)                        "Perpetue": 0.000,
(12)                        "Simplice": 0.001,
(12)                        "Scaufflaire": 0.000,
(12)                        "Woman1": 0.000,
(12)                        "Judge": 0.000,
(12)                        "Champmathieu": 0.000,
(12)                        "Brevet": 0.000,
(12)                        "Chenildieu": 0.000,
(12)                        "Cochepaille": 0.000,
(12)                        "Pontmercy": 0.023,
(12)                        "Boulatruelle": 0.000,
(12)                        "Eponine": 0.023,
(12)                        "Anzelma": 0.000,
(12)                        "Woman2": 0.000,
(12)                        "MotherInnocent": 0.000,
(12)                        "Gribier": 0.000,
(12)                        "MmeBurgon": 0.026,
(12)                        "Jondrette": 0.000,
(12)                        "Gavroche": 0.285,
(12)                        "Gillenormand": 0.024,
(12)                        "Magnon": 0.005,
(12)                        "MlleGillenormand": 0.036,
(12)                        "MmePontmercy": 0.005,
(12)                        "MlleVaubois": 0.000,
(12)                        "LtGillenormand": 0.015,
(12)                        "Marius": 0.072,
(12)                        "BaronessT": 0.004,
(12)                        "Mabeuf": 0.089,
(12)                        "Enjolras": 0.003,
(12)                        "Combeferre": 0.000,
(12)                        "Prouvaire": 0.000,
(12)                        "Feuilly": 0.004,
(12)                        "Courfeyrac": 0.001,
(12)                        "Bahorel": 0.007,
(12)                        "Bossuet": 0.028,
(12)                        "Joly": 0.000,
(12)                        "Grantaire": 0.036,
(12)                        "MotherPlutarch": 0.000,
(12)                        "Gueulemer": 0.025,
(12)                        "Babet": 0.015,
(12)                        "Claquesous": 0.042,
(12)                        "Montparnasse": 0.050,
(12)                        "Toussaint": 0.011,
(12)                        "Child1": 0.000,
(12)                        "Child2": 0.000,
(12)                        "Brujon": 0.002,
(12)                        "MmeHucheloup": 0.034,
(8)                     }
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=True)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_ladder_graph(self):
(8)                     """Weighted betweenness centrality: Ladder graph"""
(8)                     G = nx.Graph()  # ladder_graph(3)
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (2, 4), (4, 5), (3,
5)])
(8)                     b_answer = {0: 1.667, 1: 1.667, 2: 6.667, 3: 6.667, 4: 1.667, 5:
1.667}
```

```
(8)                     for b in b_answer:
(12)                        b_answer[b] /= 2
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
(4)                 def test_G(self):
(8)                     """Weighted betweenness centrality: G"""
(8)                     G = weighted_G()
(8)                     b_answer = {0: 2.0, 1: 0.0, 2: 4.0, 3: 3.0, 4: 4.0, 5: 0.0}
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_G2(self):
(8)                     """Weighted betweenness centrality: G2"""
(8)                     G = nx.DiGraph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            ("s", "u", 10),
(16)                            ("s", "x", 5),
(16)                            ("u", "v", 1),
(16)                            ("u", "x", 2),
(16)                            ("v", "y", 1),
(16)                            ("x", "u", 3),
(16)                            ("x", "v", 5),
(16)                            ("x", "y", 2),
(16)                            ("y", "s", 7),
(16)                            ("y", "v", 6),
(12)                        ]
(8)                     )
(8)                     b_answer = {"y": 5.0, "x": 5.0, "s": 4.0, "u": 2.0, "v": 2.0}
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_G3(self):
(8)                     """Weighted betweenness centrality: G3"""
(8)                     G = nx.MultiGraph(weighted_G())
(8)                     es = list(G.edges(data=True))[::2]  # duplicate every other edge
(8)                     G.add_edges_from(es)
(8)                     b_answer = {0: 2.0, 1: 0.0, 2: 4.0, 3: 3.0, 4: 4.0, 5: 0.0}
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_G4(self):
(8)                     """Weighted betweenness centrality: G4"""
(8)                     G = nx.MultiDiGraph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            ("s", "u", 10),
(16)                            ("s", "x", 5),
(16)                            ("s", "x", 6),
(16)                            ("u", "v", 1),
(16)                            ("u", "x", 2),
(16)                            ("v", "y", 1),
(16)                            ("v", "y", 1),
(16)                            ("x", "u", 3),
(16)                            ("x", "v", 5),
(16)                            ("x", "y", 2),
(16)                            ("x", "y", 3),
(16)                            ("y", "s", 7),
(16)                            ("y", "v", 6),
(16)                            ("y", "v", 6),
(12)                        ]
(8)                     )
(8)                     b_answer = {"y": 5.0, "x": 5.0, "s": 4.0, "u": 2.0, "v": 2.0}
(8)                     b = nx.betweenness_centrality(G, weight="weight", normalized=False)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(0)             class TestEdgeBetweennessCentrality:
(4)                 def test_K5(self):
```

```
(8)                 """Edge betweenness centrality: K5"""
(8)                 G = nx.complete_graph(5)
(8)                 b = nx.edge_betweenness_centrality(G, weight=None, normalized=False)
(8)                 b_answer = dict.fromkeys(G.edges(), 1)
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)             def test_normalized_K5(self):
(8)                 """Edge betweenness centrality: K5"""
(8)                 G = nx.complete_graph(5)
(8)                 b = nx.edge_betweenness_centrality(G, weight=None, normalized=True)
(8)                 b_answer = dict.fromkeys(G.edges(), 1 / 10)
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)             def test_C4(self):
(8)                 """Edge betweenness centrality: C4"""
(8)                 G = nx.cycle_graph(4)
(8)                 b = nx.edge_betweenness_centrality(G, weight=None, normalized=True)
(8)                 b_answer = {(0, 1): 2, (0, 3): 2, (1, 2): 2, (2, 3): 2}
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n] / 6, abs=1e-7)
(4)             def test_P4(self):
(8)                 """Edge betweenness centrality: P4"""
(8)                 G = nx.path_graph(4)
(8)                 b = nx.edge_betweenness_centrality(G, weight=None, normalized=False)
(8)                 b_answer = {(0, 1): 3, (1, 2): 4, (2, 3): 3}
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)             def test_normalized_P4(self):
(8)                 """Edge betweenness centrality: P4"""
(8)                 G = nx.path_graph(4)
(8)                 b = nx.edge_betweenness_centrality(G, weight=None, normalized=True)
(8)                 b_answer = {(0, 1): 3, (1, 2): 4, (2, 3): 3}
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n] / 6, abs=1e-7)
(4)             def test_balanced_tree(self):
(8)                 """Edge betweenness centrality: balanced tree"""
(8)                 G = nx.balanced_tree(r=2, h=2)
(8)                 b = nx.edge_betweenness_centrality(G, weight=None, normalized=False)
(8)                 b_answer = {(0, 1): 12, (0, 2): 12, (1, 3): 6, (1, 4): 6, (2, 5): 6,
(2, 6): 6}
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(0)         class TestWeightedEdgeBetweennessCentrality:
(4)             def test_K5(self):
(8)                 """Edge betweenness centrality: K5"""
(8)                 G = nx.complete_graph(5)
(8)                 b = nx.edge_betweenness_centrality(G, weight="weight",
normalized=False)
(8)                 b_answer = dict.fromkeys(G.edges(), 1)
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)             def test_C4(self):
(8)                 """Edge betweenness centrality: C4"""
(8)                 G = nx.cycle_graph(4)
(8)                 b = nx.edge_betweenness_centrality(G, weight="weight",
normalized=False)
(8)                 b_answer = {(0, 1): 2, (0, 3): 2, (1, 2): 2, (2, 3): 2}
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)             def test_P4(self):
(8)                 """Edge betweenness centrality: P4"""
(8)                 G = nx.path_graph(4)
(8)                 b = nx.edge_betweenness_centrality(G, weight="weight",
normalized=False)
(8)                 b_answer = {(0, 1): 3, (1, 2): 4, (2, 3): 3}
(8)                 for n in sorted(G.edges()):
(12)                    assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)             def test_balanced_tree(self):
(8)                 """Edge betweenness centrality: balanced tree"""
```

```
        G = nx.balanced_tree(r=2, h=2)
        b = nx.edge_betweenness_centrality(G, weight="weight", normalized=False)
        b_answer = {(0, 1): 12, (0, 2): 12, (1, 3): 6, (1, 4): 6, (2, 5): 6, (2, 6): 6}
        for n in sorted(G.edges()):
            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
    def test_weighted_graph(self):
        """Edge betweenness centrality: weighted"""
        eList = [
            (0, 1, 5),
            (0, 2, 4),
            (0, 3, 3),
            (0, 4, 2),
            (1, 2, 4),
            (1, 3, 1),
            (1, 4, 3),
            (2, 4, 5),
            (3, 4, 4),
        ]
        G = nx.Graph()
        G.add_weighted_edges_from(eList)
        b = nx.edge_betweenness_centrality(G, weight="weight", normalized=False)
        b_answer = {
            (0, 1): 0.0,
            (0, 2): 1.0,
            (0, 3): 2.0,
            (0, 4): 1.0,
            (1, 2): 2.0,
            (1, 3): 3.5,
            (1, 4): 1.5,
            (2, 4): 1.0,
            (3, 4): 0.5,
        }
        for n in sorted(G.edges()):
            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
    def test_normalized_weighted_graph(self):
        """Edge betweenness centrality: normalized weighted"""
        eList = [
            (0, 1, 5),
            (0, 2, 4),
            (0, 3, 3),
            (0, 4, 2),
            (1, 2, 4),
            (1, 3, 1),
            (1, 4, 3),
            (2, 4, 5),
            (3, 4, 4),
        ]
        G = nx.Graph()
        G.add_weighted_edges_from(eList)
        b = nx.edge_betweenness_centrality(G, weight="weight", normalized=True)
        b_answer = {
            (0, 1): 0.0,
            (0, 2): 1.0,
            (0, 3): 2.0,
            (0, 4): 1.0,
            (1, 2): 2.0,
            (1, 3): 3.5,
            (1, 4): 1.5,
            (2, 4): 1.0,
            (3, 4): 0.5,
        }
        norm = len(G) * (len(G) - 1) / 2
        for n in sorted(G.edges()):
            assert b[n] == pytest.approx(b_answer[n] / norm, abs=1e-7)
    def test_weighted_multigraph(self):
```

```
640: (8)                    """Edge betweenness centrality: weighted multigraph"""
641: (8)                    eList = [
642: (12)                       (0, 1, 5),
643: (12)                       (0, 1, 4),
644: (12)                       (0, 2, 4),
645: (12)                       (0, 3, 3),
646: (12)                       (0, 3, 3),
647: (12)                       (0, 4, 2),
648: (12)                       (1, 2, 4),
649: (12)                       (1, 3, 1),
650: (12)                       (1, 3, 2),
651: (12)                       (1, 4, 3),
652: (12)                       (1, 4, 4),
653: (12)                       (2, 4, 5),
654: (12)                       (3, 4, 4),
655: (12)                       (3, 4, 4),
656: (8)                    ]
657: (8)                    G = nx.MultiGraph()
658: (8)                    G.add_weighted_edges_from(eList)
659: (8)                    b = nx.edge_betweenness_centrality(G, weight="weight",
normalized=False)
660: (8)                    b_answer = {
661: (12)                       (0, 1, 0): 0.0,
662: (12)                       (0, 1, 1): 0.5,
663: (12)                       (0, 2, 0): 1.0,
664: (12)                       (0, 3, 0): 0.75,
665: (12)                       (0, 3, 1): 0.75,
666: (12)                       (0, 4, 0): 1.0,
667: (12)                       (1, 2, 0): 2.0,
668: (12)                       (1, 3, 0): 3.0,
669: (12)                       (1, 3, 1): 0.0,
670: (12)                       (1, 4, 0): 1.5,
671: (12)                       (1, 4, 1): 0.0,
672: (12)                       (2, 4, 0): 1.0,
673: (12)                       (3, 4, 0): 0.25,
674: (12)                       (3, 4, 1): 0.25,
675: (8)                    }
676: (8)                    for n in sorted(G.edges(keys=True)):
677: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
678: (4)                def test_normalized_weighted_multigraph(self):
679: (8)                    """Edge betweenness centrality: normalized weighted multigraph"""
680: (8)                    eList = [
681: (12)                       (0, 1, 5),
682: (12)                       (0, 1, 4),
683: (12)                       (0, 2, 4),
684: (12)                       (0, 3, 3),
685: (12)                       (0, 3, 3),
686: (12)                       (0, 4, 2),
687: (12)                       (1, 2, 4),
688: (12)                       (1, 3, 1),
689: (12)                       (1, 3, 2),
690: (12)                       (1, 4, 3),
691: (12)                       (1, 4, 4),
692: (12)                       (2, 4, 5),
693: (12)                       (3, 4, 4),
694: (12)                       (3, 4, 4),
695: (8)                    ]
696: (8)                    G = nx.MultiGraph()
697: (8)                    G.add_weighted_edges_from(eList)
698: (8)                    b = nx.edge_betweenness_centrality(G, weight="weight",
normalized=True)
699: (8)                    b_answer = {
700: (12)                       (0, 1, 0): 0.0,
701: (12)                       (0, 1, 1): 0.5,
702: (12)                       (0, 2, 0): 1.0,
703: (12)                       (0, 3, 0): 0.75,
704: (12)                       (0, 3, 1): 0.75,
705: (12)                       (0, 4, 0): 1.0,
706: (12)                       (1, 2, 0): 2.0,
```

```
707: (12)                        (1, 3, 0): 3.0,
708: (12)                        (1, 3, 1): 0.0,
709: (12)                        (1, 4, 0): 1.5,
710: (12)                        (1, 4, 1): 0.0,
711: (12)                        (2, 4, 0): 1.0,
712: (12)                        (3, 4, 0): 0.25,
713: (12)                        (3, 4, 1): 0.25,
714: (8)                     }
715: (8)                     norm = len(G) * (len(G) - 1) / 2
716: (8)                     for n in sorted(G.edges(keys=True)):
717: (12)                        assert b[n] == pytest.approx(b_answer[n] / norm, abs=1e-7)
```

File 155 - test_betweenness_centrality_subset.py:

```
1: (0)               import pytest
2: (0)               import networkx as nx
3: (0)               class TestSubsetBetweennessCentrality:
4: (4)                   def test_K5(self):
5: (8)                       """Betweenness Centrality Subset: K5"""
6: (8)                       G = nx.complete_graph(5)
7: (8)                       b = nx.betweenness_centrality_subset(
8: (12)                          G, sources=[0], targets=[1, 3], weight=None
9: (8)                       )
10: (8)                      b_answer = {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0}
11: (8)                      for n in sorted(G):
12: (12)                         assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
13: (4)                  def test_P5_directed(self):
14: (8)                      """Betweenness Centrality Subset: P5 directed"""
15: (8)                      G = nx.DiGraph()
16: (8)                      nx.add_path(G, range(5))
17: (8)                      b_answer = {0: 0, 1: 1, 2: 1, 3: 0, 4: 0, 5: 0}
18: (8)                      b = nx.betweenness_centrality_subset(G, sources=[0], targets=[3],
weight=None)
19: (8)                      for n in sorted(G):
20: (12)                         assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
21: (4)                  def test_P5(self):
22: (8)                      """Betweenness Centrality Subset: P5"""
23: (8)                      G = nx.Graph()
24: (8)                      nx.add_path(G, range(5))
25: (8)                      b_answer = {0: 0, 1: 0.5, 2: 0.5, 3: 0, 4: 0, 5: 0}
26: (8)                      b = nx.betweenness_centrality_subset(G, sources=[0], targets=[3],
weight=None)
27: (8)                      for n in sorted(G):
28: (12)                         assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
29: (4)                  def test_P5_multiple_target(self):
30: (8)                      """Betweenness Centrality Subset: P5 multiple target"""
31: (8)                      G = nx.Graph()
32: (8)                      nx.add_path(G, range(5))
33: (8)                      b_answer = {0: 0, 1: 1, 2: 1, 3: 0.5, 4: 0, 5: 0}
34: (8)                      b = nx.betweenness_centrality_subset(
35: (12)                         G, sources=[0], targets=[3, 4], weight=None
36: (8)                      )
37: (8)                      for n in sorted(G):
38: (12)                         assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
39: (4)                  def test_box(self):
40: (8)                      """Betweenness Centrality Subset: box"""
41: (8)                      G = nx.Graph()
42: (8)                      G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
43: (8)                      b_answer = {0: 0, 1: 0.25, 2: 0.25, 3: 0}
44: (8)                      b = nx.betweenness_centrality_subset(G, sources=[0], targets=[3],
weight=None)
45: (8)                      for n in sorted(G):
46: (12)                         assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
47: (4)                  def test_box_and_path(self):
48: (8)                      """Betweenness Centrality Subset: box and path"""
49: (8)                      G = nx.Graph()
50: (8)                      G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (3, 4), (4, 5)])
```

```
51: (8)                     b_answer = {0: 0, 1: 0.5, 2: 0.5, 3: 0.5, 4: 0, 5: 0}
52: (8)                     b = nx.betweenness_centrality_subset(
53: (12)                        G, sources=[0], targets=[3, 4], weight=None
54: (8)                     )
55: (8)                     for n in sorted(G):
56: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
57: (4)                 def test_box_and_path2(self):
58: (8)                     """Betweenness Centrality Subset: box and path multiple target"""
59: (8)                     G = nx.Graph()
60: (8)                     G.add_edges_from([(0, 1), (1, 2), (2, 3), (1, 20), (20, 3), (3, 4)])
61: (8)                     b_answer = {0: 0, 1: 1.0, 2: 0.5, 20: 0.5, 3: 0.5, 4: 0}
62: (8)                     b = nx.betweenness_centrality_subset(
63: (12)                        G, sources=[0], targets=[3, 4], weight=None
64: (8)                     )
65: (8)                     for n in sorted(G):
66: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
67: (4)                 def test_diamond_multi_path(self):
68: (8)                     """Betweenness Centrality Subset: Diamond Multi Path"""
69: (8)                     G = nx.Graph()
70: (8)                     G.add_edges_from(
71: (12)                        [
72: (16)                            (1, 2),
73: (16)                            (1, 3),
74: (16)                            (1, 4),
75: (16)                            (1, 5),
76: (16)                            (1, 10),
77: (16)                            (10, 11),
78: (16)                            (11, 12),
79: (16)                            (12, 9),
80: (16)                            (2, 6),
81: (16)                            (3, 6),
82: (16)                            (4, 6),
83: (16)                            (5, 7),
84: (16)                            (7, 8),
85: (16)                            (6, 8),
86: (16)                            (8, 9),
87: (12)                        ]
88: (8)                     )
89: (8)                     b = nx.betweenness_centrality_subset(G, sources=[1], targets=[9],
weight=None)
90: (8)                     expected_b = {
91: (12)                        1: 0,
92: (12)                        2: 1.0 / 10,
93: (12)                        3: 1.0 / 10,
94: (12)                        4: 1.0 / 10,
95: (12)                        5: 1.0 / 10,
96: (12)                        6: 3.0 / 10,
97: (12)                        7: 1.0 / 10,
98: (12)                        8: 4.0 / 10,
99: (12)                        9: 0,
100: (12)                       10: 1.0 / 10,
101: (12)                       11: 1.0 / 10,
102: (12)                       12: 1.0 / 10,
103: (8)                    }
104: (8)                    for n in sorted(G):
105: (12)                       assert b[n] == pytest.approx(expected_b[n], abs=1e-7)
106: (4)                def test_normalized_p2(self):
107: (8)                    """
108: (8)                    Betweenness Centrality Subset: Normalized P2
109: (8)                    if n <= 2:  no normalization, betweenness centrality should be 0 for
all nodes.
110: (8)                    """
111: (8)                    G = nx.Graph()
112: (8)                    nx.add_path(G, range(2))
113: (8)                    b_answer = {0: 0, 1: 0.0}
114: (8)                    b = nx.betweenness_centrality_subset(
115: (12)                       G, sources=[0], targets=[1], normalized=True, weight=None
116: (8)                    )
117: (8)                    for n in sorted(G):
```

```
118: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
119: (4)                 def test_normalized_P5_directed(self):
120: (8)                     """Betweenness Centrality Subset: Normalized Directed P5"""
121: (8)                     G = nx.DiGraph()
122: (8)                     nx.add_path(G, range(5))
123: (8)                     b_answer = {0: 0, 1: 1.0 / 12.0, 2: 1.0 / 12.0, 3: 0, 4: 0, 5: 0}
124: (8)                     b = nx.betweenness_centrality_subset(
125: (12)                        G, sources=[0], targets=[3], normalized=True, weight=None
126: (8)                     )
127: (8)                     for n in sorted(G):
128: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
129: (4)                 def test_weighted_graph(self):
130: (8)                     """Betweenness Centrality Subset: Weighted Graph"""
131: (8)                     G = nx.DiGraph()
132: (8)                     G.add_edge(0, 1, weight=3)
133: (8)                     G.add_edge(0, 2, weight=2)
134: (8)                     G.add_edge(0, 3, weight=6)
135: (8)                     G.add_edge(0, 4, weight=4)
136: (8)                     G.add_edge(1, 3, weight=5)
137: (8)                     G.add_edge(1, 5, weight=5)
138: (8)                     G.add_edge(2, 4, weight=1)
139: (8)                     G.add_edge(3, 4, weight=2)
140: (8)                     G.add_edge(3, 5, weight=1)
141: (8)                     G.add_edge(4, 5, weight=4)
142: (8)                     b_answer = {0: 0.0, 1: 0.0, 2: 0.5, 3: 0.5, 4: 0.5, 5: 0.0}
143: (8)                     b = nx.betweenness_centrality_subset(
144: (12)                        G, sources=[0], targets=[5], normalized=False, weight="weight"
145: (8)                     )
146: (8)                     for n in sorted(G):
147: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
148: (0)             class TestEdgeSubsetBetweennessCentrality:
149: (4)                 def test_K5(self):
150: (8)                     """Edge betweenness subset centrality: K5"""
151: (8)                     G = nx.complete_graph(5)
152: (8)                     b = nx.edge_betweenness_centrality_subset(
153: (12)                        G, sources=[0], targets=[1, 3], weight=None
154: (8)                     )
155: (8)                     b_answer = dict.fromkeys(G.edges(), 0)
156: (8)                     b_answer[(0, 3)] = b_answer[(0, 1)] = 0.5
157: (8)                     for n in sorted(G.edges()):
158: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
159: (4)                 def test_P5_directed(self):
160: (8)                     """Edge betweenness subset centrality: P5 directed"""
161: (8)                     G = nx.DiGraph()
162: (8)                     nx.add_path(G, range(5))
163: (8)                     b_answer = dict.fromkeys(G.edges(), 0)
164: (8)                     b_answer[(0, 1)] = b_answer[(1, 2)] = b_answer[(2, 3)] = 1
165: (8)                     b = nx.edge_betweenness_centrality_subset(
166: (12)                        G, sources=[0], targets=[3], weight=None
167: (8)                     )
168: (8)                     for n in sorted(G.edges()):
169: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
170: (4)                 def test_P5(self):
171: (8)                     """Edge betweenness subset centrality: P5"""
172: (8)                     G = nx.Graph()
173: (8)                     nx.add_path(G, range(5))
174: (8)                     b_answer = dict.fromkeys(G.edges(), 0)
175: (8)                     b_answer[(0, 1)] = b_answer[(1, 2)] = b_answer[(2, 3)] = 0.5
176: (8)                     b = nx.edge_betweenness_centrality_subset(
177: (12)                        G, sources=[0], targets=[3], weight=None
178: (8)                     )
179: (8)                     for n in sorted(G.edges()):
180: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
181: (4)                 def test_P5_multiple_target(self):
182: (8)                     """Edge betweenness subset centrality: P5 multiple target"""
183: (8)                     G = nx.Graph()
184: (8)                     nx.add_path(G, range(5))
185: (8)                     b_answer = dict.fromkeys(G.edges(), 0)
186: (8)                     b_answer[(0, 1)] = b_answer[(1, 2)] = b_answer[(2, 3)] = 1
```

```
(8)                     b_answer[(3, 4)] = 0.5
(8)                     b = nx.edge_betweenness_centrality_subset(
(12)                        G, sources=[0], targets=[3, 4], weight=None
(8)                     )
(8)                     for n in sorted(G.edges()):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_box(self):
(8)                     """Edge betweenness subset centrality: box"""
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     b_answer = dict.fromkeys(G.edges(), 0)
(8)                     b_answer[(0, 1)] = b_answer[(0, 2)] = 0.25
(8)                     b_answer[(1, 3)] = b_answer[(2, 3)] = 0.25
(8)                     b = nx.edge_betweenness_centrality_subset(
(12)                        G, sources=[0], targets=[3], weight=None
(8)                     )
(8)                     for n in sorted(G.edges()):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_box_and_path(self):
(8)                     """Edge betweenness subset centrality: box and path"""
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (3, 4), (4, 5)])
(8)                     b_answer = dict.fromkeys(G.edges(), 0)
(8)                     b_answer[(0, 1)] = b_answer[(0, 2)] = 0.5
(8)                     b_answer[(1, 3)] = b_answer[(2, 3)] = 0.5
(8)                     b_answer[(3, 4)] = 0.5
(8)                     b = nx.edge_betweenness_centrality_subset(
(12)                        G, sources=[0], targets=[3, 4], weight=None
(8)                     )
(8)                     for n in sorted(G.edges()):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_box_and_path2(self):
(8)                     """Edge betweenness subset centrality: box and path multiple target"""
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (1, 2), (2, 3), (1, 20), (20, 3), (3, 4)])
(8)                     b_answer = dict.fromkeys(G.edges(), 0)
(8)                     b_answer[(0, 1)] = 1.0
(8)                     b_answer[(1, 20)] = b_answer[(3, 20)] = 0.5
(8)                     b_answer[(1, 2)] = b_answer[(2, 3)] = 0.5
(8)                     b_answer[(3, 4)] = 0.5
(8)                     b = nx.edge_betweenness_centrality_subset(
(12)                        G, sources=[0], targets=[3, 4], weight=None
(8)                     )
(8)                     for n in sorted(G.edges()):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
(4)                 def test_diamond_multi_path(self):
(8)                     """Edge betweenness subset centrality: Diamond Multi Path"""
(8)                     G = nx.Graph()
(8)                     G.add_edges_from(
(12)                        [
(16)                            (1, 2),
(16)                            (1, 3),
(16)                            (1, 4),
(16)                            (1, 5),
(16)                            (1, 10),
(16)                            (10, 11),
(16)                            (11, 12),
(16)                            (12, 9),
(16)                            (2, 6),
(16)                            (3, 6),
(16)                            (4, 6),
(16)                            (5, 7),
(16)                            (7, 8),
(16)                            (6, 8),
(16)                            (8, 9),
(12)                        ]
(8)                     )
(8)                     b_answer = dict.fromkeys(G.edges(), 0)
(8)                     b_answer[(8, 9)] = 0.4
```

```
256: (8)                    b_answer[(6, 8)] = b_answer[(7, 8)] = 0.2
257: (8)                    b_answer[(2, 6)] = b_answer[(3, 6)] = b_answer[(4, 6)] = 0.2 / 3.0
258: (8)                    b_answer[(1, 2)] = b_answer[(1, 3)] = b_answer[(1, 4)] = 0.2 / 3.0
259: (8)                    b_answer[(5, 7)] = 0.2
260: (8)                    b_answer[(1, 5)] = 0.2
261: (8)                    b_answer[(9, 12)] = 0.1
262: (8)                    b_answer[(11, 12)] = b_answer[(10, 11)] = b_answer[(1, 10)] = 0.1
263: (8)                    b = nx.edge_betweenness_centrality_subset(
264: (12)                       G, sources=[1], targets=[9], weight=None
265: (8)                    )
266: (8)                    for n in G.edges():
267: (12)                       sort_n = tuple(sorted(n))
268: (12)                       assert b[n] == pytest.approx(b_answer[sort_n], abs=1e-7)
269: (4)                def test_normalized_p1(self):
270: (8)                    """
271: (8)                    Edge betweenness subset centrality: P1
272: (8)                    if n <= 1: no normalization b=0 for all nodes
273: (8)                    """
274: (8)                    G = nx.Graph()
275: (8)                    nx.add_path(G, range(1))
276: (8)                    b_answer = dict.fromkeys(G.edges(), 0)
277: (8)                    b = nx.edge_betweenness_centrality_subset(
278: (12)                       G, sources=[0], targets=[0], normalized=True, weight=None
279: (8)                    )
280: (8)                    for n in G.edges():
281: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
282: (4)                def test_normalized_P5_directed(self):
283: (8)                    """Edge betweenness subset centrality: Normalized Directed P5"""
284: (8)                    G = nx.DiGraph()
285: (8)                    nx.add_path(G, range(5))
286: (8)                    b_answer = dict.fromkeys(G.edges(), 0)
287: (8)                    b_answer[(0, 1)] = b_answer[(1, 2)] = b_answer[(2, 3)] = 0.05
288: (8)                    b = nx.edge_betweenness_centrality_subset(
289: (12)                       G, sources=[0], targets=[3], normalized=True, weight=None
290: (8)                    )
291: (8)                    for n in G.edges():
292: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
293: (4)                def test_weighted_graph(self):
294: (8)                    """Edge betweenness subset centrality: Weighted Graph"""
295: (8)                    G = nx.DiGraph()
296: (8)                    G.add_edge(0, 1, weight=3)
297: (8)                    G.add_edge(0, 2, weight=2)
298: (8)                    G.add_edge(0, 3, weight=6)
299: (8)                    G.add_edge(0, 4, weight=4)
300: (8)                    G.add_edge(1, 3, weight=5)
301: (8)                    G.add_edge(1, 5, weight=5)
302: (8)                    G.add_edge(2, 4, weight=1)
303: (8)                    G.add_edge(3, 4, weight=2)
304: (8)                    G.add_edge(3, 5, weight=1)
305: (8)                    G.add_edge(4, 5, weight=4)
306: (8)                    b_answer = dict.fromkeys(G.edges(), 0)
307: (8)                    b_answer[(0, 2)] = b_answer[(2, 4)] = b_answer[(4, 5)] = 0.5
308: (8)                    b_answer[(0, 3)] = b_answer[(3, 5)] = 0.5
309: (8)                    b = nx.edge_betweenness_centrality_subset(
310: (12)                       G, sources=[0], targets=[5], normalized=False, weight="weight"
311: (8)                    )
312: (8)                    for n in G.edges():
313: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)

----------------------------------------

File 156 - test_closeness_centrality.py:

1: (0)              """
2: (0)              Tests for closeness centrality.
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              class TestClosenessCentrality:
```

```
    @classmethod
    def setup_class(cls):
        cls.K = nx.krackhardt_kite_graph()
        cls.P3 = nx.path_graph(3)
        cls.P4 = nx.path_graph(4)
        cls.K5 = nx.complete_graph(5)
        cls.C4 = nx.cycle_graph(4)
        cls.T = nx.balanced_tree(r=2, h=2)
        cls.Gb = nx.Graph()
        cls.Gb.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (2, 4), (4, 5), (3, 5)])

        F = nx.florentine_families_graph()
        cls.F = F
        cls.LM = nx.les_miserables_graph()
        cls.undirected_G = nx.fast_gnp_random_graph(n=100, p=0.6, seed=123)
        cls.undirected_G_cc = nx.closeness_centrality(cls.undirected_G)
    def test_wf_improved(self):
        G = nx.union(self.P4, nx.path_graph([4, 5, 6]))
        c = nx.closeness_centrality(G)
        cwf = nx.closeness_centrality(G, wf_improved=False)
        res = {0: 0.25, 1: 0.375, 2: 0.375, 3: 0.25, 4: 0.222, 5: 0.333, 6: 0.222}

        wf_res = {0: 0.5, 1: 0.75, 2: 0.75, 3: 0.5, 4: 0.667, 5: 1.0, 6: 0.667}

        for n in G:
            assert c[n] == pytest.approx(res[n], abs=1e-3)
            assert cwf[n] == pytest.approx(wf_res[n], abs=1e-3)
    def test_digraph(self):
        G = nx.path_graph(3, create_using=nx.DiGraph())
        c = nx.closeness_centrality(G)
        cr = nx.closeness_centrality(G.reverse())
        d = {0: 0.0, 1: 0.500, 2: 0.667}
        dr = {0: 0.667, 1: 0.500, 2: 0.0}
        for n in sorted(self.P3):
            assert c[n] == pytest.approx(d[n], abs=1e-3)
            assert cr[n] == pytest.approx(dr[n], abs=1e-3)
    def test_k5_closeness(self):
        c = nx.closeness_centrality(self.K5)
        d = {0: 1.000, 1: 1.000, 2: 1.000, 3: 1.000, 4: 1.000}
        for n in sorted(self.K5):
            assert c[n] == pytest.approx(d[n], abs=1e-3)
    def test_p3_closeness(self):
        c = nx.closeness_centrality(self.P3)
        d = {0: 0.667, 1: 1.000, 2: 0.667}
        for n in sorted(self.P3):
            assert c[n] == pytest.approx(d[n], abs=1e-3)
    def test_krackhardt_closeness(self):
        c = nx.closeness_centrality(self.K)
        d = {
            0: 0.529,
            1: 0.529,
            2: 0.500,
            3: 0.600,
            4: 0.500,
            5: 0.643,
            6: 0.643,
            7: 0.600,
            8: 0.429,
            9: 0.310,
        }
        for n in sorted(self.K):
            assert c[n] == pytest.approx(d[n], abs=1e-3)
    def test_florentine_families_closeness(self):
        c = nx.closeness_centrality(self.F)
        d = {
            "Acciaiuoli": 0.368,
            "Albizzi": 0.483,
            "Barbadori": 0.4375,
            "Bischeri": 0.400,
```

```
(12)                        "Castellani": 0.389,
(12)                        "Ginori": 0.333,
(12)                        "Guadagni": 0.467,
(12)                        "Lamberteschi": 0.326,
(12)                        "Medici": 0.560,
(12)                        "Pazzi": 0.286,
(12)                        "Peruzzi": 0.368,
(12)                        "Ridolfi": 0.500,
(12)                        "Salviati": 0.389,
(12)                        "Strozzi": 0.4375,
(12)                        "Tornabuoni": 0.483,
(8)                     }
(8)                     for n in sorted(self.F):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_les_miserables_closeness(self):
(8)                     c = nx.closeness_centrality(self.LM)
(8)                     d = {
(12)                        "Napoleon": 0.302,
(12)                        "Myriel": 0.429,
(12)                        "MlleBaptistine": 0.413,
(12)                        "MmeMagloire": 0.413,
(12)                        "CountessDeLo": 0.302,
(12)                        "Geborand": 0.302,
(12)                        "Champtercier": 0.302,
(12)                        "Cravatte": 0.302,
(12)                        "Count": 0.302,
(12)                        "OldMan": 0.302,
(12)                        "Valjean": 0.644,
(12)                        "Labarre": 0.394,
(12)                        "Marguerite": 0.413,
(12)                        "MmeDeR": 0.394,
(12)                        "Isabeau": 0.394,
(12)                        "Gervais": 0.394,
(12)                        "Listolier": 0.341,
(12)                        "Tholomyes": 0.392,
(12)                        "Fameuil": 0.341,
(12)                        "Blacheville": 0.341,
(12)                        "Favourite": 0.341,
(12)                        "Dahlia": 0.341,
(12)                        "Zephine": 0.341,
(12)                        "Fantine": 0.461,
(12)                        "MmeThenardier": 0.461,
(12)                        "Thenardier": 0.517,
(12)                        "Cosette": 0.478,
(12)                        "Javert": 0.517,
(12)                        "Fauchelevent": 0.402,
(12)                        "Bamatabois": 0.427,
(12)                        "Perpetue": 0.318,
(12)                        "Simplice": 0.418,
(12)                        "Scaufflaire": 0.394,
(12)                        "Woman1": 0.396,
(12)                        "Judge": 0.404,
(12)                        "Champmathieu": 0.404,
(12)                        "Brevet": 0.404,
(12)                        "Chenildieu": 0.404,
(12)                        "Cochepaille": 0.404,
(12)                        "Pontmercy": 0.373,
(12)                        "Boulatruelle": 0.342,
(12)                        "Eponine": 0.396,
(12)                        "Anzelma": 0.352,
(12)                        "Woman2": 0.402,
(12)                        "MotherInnocent": 0.398,
(12)                        "Gribier": 0.288,
(12)                        "MmeBurgon": 0.344,
(12)                        "Jondrette": 0.257,
(12)                        "Gavroche": 0.514,
(12)                        "Gillenormand": 0.442,
(12)                        "Magnon": 0.335,
(12)                        "MlleGillenormand": 0.442,
```

```
(12)                        "MmePontmercy": 0.315,
(12)                        "MlleVaubois": 0.308,
(12)                        "LtGillenormand": 0.365,
(12)                        "Marius": 0.531,
(12)                        "BaronessT": 0.352,
(12)                        "Mabeuf": 0.396,
(12)                        "Enjolras": 0.481,
(12)                        "Combeferre": 0.392,
(12)                        "Prouvaire": 0.357,
(12)                        "Feuilly": 0.392,
(12)                        "Courfeyrac": 0.400,
(12)                        "Bahorel": 0.394,
(12)                        "Bossuet": 0.475,
(12)                        "Joly": 0.394,
(12)                        "Grantaire": 0.358,
(12)                        "MotherPlutarch": 0.285,
(12)                        "Gueulemer": 0.463,
(12)                        "Babet": 0.463,
(12)                        "Claquesous": 0.452,
(12)                        "Montparnasse": 0.458,
(12)                        "Toussaint": 0.402,
(12)                        "Child1": 0.342,
(12)                        "Child2": 0.342,
(12)                        "Brujon": 0.380,
(12)                        "MmeHucheloup": 0.353,
(8)                     }
(8)                     for n in sorted(self.LM):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_weighted_closeness(self):
(8)                     edges = [
(12)                        ("s", "u", 10),
(12)                        ("s", "x", 5),
(12)                        ("u", "v", 1),
(12)                        ("u", "x", 2),
(12)                        ("v", "y", 1),
(12)                        ("x", "u", 3),
(12)                        ("x", "v", 5),
(12)                        ("x", "y", 2),
(12)                        ("y", "s", 7),
(12)                        ("y", "v", 6),
(8)                     ]
(8)                     XG = nx.Graph()
(8)                     XG.add_weighted_edges_from(edges)
(8)                     c = nx.closeness_centrality(XG, distance="weight")
(8)                     d = {"y": 0.200, "x": 0.286, "s": 0.138, "u": 0.235, "v": 0.200}
(8)                     for n in sorted(XG):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 @staticmethod
(4)                 def pick_add_edge(g):
(8)                     u = nx.utils.arbitrary_element(g)
(8)                     possible_nodes = set(g.nodes())
(8)                     neighbors = list(g.neighbors(u)) + [u]
(8)                     possible_nodes.difference_update(neighbors)
(8)                     v = nx.utils.arbitrary_element(possible_nodes)
(8)                     return (u, v)
(4)                 @staticmethod
(4)                 def pick_remove_edge(g):
(8)                     u = nx.utils.arbitrary_element(g)
(8)                     possible_nodes = list(g.neighbors(u))
(8)                     v = nx.utils.arbitrary_element(possible_nodes)
(8)                     return (u, v)
(4)                 def test_directed_raises(self):
(8)                     with pytest.raises(nx.NetworkXNotImplemented):
(12)                        dir_G = nx.gn_graph(n=5)
(12)                        prev_cc = None
(12)                        edge = self.pick_add_edge(dir_G)
(12)                        insert = True
(12)                        nx.incremental_closeness_centrality(dir_G, edge, prev_cc, insert)
(4)                 def test_wrong_size_prev_cc_raises(self):
```

```
211: (8)                    with pytest.raises(nx.NetworkXError):
212: (12)                       G = self.undirected_G.copy()
213: (12)                       edge = self.pick_add_edge(G)
214: (12)                       insert = True
215: (12)                       prev_cc = self.undirected_G_cc.copy()
216: (12)                       prev_cc.pop(0)
217: (12)                       nx.incremental_closeness_centrality(G, edge, prev_cc, insert)
218: (4)                def test_wrong_nodes_prev_cc_raises(self):
219: (8)                    with pytest.raises(nx.NetworkXError):
220: (12)                       G = self.undirected_G.copy()
221: (12)                       edge = self.pick_add_edge(G)
222: (12)                       insert = True
223: (12)                       prev_cc = self.undirected_G_cc.copy()
224: (12)                       num_nodes = len(prev_cc)
225: (12)                       prev_cc.pop(0)
226: (12)                       prev_cc[num_nodes] = 0.5
227: (12)                       nx.incremental_closeness_centrality(G, edge, prev_cc, insert)
228: (4)                def test_zero_centrality(self):
229: (8)                    G = nx.path_graph(3)
230: (8)                    prev_cc = nx.closeness_centrality(G)
231: (8)                    edge = self.pick_remove_edge(G)
232: (8)                    test_cc = nx.incremental_closeness_centrality(G, edge, prev_cc,
insertion=False)
233: (8)                    G.remove_edges_from([edge])
234: (8)                    real_cc = nx.closeness_centrality(G)
235: (8)                    shared_items = set(test_cc.items()) & set(real_cc.items())
236: (8)                    assert len(shared_items) == len(real_cc)
237: (8)                    assert 0 in test_cc.values()
238: (4)                def test_incremental(self):
239: (8)                    G = self.undirected_G.copy()
240: (8)                    prev_cc = None
241: (8)                    for i in range(5):
242: (12)                       if i % 2 == 0:
243: (16)                           insert = False
244: (16)                           edge = self.pick_remove_edge(G)
245: (12)                       else:
246: (16)                           insert = True
247: (16)                           edge = self.pick_add_edge(G)
248: (12)                       test_cc = nx.incremental_closeness_centrality(G, edge, prev_cc,
insert)
249: (12)                       if insert:
250: (16)                           G.add_edges_from([edge])
251: (12)                       else:
252: (16)                           G.remove_edges_from([edge])
253: (12)                       real_cc = nx.closeness_centrality(G)
254: (12)                       assert set(test_cc.items()) == set(real_cc.items())
255: (12)                       prev_cc = test_cc
```

File 157 - test_current_flow_betweenness_centrality.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import approximate_current_flow_betweenness_centrality as
approximate_cfbc
4: (0)              from networkx import edge_current_flow_betweenness_centrality as
edge_current_flow
5: (0)              np = pytest.importorskip("numpy")
6: (0)              pytest.importorskip("scipy")
7: (0)              class TestFlowBetweennessCentrality:
8: (4)                  def test_K4_normalized(self):
9: (8)                      """Betweenness centrality: K4"""
10: (8)                     G = nx.complete_graph(4)
11: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True)
12: (8)                     b_answer = {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
13: (8)                     for n in sorted(G):
14: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
15: (8)                     G.add_edge(0, 1, weight=0.5, other=0.3)
```

```
16: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True,
weight=None)
17: (8)                     for n in sorted(G):
18: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
19: (8)                     wb_answer = {0: 0.2222222, 1: 0.2222222, 2: 0.30555555, 3: 0.30555555}
20: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True,
weight="weight")
21: (8)                     for n in sorted(G):
22: (12)                        assert b[n] == pytest.approx(wb_answer[n], abs=1e-7)
23: (8)                     wb_answer = {0: 0.2051282, 1: 0.2051282, 2: 0.33974358, 3: 0.33974358}
24: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True,
weight="other")
25: (8)                     for n in sorted(G):
26: (12)                        assert b[n] == pytest.approx(wb_answer[n], abs=1e-7)
27: (4)                 def test_K4(self):
28: (8)                     """Betweenness centrality: K4"""
29: (8)                     G = nx.complete_graph(4)
30: (8)                     for solver in ["full", "lu", "cg"]:
31: (12)                        b = nx.current_flow_betweenness_centrality(
32: (16)                            G, normalized=False, solver=solver
33: (12)                        )
34: (12)                        b_answer = {0: 0.75, 1: 0.75, 2: 0.75, 3: 0.75}
35: (12)                        for n in sorted(G):
36: (16)                            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
37: (4)                 def test_P4_normalized(self):
38: (8)                     """Betweenness centrality: P4 normalized"""
39: (8)                     G = nx.path_graph(4)
40: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True)
41: (8)                     b_answer = {0: 0, 1: 2.0 / 3, 2: 2.0 / 3, 3: 0}
42: (8)                     for n in sorted(G):
43: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
44: (4)                 def test_P4(self):
45: (8)                     """Betweenness centrality: P4"""
46: (8)                     G = nx.path_graph(4)
47: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=False)
48: (8)                     b_answer = {0: 0, 1: 2, 2: 2, 3: 0}
49: (8)                     for n in sorted(G):
50: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
51: (4)                 def test_star(self):
52: (8)                     """Betweenness centrality: star"""
53: (8)                     G = nx.Graph()
54: (8)                     nx.add_star(G, ["a", "b", "c", "d"])
55: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True)
56: (8)                     b_answer = {"a": 1.0, "b": 0.0, "c": 0.0, "d": 0.0}
57: (8)                     for n in sorted(G):
58: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
59: (4)                 def test_solvers2(self):
60: (8)                     """Betweenness centrality: alternate solvers"""
61: (8)                     G = nx.complete_graph(4)
62: (8)                     for solver in ["full", "lu", "cg"]:
63: (12)                        b = nx.current_flow_betweenness_centrality(
64: (16)                            G, normalized=False, solver=solver
65: (12)                        )
66: (12)                        b_answer = {0: 0.75, 1: 0.75, 2: 0.75, 3: 0.75}
67: (12)                        for n in sorted(G):
68: (16)                            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
69: (0)             class TestApproximateFlowBetweennessCentrality:
70: (4)                 def test_K4_normalized(self):
71: (8)                     "Approximate current-flow betweenness centrality: K4 normalized"
72: (8)                     G = nx.complete_graph(4)
73: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True)
74: (8)                     epsilon = 0.1
75: (8)                     ba = approximate_cfbc(G, normalized=True, epsilon=0.5 * epsilon)
76: (8)                     for n in sorted(G):
77: (12)                        np.testing.assert_allclose(b[n], ba[n], atol=epsilon)
78: (4)                 def test_K4(self):
79: (8)                     "Approximate current-flow betweenness centrality: K4"
80: (8)                     G = nx.complete_graph(4)
81: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=False)
```

```
 82: (8)                     epsilon = 0.1
 83: (8)                     ba = approximate_cfbc(G, normalized=False, epsilon=0.5 * epsilon)
 84: (8)                     for n in sorted(G):
 85: (12)                        np.testing.assert_allclose(b[n], ba[n], atol=epsilon * len(G) **
 2)
 86: (4)                 def test_star(self):
 87: (8)                     "Approximate current-flow betweenness centrality: star"
 88: (8)                     G = nx.Graph()
 89: (8)                     nx.add_star(G, ["a", "b", "c", "d"])
 90: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True)
 91: (8)                     epsilon = 0.1
 92: (8)                     ba = approximate_cfbc(G, normalized=True, epsilon=0.5 * epsilon)
 93: (8)                     for n in sorted(G):
 94: (12)                        np.testing.assert_allclose(b[n], ba[n], atol=epsilon)
 95: (4)                 def test_grid(self):
 96: (8)                     "Approximate current-flow betweenness centrality: 2d grid"
 97: (8)                     G = nx.grid_2d_graph(4, 4)
 98: (8)                     b = nx.current_flow_betweenness_centrality(G, normalized=True)
 99: (8)                     epsilon = 0.1
 100: (8)                    ba = approximate_cfbc(G, normalized=True, epsilon=0.5 * epsilon)
 101: (8)                    for n in sorted(G):
 102: (12)                       np.testing.assert_allclose(b[n], ba[n], atol=epsilon)
 103: (4)                def test_seed(self):
 104: (8)                    G = nx.complete_graph(4)
 105: (8)                    b = approximate_cfbc(G, normalized=False, epsilon=0.05, seed=1)
 106: (8)                    b_answer = {0: 0.75, 1: 0.75, 2: 0.75, 3: 0.75}
 107: (8)                    for n in sorted(G):
 108: (12)                       np.testing.assert_allclose(b[n], b_answer[n], atol=0.1)
 109: (4)                def test_solvers(self):
 110: (8)                    "Approximate current-flow betweenness centrality: solvers"
 111: (8)                    G = nx.complete_graph(4)
 112: (8)                    epsilon = 0.1
 113: (8)                    for solver in ["full", "lu", "cg"]:
 114: (12)                       b = approximate_cfbc(
 115: (16)                           G, normalized=False, solver=solver, epsilon=0.5 * epsilon
 116: (12)                       )
 117: (12)                       b_answer = {0: 0.75, 1: 0.75, 2: 0.75, 3: 0.75}
 118: (12)                       for n in sorted(G):
 119: (16)                           np.testing.assert_allclose(b[n], b_answer[n], atol=epsilon)
 120: (4)                def test_lower_kmax(self):
 121: (8)                    G = nx.complete_graph(4)
 122: (8)                    with pytest.raises(nx.NetworkXError, match="Increase kmax or
 epsilon"):
 123: (12)                       nx.approximate_current_flow_betweenness_centrality(G, kmax=4)
 124: (0)            class TestWeightedFlowBetweennessCentrality:
 125: (4)                pass
 126: (0)            class TestEdgeFlowBetweennessCentrality:
 127: (4)                def test_K4(self):
 128: (8)                    """Edge flow betweenness centrality: K4"""
 129: (8)                    G = nx.complete_graph(4)
 130: (8)                    b = edge_current_flow(G, normalized=True)
 131: (8)                    b_answer = dict.fromkeys(G.edges(), 0.25)
 132: (8)                    for (s, t), v1 in b_answer.items():
 133: (12)                       v2 = b.get((s, t), b.get((t, s)))
 134: (12)                       assert v1 == pytest.approx(v2, abs=1e-7)
 135: (4)                def test_K4_normalized(self):
 136: (8)                    """Edge flow betweenness centrality: K4"""
 137: (8)                    G = nx.complete_graph(4)
 138: (8)                    b = edge_current_flow(G, normalized=False)
 139: (8)                    b_answer = dict.fromkeys(G.edges(), 0.75)
 140: (8)                    for (s, t), v1 in b_answer.items():
 141: (12)                       v2 = b.get((s, t), b.get((t, s)))
 142: (12)                       assert v1 == pytest.approx(v2, abs=1e-7)
 143: (4)                def test_C4(self):
 144: (8)                    """Edge flow betweenness centrality: C4"""
 145: (8)                    G = nx.cycle_graph(4)
 146: (8)                    b = edge_current_flow(G, normalized=False)
 147: (8)                    b_answer = {(0, 1): 1.25, (0, 3): 1.25, (1, 2): 1.25, (2, 3): 1.25}
 148: (8)                    for (s, t), v1 in b_answer.items():
```

```
149: (12)                        v2 = b.get((s, t), b.get((t, s)))
150: (12)                        assert v1 == pytest.approx(v2, abs=1e-7)
151: (4)                    def test_P4(self):
152: (8)                        """Edge betweenness centrality: P4"""
153: (8)                        G = nx.path_graph(4)
154: (8)                        b = edge_current_flow(G, normalized=False)
155: (8)                        b_answer = {(0, 1): 1.5, (1, 2): 2.0, (2, 3): 1.5}
156: (8)                        for (s, t), v1 in b_answer.items():
157: (12)                           v2 = b.get((s, t), b.get((t, s)))
158: (12)                           assert v1 == pytest.approx(v2, abs=1e-7)
159: (0)                @pytest.mark.parametrize(
160: (4)                    "centrality_func",
161: (4)                    (
162: (8)                        nx.current_flow_betweenness_centrality,
163: (8)                        nx.edge_current_flow_betweenness_centrality,
164: (8)                        nx.approximate_current_flow_betweenness_centrality,
165: (4)                    ),
166: (0)                )
167: (0)                def test_unconnected_graphs_betweenness_centrality(centrality_func):
168: (4)                    G = nx.Graph([(1, 2), (3, 4)])
169: (4)                    G.add_node(5)
170: (4)                    with pytest.raises(nx.NetworkXError, match="Graph not connected"):
171: (8)                        centrality_func(G)
```

File 158 - test_current_flow_closeness.py:

```
1: (0)                  import pytest
2: (0)                  pytest.importorskip("numpy")
3: (0)                  pytest.importorskip("scipy")
4: (0)                  import networkx as nx
5: (0)                  class TestFlowClosenessCentrality:
6: (4)                      def test_K4(self):
7: (8)                          """Closeness centrality: K4"""
8: (8)                          G = nx.complete_graph(4)
9: (8)                          b = nx.current_flow_closeness_centrality(G)
10: (8)                         b_answer = {0: 2.0 / 3, 1: 2.0 / 3, 2: 2.0 / 3, 3: 2.0 / 3}
11: (8)                         for n in sorted(G):
12: (12)                            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
13: (4)                     def test_P4(self):
14: (8)                         """Closeness centrality: P4"""
15: (8)                         G = nx.path_graph(4)
16: (8)                         b = nx.current_flow_closeness_centrality(G)
17: (8)                         b_answer = {0: 1.0 / 6, 1: 1.0 / 4, 2: 1.0 / 4, 3: 1.0 / 6}
18: (8)                         for n in sorted(G):
19: (12)                            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
20: (4)                     def test_star(self):
21: (8)                         """Closeness centrality: star"""
22: (8)                         G = nx.Graph()
23: (8)                         nx.add_star(G, ["a", "b", "c", "d"])
24: (8)                         b = nx.current_flow_closeness_centrality(G)
25: (8)                         b_answer = {"a": 1.0 / 3, "b": 0.6 / 3, "c": 0.6 / 3, "d": 0.6 / 3}
26: (8)                         for n in sorted(G):
27: (12)                            assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
28: (4)                     def test_current_flow_closeness_centrality_not_connected(self):
29: (8)                         G = nx.Graph()
30: (8)                         G.add_nodes_from([1, 2, 3])
31: (8)                         with pytest.raises(nx.NetworkXError):
32: (12)                            nx.current_flow_closeness_centrality(G)
33: (0)                 class TestWeightedFlowClosenessCentrality:
34: (4)                     pass
```

File 159 - test_current_flow_betweenness_centrality_subset.py:

```
1: (0)                  import pytest
2: (0)                  pytest.importorskip("numpy")
```

```
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx import edge_current_flow_betweenness_centrality as
edge_current_flow
6: (0)              from networkx import (
7: (4)                  edge_current_flow_betweenness_centrality_subset as
edge_current_flow_subset,
8: (0)              )
9: (0)              class TestFlowBetweennessCentrality:
10: (4)                 def test_K4_normalized(self):
11: (8)                     """Betweenness centrality: K4"""
12: (8)                     G = nx.complete_graph(4)
13: (8)                     b = nx.current_flow_betweenness_centrality_subset(
14: (12)                        G, list(G), list(G), normalized=True
15: (8)                     )
16: (8)                     b_answer = nx.current_flow_betweenness_centrality(G, normalized=True)
17: (8)                     for n in sorted(G):
18: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
19: (4)                 def test_K4(self):
20: (8)                     """Betweenness centrality: K4"""
21: (8)                     G = nx.complete_graph(4)
22: (8)                     b = nx.current_flow_betweenness_centrality_subset(
23: (12)                        G, list(G), list(G), normalized=True
24: (8)                     )
25: (8)                     b_answer = nx.current_flow_betweenness_centrality(G, normalized=True)
26: (8)                     for n in sorted(G):
27: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
28: (8)                     G.add_edge(0, 1, weight=0.5, other=0.3)
29: (8)                     b = nx.current_flow_betweenness_centrality_subset(
30: (12)                        G, list(G), list(G), normalized=True, weight=None
31: (8)                     )
32: (8)                     for n in sorted(G):
33: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
34: (8)                     b = nx.current_flow_betweenness_centrality_subset(
35: (12)                        G, list(G), list(G), normalized=True
36: (8)                     )
37: (8)                     b_answer = nx.current_flow_betweenness_centrality(G, normalized=True)
38: (8)                     for n in sorted(G):
39: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
40: (8)                     b = nx.current_flow_betweenness_centrality_subset(
41: (12)                        G, list(G), list(G), normalized=True, weight="other"
42: (8)                     )
43: (8)                     b_answer = nx.current_flow_betweenness_centrality(
44: (12)                        G, normalized=True, weight="other"
45: (8)                     )
46: (8)                     for n in sorted(G):
47: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
48: (4)                 def test_P4_normalized(self):
49: (8)                     """Betweenness centrality: P4 normalized"""
50: (8)                     G = nx.path_graph(4)
51: (8)                     b = nx.current_flow_betweenness_centrality_subset(
52: (12)                        G, list(G), list(G), normalized=True
53: (8)                     )
54: (8)                     b_answer = nx.current_flow_betweenness_centrality(G, normalized=True)
55: (8)                     for n in sorted(G):
56: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
57: (4)                 def test_P4(self):
58: (8)                     """Betweenness centrality: P4"""
59: (8)                     G = nx.path_graph(4)
60: (8)                     b = nx.current_flow_betweenness_centrality_subset(
61: (12)                        G, list(G), list(G), normalized=True
62: (8)                     )
63: (8)                     b_answer = nx.current_flow_betweenness_centrality(G, normalized=True)
64: (8)                     for n in sorted(G):
65: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
66: (4)                 def test_star(self):
67: (8)                     """Betweenness centrality: star"""
68: (8)                     G = nx.Graph()
69: (8)                     nx.add_star(G, ["a", "b", "c", "d"])
```

```
70: (8)                     b = nx.current_flow_betweenness_centrality_subset(
71: (12)                        G, list(G), list(G), normalized=True
72: (8)                     )
73: (8)                     b_answer = nx.current_flow_betweenness_centrality(G, normalized=True)
74: (8)                     for n in sorted(G):
75: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
76: (0)             class TestEdgeFlowBetweennessCentrality:
77: (4)                 def test_K4_normalized(self):
78: (8)                     """Betweenness centrality: K4"""
79: (8)                     G = nx.complete_graph(4)
80: (8)                     b = edge_current_flow_subset(G, list(G), list(G), normalized=True)
81: (8)                     b_answer = edge_current_flow(G, normalized=True)
82: (8)                     for (s, t), v1 in b_answer.items():
83: (12)                        v2 = b.get((s, t), b.get((t, s)))
84: (12)                        assert v1 == pytest.approx(v2, abs=1e-7)
85: (4)                 def test_K4(self):
86: (8)                     """Betweenness centrality: K4"""
87: (8)                     G = nx.complete_graph(4)
88: (8)                     b = edge_current_flow_subset(G, list(G), list(G), normalized=False)
89: (8)                     b_answer = edge_current_flow(G, normalized=False)
90: (8)                     for (s, t), v1 in b_answer.items():
91: (12)                        v2 = b.get((s, t), b.get((t, s)))
92: (12)                        assert v1 == pytest.approx(v2, abs=1e-7)
93: (8)                     G.add_edge(0, 1, weight=0.5, other=0.3)
94: (8)                     b = edge_current_flow_subset(G, list(G), list(G), normalized=False,
weight=None)
95: (8)                     for (s, t), v1 in b_answer.items():
96: (12)                        v2 = b.get((s, t), b.get((t, s)))
97: (12)                        assert v1 == pytest.approx(v2, abs=1e-7)
98: (8)                     b = edge_current_flow_subset(G, list(G), list(G), normalized=False)
99: (8)                     b_answer = edge_current_flow(G, normalized=False)
100: (8)                    for (s, t), v1 in b_answer.items():
101: (12)                       v2 = b.get((s, t), b.get((t, s)))
102: (12)                       assert v1 == pytest.approx(v2, abs=1e-7)
103: (8)                    b = edge_current_flow_subset(
104: (12)                       G, list(G), list(G), normalized=False, weight="other"
105: (8)                    )
106: (8)                    b_answer = edge_current_flow(G, normalized=False, weight="other")
107: (8)                    for (s, t), v1 in b_answer.items():
108: (12)                       v2 = b.get((s, t), b.get((t, s)))
109: (12)                       assert v1 == pytest.approx(v2, abs=1e-7)
110: (4)                def test_C4(self):
111: (8)                    """Edge betweenness centrality: C4"""
112: (8)                    G = nx.cycle_graph(4)
113: (8)                    b = edge_current_flow_subset(G, list(G), list(G), normalized=True)
114: (8)                    b_answer = edge_current_flow(G, normalized=True)
115: (8)                    for (s, t), v1 in b_answer.items():
116: (12)                       v2 = b.get((s, t), b.get((t, s)))
117: (12)                       assert v1 == pytest.approx(v2, abs=1e-7)
118: (4)                def test_P4(self):
119: (8)                    """Edge betweenness centrality: P4"""
120: (8)                    G = nx.path_graph(4)
121: (8)                    b = edge_current_flow_subset(G, list(G), list(G), normalized=True)
122: (8)                    b_answer = edge_current_flow(G, normalized=True)
123: (8)                    for (s, t), v1 in b_answer.items():
124: (12)                       v2 = b.get((s, t), b.get((t, s)))
125: (12)                       assert v1 == pytest.approx(v2, abs=1e-7)

----------------------------------------

File 160 - test_dispersion.py:

1: (0)              import networkx as nx
2: (0)              def small_ego_G():
3: (4)                  """The sample network from https://arxiv.org/pdf/1310.6753v1.pdf"""
4: (4)                  edges = [
5: (8)                      ("a", "b"),
6: (8)                      ("a", "c"),
7: (8)                      ("b", "c"),
```

```
8: (8)                      ("b", "d"),
9: (8)                      ("b", "e"),
10: (8)                     ("b", "f"),
11: (8)                     ("c", "d"),
12: (8)                     ("c", "f"),
13: (8)                     ("c", "h"),
14: (8)                     ("d", "f"),
15: (8)                     ("e", "f"),
16: (8)                     ("f", "h"),
17: (8)                     ("h", "j"),
18: (8)                     ("h", "k"),
19: (8)                     ("i", "j"),
20: (8)                     ("i", "k"),
21: (8)                     ("j", "k"),
22: (8)                     ("u", "a"),
23: (8)                     ("u", "b"),
24: (8)                     ("u", "c"),
25: (8)                     ("u", "d"),
26: (8)                     ("u", "e"),
27: (8)                     ("u", "f"),
28: (8)                     ("u", "g"),
29: (8)                     ("u", "h"),
30: (8)                     ("u", "i"),
31: (8)                     ("u", "j"),
32: (8)                     ("u", "k"),
33: (4)                 ]
34: (4)                 G = nx.Graph()
35: (4)                 G.add_edges_from(edges)
36: (4)                 return G
37: (0)             class TestDispersion:
38: (4)                 def test_article(self):
39: (8)                     """our algorithm matches article's"""
40: (8)                     G = small_ego_G()
41: (8)                     disp_uh = nx.dispersion(G, "u", "h", normalized=False)
42: (8)                     disp_ub = nx.dispersion(G, "u", "b", normalized=False)
43: (8)                     assert disp_uh == 4
44: (8)                     assert disp_ub == 1
45: (4)                 def test_results_length(self):
46: (8)                     """there is a result for every node"""
47: (8)                     G = small_ego_G()
48: (8)                     disp = nx.dispersion(G)
49: (8)                     disp_Gu = nx.dispersion(G, "u")
50: (8)                     disp_uv = nx.dispersion(G, "u", "h")
51: (8)                     assert len(disp) == len(G)
52: (8)                     assert len(disp_Gu) == len(G) - 1
53: (8)                     assert isinstance(disp_uv, float)
54: (4)                 def test_dispersion_v_only(self):
55: (8)                     G = small_ego_G()
56: (8)                     disp_G_h = nx.dispersion(G, v="h", normalized=False)
57: (8)                     disp_G_h_normalized = nx.dispersion(G, v="h", normalized=True)
58: (8)                     assert disp_G_h == {"c": 0, "f": 0, "j": 0, "k": 0, "u": 4}
59: (8)                     assert disp_G_h_normalized == {"c": 0.0, "f": 0.0, "j": 0.0, "k": 0.0,
"u": 1.0}
60: (4)                 def test_impossible_things(self):
61: (8)                     G = nx.karate_club_graph()
62: (8)                     disp = nx.dispersion(G)
63: (8)                     for u in disp:
64: (12)                        for v in disp[u]:
65: (16)                            assert disp[u][v] >= 0
```

File 161 - test_degree_centrality.py:

```
1: (0)              """
2: (4)                  Unit tests for degree centrality.
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
```

```
(0)  class TestDegreeCentrality:
(4)      def setup_method(self):
(8)          self.K = nx.krackhardt_kite_graph()
(8)          self.P3 = nx.path_graph(3)
(8)          self.K5 = nx.complete_graph(5)
(8)          F = nx.Graph()  # Florentine families
(8)          F.add_edge("Acciaiuoli", "Medici")
(8)          F.add_edge("Castellani", "Peruzzi")
(8)          F.add_edge("Castellani", "Strozzi")
(8)          F.add_edge("Castellani", "Barbadori")
(8)          F.add_edge("Medici", "Barbadori")
(8)          F.add_edge("Medici", "Ridolfi")
(8)          F.add_edge("Medici", "Tornabuoni")
(8)          F.add_edge("Medici", "Albizzi")
(8)          F.add_edge("Medici", "Salviati")
(8)          F.add_edge("Salviati", "Pazzi")
(8)          F.add_edge("Peruzzi", "Strozzi")
(8)          F.add_edge("Peruzzi", "Bischeri")
(8)          F.add_edge("Strozzi", "Ridolfi")
(8)          F.add_edge("Strozzi", "Bischeri")
(8)          F.add_edge("Ridolfi", "Tornabuoni")
(8)          F.add_edge("Tornabuoni", "Guadagni")
(8)          F.add_edge("Albizzi", "Ginori")
(8)          F.add_edge("Albizzi", "Guadagni")
(8)          F.add_edge("Bischeri", "Guadagni")
(8)          F.add_edge("Guadagni", "Lamberteschi")
(8)          self.F = F
(8)          G = nx.DiGraph()
(8)          G.add_edge(0, 5)
(8)          G.add_edge(1, 5)
(8)          G.add_edge(2, 5)
(8)          G.add_edge(3, 5)
(8)          G.add_edge(4, 5)
(8)          G.add_edge(5, 6)
(8)          G.add_edge(5, 7)
(8)          G.add_edge(5, 8)
(8)          self.G = G
(4)      def test_degree_centrality_1(self):
(8)          d = nx.degree_centrality(self.K5)
(8)          exact = dict(zip(range(5), [1] * 5))
(8)          for n, dc in d.items():
(12)             assert exact[n] == pytest.approx(dc, abs=1e-7)
(4)      def test_degree_centrality_2(self):
(8)          d = nx.degree_centrality(self.P3)
(8)          exact = {0: 0.5, 1: 1, 2: 0.5}
(8)          for n, dc in d.items():
(12)             assert exact[n] == pytest.approx(dc, abs=1e-7)
(4)      def test_degree_centrality_3(self):
(8)          d = nx.degree_centrality(self.K)
(8)          exact = {
(12)             0: 0.444,
(12)             1: 0.444,
(12)             2: 0.333,
(12)             3: 0.667,
(12)             4: 0.333,
(12)             5: 0.556,
(12)             6: 0.556,
(12)             7: 0.333,
(12)             8: 0.222,
(12)             9: 0.111,
(8)          }
(8)          for n, dc in d.items():
(12)             assert exact[n] == pytest.approx(float(f"{dc:.3f}"), abs=1e-7)
(4)      def test_degree_centrality_4(self):
(8)          d = nx.degree_centrality(self.F)
(8)          names = sorted(self.F.nodes())
(8)          dcs = [
(12)             0.071,
(12)             0.214,
```

```
75: (12)                        0.143,
76: (12)                        0.214,
77: (12)                        0.214,
78: (12)                        0.071,
79: (12)                        0.286,
80: (12)                        0.071,
81: (12)                        0.429,
82: (12)                        0.071,
83: (12)                        0.214,
84: (12)                        0.214,
85: (12)                        0.143,
86: (12)                        0.286,
87: (12)                        0.214,
88: (8)                     ]
89: (8)                     exact = dict(zip(names, dcs))
90: (8)                     for n, dc in d.items():
91: (12)                        assert exact[n] == pytest.approx(float(f"{dc:.3f}"), abs=1e-7)
92: (4)                 def test_indegree_centrality(self):
93: (8)                     d = nx.in_degree_centrality(self.G)
94: (8)                     exact = {
95: (12)                        0: 0.0,
96: (12)                        1: 0.0,
97: (12)                        2: 0.0,
98: (12)                        3: 0.0,
99: (12)                        4: 0.0,
100: (12)                       5: 0.625,
101: (12)                       6: 0.125,
102: (12)                       7: 0.125,
103: (12)                       8: 0.125,
104: (8)                    }
105: (8)                    for n, dc in d.items():
106: (12)                       assert exact[n] == pytest.approx(dc, abs=1e-7)
107: (4)                def test_outdegree_centrality(self):
108: (8)                    d = nx.out_degree_centrality(self.G)
109: (8)                    exact = {
110: (12)                       0: 0.125,
111: (12)                       1: 0.125,
112: (12)                       2: 0.125,
113: (12)                       3: 0.125,
114: (12)                       4: 0.125,
115: (12)                       5: 0.375,
116: (12)                       6: 0.0,
117: (12)                       7: 0.0,
118: (12)                       8: 0.0,
119: (8)                    }
120: (8)                    for n, dc in d.items():
121: (12)                       assert exact[n] == pytest.approx(dc, abs=1e-7)
122: (4)                def test_small_graph_centrality(self):
123: (8)                    G = nx.empty_graph(create_using=nx.DiGraph)
124: (8)                    assert {} == nx.degree_centrality(G)
125: (8)                    assert {} == nx.out_degree_centrality(G)
126: (8)                    assert {} == nx.in_degree_centrality(G)
127: (8)                    G = nx.empty_graph(1, create_using=nx.DiGraph)
128: (8)                    assert {0: 1} == nx.degree_centrality(G)
129: (8)                    assert {0: 1} == nx.out_degree_centrality(G)
130: (8)                    assert {0: 1} == nx.in_degree_centrality(G)

----------------------------------------

File 162 - test_group.py:

1: (0)              """
2: (0)              Tests for Group Centrality Measures
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              class TestGroupBetweennessCentrality:
7: (4)                  def test_group_betweenness_single_node(self):
8: (8)                      """
```

```
 9: (8)                         Group betweenness centrality for single node group
 10: (8)                        """
 11: (8)                        G = nx.path_graph(5)
 12: (8)                        C = [1]
 13: (8)                        b = nx.group_betweenness_centrality(
 14: (12)                           G, C, weight=None, normalized=False, endpoints=False
 15: (8)                        )
 16: (8)                        b_answer = 3.0
 17: (8)                        assert b == b_answer
 18: (4)                    def test_group_betweenness_with_endpoints(self):
 19: (8)                        """
 20: (8)                        Group betweenness centrality for single node group
 21: (8)                        """
 22: (8)                        G = nx.path_graph(5)
 23: (8)                        C = [1]
 24: (8)                        b = nx.group_betweenness_centrality(
 25: (12)                           G, C, weight=None, normalized=False, endpoints=True
 26: (8)                        )
 27: (8)                        b_answer = 7.0
 28: (8)                        assert b == b_answer
 29: (4)                    def test_group_betweenness_normalized(self):
 30: (8)                        """
 31: (8)                        Group betweenness centrality for group with more than
 32: (8)                        1 node and normalized
 33: (8)                        """
 34: (8)                        G = nx.path_graph(5)
 35: (8)                        C = [1, 3]
 36: (8)                        b = nx.group_betweenness_centrality(
 37: (12)                           G, C, weight=None, normalized=True, endpoints=False
 38: (8)                        )
 39: (8)                        b_answer = 1.0
 40: (8)                        assert b == b_answer
 41: (4)                    def test_two_group_betweenness_value_zero(self):
 42: (8)                        """
 43: (8)                        Group betweenness centrality value of 0
 44: (8)                        """
 45: (8)                        G = nx.cycle_graph(7)
 46: (8)                        C = [[0, 1, 6], [0, 1, 5]]
 47: (8)                        b = nx.group_betweenness_centrality(G, C, weight=None,
normalized=False)
 48: (8)                        b_answer = [0.0, 3.0]
 49: (8)                        assert b == b_answer
 50: (4)                    def test_group_betweenness_value_zero(self):
 51: (8)                        """
 52: (8)                        Group betweenness centrality value of 0
 53: (8)                        """
 54: (8)                        G = nx.cycle_graph(6)
 55: (8)                        C = [0, 1, 5]
 56: (8)                        b = nx.group_betweenness_centrality(G, C, weight=None,
normalized=False)
 57: (8)                        b_answer = 0.0
 58: (8)                        assert b == b_answer
 59: (4)                    def test_group_betweenness_disconnected_graph(self):
 60: (8)                        """
 61: (8)                        Group betweenness centrality in a disconnected graph
 62: (8)                        """
 63: (8)                        G = nx.path_graph(5)
 64: (8)                        G.remove_edge(0, 1)
 65: (8)                        C = [1]
 66: (8)                        b = nx.group_betweenness_centrality(G, C, weight=None,
normalized=False)
 67: (8)                        b_answer = 0.0
 68: (8)                        assert b == b_answer
 69: (4)                    def test_group_betweenness_node_not_in_graph(self):
 70: (8)                        """
 71: (8)                        Node(s) in C not in graph, raises NodeNotFound exception
 72: (8)                        """
 73: (8)                        with pytest.raises(nx.NodeNotFound):
 74: (12)                           nx.group_betweenness_centrality(nx.path_graph(5), [4, 7, 8])
```

```
75: (4)                 def test_group_betweenness_directed_weighted(self):
76: (8)                     """
77: (8)                     Group betweenness centrality in a directed and weighted graph
78: (8)                     """
79: (8)                     G = nx.DiGraph()
80: (8)                     G.add_edge(1, 0, weight=1)
81: (8)                     G.add_edge(0, 2, weight=2)
82: (8)                     G.add_edge(1, 2, weight=3)
83: (8)                     G.add_edge(3, 1, weight=4)
84: (8)                     G.add_edge(2, 3, weight=1)
85: (8)                     G.add_edge(4, 3, weight=6)
86: (8)                     G.add_edge(2, 4, weight=7)
87: (8)                     C = [1, 2]
88: (8)                     b = nx.group_betweenness_centrality(G, C, weight="weight",
normalized=False)
89: (8)                     b_answer = 5.0
90: (8)                     assert b == b_answer
91: (0)             class TestProminentGroup:
92: (4)                 np = pytest.importorskip("numpy")
93: (4)                 pd = pytest.importorskip("pandas")
94: (4)                 def test_prominent_group_single_node(self):
95: (8)                     """
96: (8)                     Prominent group for single node
97: (8)                     """
98: (8)                     G = nx.path_graph(5)
99: (8)                     k = 1
100: (8)                    b, g = nx.prominent_group(G, k, normalized=False, endpoints=False)
101: (8)                    b_answer, g_answer = 4.0, [2]
102: (8)                    assert b == b_answer and g == g_answer
103: (4)                def test_prominent_group_with_c(self):
104: (8)                    """
105: (8)                    Prominent group without some nodes
106: (8)                    """
107: (8)                    G = nx.path_graph(5)
108: (8)                    k = 1
109: (8)                    b, g = nx.prominent_group(G, k, normalized=False, C=[2])
110: (8)                    b_answer, g_answer = 3.0, [1]
111: (8)                    assert b == b_answer and g == g_answer
112: (4)                def test_prominent_group_normalized_endpoints(self):
113: (8)                    """
114: (8)                    Prominent group with normalized result, with endpoints
115: (8)                    """
116: (8)                    G = nx.cycle_graph(7)
117: (8)                    k = 2
118: (8)                    b, g = nx.prominent_group(G, k, normalized=True, endpoints=True)
119: (8)                    b_answer, g_answer = 1.7, [2, 5]
120: (8)                    assert b == b_answer and g == g_answer
121: (4)                def test_prominent_group_disconnected_graph(self):
122: (8)                    """
123: (8)                    Prominent group of disconnected graph
124: (8)                    """
125: (8)                    G = nx.path_graph(6)
126: (8)                    G.remove_edge(0, 1)
127: (8)                    k = 1
128: (8)                    b, g = nx.prominent_group(G, k, weight=None, normalized=False)
129: (8)                    b_answer, g_answer = 4.0, [3]
130: (8)                    assert b == b_answer and g == g_answer
131: (4)                def test_prominent_group_node_not_in_graph(self):
132: (8)                    """
133: (8)                    Node(s) in C not in graph, raises NodeNotFound exception
134: (8)                    """
135: (8)                    with pytest.raises(nx.NodeNotFound):
136: (12)                       nx.prominent_group(nx.path_graph(5), 1, C=[10])
137: (4)                def test_group_betweenness_directed_weighted(self):
138: (8)                    """
139: (8)                    Group betweenness centrality in a directed and weighted graph
140: (8)                    """
141: (8)                    G = nx.DiGraph()
142: (8)                    G.add_edge(1, 0, weight=1)
```

```
143: (8)                    G.add_edge(0, 2, weight=2)
144: (8)                    G.add_edge(1, 2, weight=3)
145: (8)                    G.add_edge(3, 1, weight=4)
146: (8)                    G.add_edge(2, 3, weight=1)
147: (8)                    G.add_edge(4, 3, weight=6)
148: (8)                    G.add_edge(2, 4, weight=7)
149: (8)                    k = 2
150: (8)                    b, g = nx.prominent_group(G, k, weight="weight", normalized=False)
151: (8)                    b_answer, g_answer = 5.0, [1, 2]
152: (8)                    assert b == b_answer and g == g_answer
153: (4)                def test_prominent_group_greedy_algorithm(self):
154: (8)                    """
155: (8)                    Group betweenness centrality in a greedy algorithm
156: (8)                    """
157: (8)                    G = nx.cycle_graph(7)
158: (8)                    k = 2
159: (8)                    b, g = nx.prominent_group(G, k, normalized=True, endpoints=True,
greedy=True)
160: (8)                    b_answer, g_answer = 1.7, [6, 3]
161: (8)                    assert b == b_answer and g == g_answer
162: (0)            class TestGroupClosenessCentrality:
163: (4)                def test_group_closeness_single_node(self):
164: (8)                    """
165: (8)                    Group closeness centrality for a single node group
166: (8)                    """
167: (8)                    G = nx.path_graph(5)
168: (8)                    c = nx.group_closeness_centrality(G, [1])
169: (8)                    c_answer = nx.closeness_centrality(G, 1)
170: (8)                    assert c == c_answer
171: (4)                def test_group_closeness_disconnected(self):
172: (8)                    """
173: (8)                    Group closeness centrality for a disconnected graph
174: (8)                    """
175: (8)                    G = nx.Graph()
176: (8)                    G.add_nodes_from([1, 2, 3, 4])
177: (8)                    c = nx.group_closeness_centrality(G, [1, 2])
178: (8)                    c_answer = 0
179: (8)                    assert c == c_answer
180: (4)                def test_group_closeness_multiple_node(self):
181: (8)                    """
182: (8)                    Group closeness centrality for a group with more than
183: (8)                    1 node
184: (8)                    """
185: (8)                    G = nx.path_graph(4)
186: (8)                    c = nx.group_closeness_centrality(G, [1, 2])
187: (8)                    c_answer = 1
188: (8)                    assert c == c_answer
189: (4)                def test_group_closeness_node_not_in_graph(self):
190: (8)                    """
191: (8)                    Node(s) in S not in graph, raises NodeNotFound exception
192: (8)                    """
193: (8)                    with pytest.raises(nx.NodeNotFound):
194: (12)                       nx.group_closeness_centrality(nx.path_graph(5), [6, 7, 8])
195: (0)            class TestGroupDegreeCentrality:
196: (4)                def test_group_degree_centrality_single_node(self):
197: (8)                    """
198: (8)                    Group degree centrality for a single node group
199: (8)                    """
200: (8)                    G = nx.path_graph(4)
201: (8)                    d = nx.group_degree_centrality(G, [1])
202: (8)                    d_answer = nx.degree_centrality(G)[1]
203: (8)                    assert d == d_answer
204: (4)                def test_group_degree_centrality_multiple_node(self):
205: (8)                    """
206: (8)                    Group degree centrality for group with more than
207: (8)                    1 node
208: (8)                    """
209: (8)                    G = nx.Graph()
210: (8)                    G.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8])
```

```
211: (8)                    G.add_edges_from(
212: (12)                       [(1, 2), (1, 3), (1, 6), (1, 7), (1, 8), (2, 3), (2, 4), (2, 5)]
213: (8)                    )
214: (8)                    d = nx.group_degree_centrality(G, [1, 2])
215: (8)                    d_answer = 1
216: (8)                    assert d == d_answer
217: (4)                def test_group_in_degree_centrality(self):
218: (8)                    """
219: (8)                    Group in-degree centrality in a DiGraph
220: (8)                    """
221: (8)                    G = nx.DiGraph()
222: (8)                    G.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8])
223: (8)                    G.add_edges_from(
224: (12)                       [(1, 2), (1, 3), (1, 6), (1, 7), (1, 8), (2, 3), (2, 4), (2, 5)]
225: (8)                    )
226: (8)                    d = nx.group_in_degree_centrality(G, [1, 2])
227: (8)                    d_answer = 0
228: (8)                    assert d == d_answer
229: (4)                def test_group_out_degree_centrality(self):
230: (8)                    """
231: (8)                    Group out-degree centrality in a DiGraph
232: (8)                    """
233: (8)                    G = nx.DiGraph()
234: (8)                    G.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8])
235: (8)                    G.add_edges_from(
236: (12)                       [(1, 2), (1, 3), (1, 6), (1, 7), (1, 8), (2, 3), (2, 4), (2, 5)]
237: (8)                    )
238: (8)                    d = nx.group_out_degree_centrality(G, [1, 2])
239: (8)                    d_answer = 1
240: (8)                    assert d == d_answer
241: (4)                def test_group_degree_centrality_node_not_in_graph(self):
242: (8)                    """
243: (8)                    Node(s) in S not in graph, raises NetworkXError
244: (8)                    """
245: (8)                    with pytest.raises(nx.NetworkXError):
246: (12)                       nx.group_degree_centrality(nx.path_graph(5), [6, 7, 8])

----------------------------------------

File 163 - test_eigenvector_centrality.py:

1: (0)              import math
2: (0)              import pytest
3: (0)              np = pytest.importorskip("numpy")
4: (0)              pytest.importorskip("scipy")
5: (0)              import networkx as nx
6: (0)              class TestEigenvectorCentrality:
7: (4)                  def test_K5(self):
8: (8)                      """Eigenvector centrality: K5"""
9: (8)                      G = nx.complete_graph(5)
10: (8)                     b = nx.eigenvector_centrality(G)
11: (8)                     v = math.sqrt(1 / 5.0)
12: (8)                     b_answer = dict.fromkeys(G, v)
13: (8)                     for n in sorted(G):
14: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
15: (8)                     nstart = {n: 1 for n in G}
16: (8)                     b = nx.eigenvector_centrality(G, nstart=nstart)
17: (8)                     for n in sorted(G):
18: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
19: (8)                     b = nx.eigenvector_centrality_numpy(G)
20: (8)                     for n in sorted(G):
21: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
22: (4)                 def test_P3(self):
23: (8)                     """Eigenvector centrality: P3"""
24: (8)                     G = nx.path_graph(3)
25: (8)                     b_answer = {0: 0.5, 1: 0.7071, 2: 0.5}
26: (8)                     b = nx.eigenvector_centrality_numpy(G)
27: (8)                     for n in sorted(G):
28: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
```

```
(8)                     b = nx.eigenvector_centrality(G)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
(4)                 def test_P3_unweighted(self):
(8)                     """Eigenvector centrality: P3"""
(8)                     G = nx.path_graph(3)
(8)                     b_answer = {0: 0.5, 1: 0.7071, 2: 0.5}
(8)                     b = nx.eigenvector_centrality_numpy(G, weight=None)
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
(4)                 def test_maxiter(self):
(8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
(12)                        G = nx.path_graph(3)
(12)                        nx.eigenvector_centrality(G, max_iter=0)
(0)             class TestEigenvectorCentralityDirected:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     G = nx.DiGraph()
(8)                     edges = [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (2, 4),
(12)                        (3, 2),
(12)                        (3, 5),
(12)                        (4, 2),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (5, 6),
(12)                        (5, 7),
(12)                        (5, 8),
(12)                        (6, 8),
(12)                        (7, 1),
(12)                        (7, 5),
(12)                        (7, 8),
(12)                        (8, 6),
(12)                        (8, 7),
(8)                     ]
(8)                     G.add_edges_from(edges, weight=2.0)
(8)                     cls.G = G.reverse()
(8)                     cls.G.evc = [
(12)                        0.25368793,
(12)                        0.19576478,
(12)                        0.32817092,
(12)                        0.40430835,
(12)                        0.48199885,
(12)                        0.15724483,
(12)                        0.51346196,
(12)                        0.32475403,
(8)                     ]
(8)                     H = nx.DiGraph()
(8)                     edges = [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (2, 4),
(12)                        (3, 2),
(12)                        (3, 5),
(12)                        (4, 2),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (5, 6),
(12)                        (5, 7),
(12)                        (5, 8),
(12)                        (6, 8),
(12)                        (7, 1),
(12)                        (7, 5),
(12)                        (7, 8),
(12)                        (8, 6),
(12)                        (8, 7),
(8)                     ]
```

```
98: (8)                     G.add_edges_from(edges)
99: (8)                     cls.H = G.reverse()
100: (8)                    cls.H.evc = [
101: (12)                       0.25368793,
102: (12)                       0.19576478,
103: (12)                       0.32817092,
104: (12)                       0.40430835,
105: (12)                       0.48199885,
106: (12)                       0.15724483,
107: (12)                       0.51346196,
108: (12)                       0.32475403,
109: (8)                    ]
110: (4)                def test_eigenvector_centrality_weighted(self):
111: (8)                    G = self.G
112: (8)                    p = nx.eigenvector_centrality(G)
113: (8)                    for a, b in zip(list(p.values()), self.G.evc):
114: (12)                       assert a == pytest.approx(b, abs=1e-4)
115: (4)                def test_eigenvector_centrality_weighted_numpy(self):
116: (8)                    G = self.G
117: (8)                    p = nx.eigenvector_centrality_numpy(G)
118: (8)                    for a, b in zip(list(p.values()), self.G.evc):
119: (12)                       assert a == pytest.approx(b, abs=1e-7)
120: (4)                def test_eigenvector_centrality_unweighted(self):
121: (8)                    G = self.H
122: (8)                    p = nx.eigenvector_centrality(G)
123: (8)                    for a, b in zip(list(p.values()), self.G.evc):
124: (12)                       assert a == pytest.approx(b, abs=1e-4)
125: (4)                def test_eigenvector_centrality_unweighted_numpy(self):
126: (8)                    G = self.H
127: (8)                    p = nx.eigenvector_centrality_numpy(G)
128: (8)                    for a, b in zip(list(p.values()), self.G.evc):
129: (12)                       assert a == pytest.approx(b, abs=1e-7)
130: (0)            class TestEigenvectorCentralityExceptions:
131: (4)                def test_multigraph(self):
132: (8)                    with pytest.raises(nx.NetworkXException):
133: (12)                       nx.eigenvector_centrality(nx.MultiGraph())
134: (4)                def test_multigraph_numpy(self):
135: (8)                    with pytest.raises(nx.NetworkXException):
136: (12)                       nx.eigenvector_centrality_numpy(nx.MultiGraph())
137: (4)                def test_empty(self):
138: (8)                    with pytest.raises(nx.NetworkXException):
139: (12)                       nx.eigenvector_centrality(nx.Graph())
140: (4)                def test_empty_numpy(self):
141: (8)                    with pytest.raises(nx.NetworkXException):
142: (12)                       nx.eigenvector_centrality_numpy(nx.Graph())
143: (4)                def test_zero_nstart(self):
144: (8)                    G = nx.Graph([(1, 2), (1, 3), (2, 3)])
145: (8)                    with pytest.raises(
146: (12)                       nx.NetworkXException, match="initial vector cannot have all zero
values"
147: (8)                    ):
148: (12)                       nx.eigenvector_centrality(G, nstart={v: 0 for v in G})
```

File 164 - test_katz_centrality.py:

```
1: (0)              import math
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestKatzCentrality:
5: (4)                  def test_K5(self):
6: (8)                      """Katz centrality: K5"""
7: (8)                      G = nx.complete_graph(5)
8: (8)                      alpha = 0.1
9: (8)                      b = nx.katz_centrality(G, alpha)
10: (8)                     v = math.sqrt(1 / 5.0)
11: (8)                     b_answer = dict.fromkeys(G, v)
12: (8)                     for n in sorted(G):
```

```
13: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
14: (8)                     nstart = {n: 1 for n in G}
15: (8)                     b = nx.katz_centrality(G, alpha, nstart=nstart)
16: (8)                     for n in sorted(G):
17: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
18: (4)                 def test_P3(self):
19: (8)                     """Katz centrality: P3"""
20: (8)                     alpha = 0.1
21: (8)                     G = nx.path_graph(3)
22: (8)                     b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
23: (8)                     b = nx.katz_centrality(G, alpha)
24: (8)                     for n in sorted(G):
25: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
26: (4)                 def test_maxiter(self):
27: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
28: (12)                        nx.katz_centrality(nx.path_graph(3), 0.1, max_iter=0)
29: (4)                 def test_beta_as_scalar(self):
30: (8)                     alpha = 0.1
31: (8)                     beta = 0.1
32: (8)                     b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
33: (8)                     G = nx.path_graph(3)
34: (8)                     b = nx.katz_centrality(G, alpha, beta)
35: (8)                     for n in sorted(G):
36: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
37: (4)                 def test_beta_as_dict(self):
38: (8)                     alpha = 0.1
39: (8)                     beta = {0: 1.0, 1: 1.0, 2: 1.0}
40: (8)                     b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
41: (8)                     G = nx.path_graph(3)
42: (8)                     b = nx.katz_centrality(G, alpha, beta)
43: (8)                     for n in sorted(G):
44: (12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
45: (4)                 def test_multiple_alpha(self):
46: (8)                     alpha_list = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
47: (8)                     for alpha in alpha_list:
48: (12)                        b_answer = {
49: (16)                            0.1: {
50: (20)                                0: 0.5598852584152165,
51: (20)                                1: 0.6107839182711449,
52: (20)                                2: 0.5598852584152162,
53: (16)                            },
54: (16)                            0.2: {
55: (20)                                0: 0.5454545454545454,
56: (20)                                1: 0.6363636363636365,
57: (20)                                2: 0.5454545454545454,
58: (16)                            },
59: (16)                            0.3: {
60: (20)                                0: 0.5333964609104419,
61: (20)                                1: 0.6564879518897746,
62: (20)                                2: 0.5333964609104419,
63: (16)                            },
64: (16)                            0.4: {
65: (20)                                0: 0.5232045649263551,
66: (20)                                1: 0.6726915834767423,
67: (20)                                2: 0.5232045649263551,
68: (16)                            },
69: (16)                            0.5: {
70: (20)                                0: 0.5144957746691622,
71: (20)                                1: 0.6859943117075809,
72: (20)                                2: 0.5144957746691622,
73: (16)                            },
74: (16)                            0.6: {
75: (20)                                0: 0.5069794004195823,
76: (20)                                1: 0.6970966755769258,
77: (20)                                2: 0.5069794004195823,
78: (16)                            },
```

```
79: (12)                        }
80: (12)                        G = nx.path_graph(3)
81: (12)                        b = nx.katz_centrality(G, alpha)
82: (12)                        for n in sorted(G):
83: (16)                            assert b[n] == pytest.approx(b_answer[alpha][n], abs=1e-4)
84: (4)                 def test_multigraph(self):
85: (8)                     with pytest.raises(nx.NetworkXException):
86: (12)                        nx.katz_centrality(nx.MultiGraph(), 0.1)
87: (4)                 def test_empty(self):
88: (8)                     e = nx.katz_centrality(nx.Graph(), 0.1)
89: (8)                     assert e == {}
90: (4)                 def test_bad_beta(self):
91: (8)                     with pytest.raises(nx.NetworkXException):
92: (12)                        G = nx.Graph([(0, 1)])
93: (12)                        beta = {0: 77}
94: (12)                        nx.katz_centrality(G, 0.1, beta=beta)
95: (4)                 def test_bad_beta_number(self):
96: (8)                     with pytest.raises(nx.NetworkXException):
97: (12)                        G = nx.Graph([(0, 1)])
98: (12)                        nx.katz_centrality(G, 0.1, beta="foo")
99: (0)             class TestKatzCentralityNumpy:
100: (4)                @classmethod
101: (4)                def setup_class(cls):
102: (8)                    global np
103: (8)                    np = pytest.importorskip("numpy")
104: (8)                    pytest.importorskip("scipy")
105: (4)                def test_K5(self):
106: (8)                    """Katz centrality: K5"""
107: (8)                    G = nx.complete_graph(5)
108: (8)                    alpha = 0.1
109: (8)                    b = nx.katz_centrality(G, alpha)
110: (8)                    v = math.sqrt(1 / 5.0)
111: (8)                    b_answer = dict.fromkeys(G, v)
112: (8)                    for n in sorted(G):
113: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
114: (8)                    b = nx.eigenvector_centrality_numpy(G)
115: (8)                    for n in sorted(G):
116: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
117: (4)                def test_P3(self):
118: (8)                    """Katz centrality: P3"""
119: (8)                    alpha = 0.1
120: (8)                    G = nx.path_graph(3)
121: (8)                    b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
122: (8)                    b = nx.katz_centrality_numpy(G, alpha)
123: (8)                    for n in sorted(G):
124: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
125: (4)                def test_beta_as_scalar(self):
126: (8)                    alpha = 0.1
127: (8)                    beta = 0.1
128: (8)                    b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
129: (8)                    G = nx.path_graph(3)
130: (8)                    b = nx.katz_centrality_numpy(G, alpha, beta)
131: (8)                    for n in sorted(G):
132: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
133: (4)                def test_beta_as_dict(self):
134: (8)                    alpha = 0.1
135: (8)                    beta = {0: 1.0, 1: 1.0, 2: 1.0}
136: (8)                    b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
137: (8)                    G = nx.path_graph(3)
138: (8)                    b = nx.katz_centrality_numpy(G, alpha, beta)
139: (8)                    for n in sorted(G):
140: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
141: (4)                def test_multiple_alpha(self):
142: (8)                    alpha_list = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
143: (8)                    for alpha in alpha_list:
144: (12)                       b_answer = {
```

```
145: (16)                            0.1: {
146: (20)                                0: 0.5598852584152165,
147: (20)                                1: 0.6107839182711449,
148: (20)                                2: 0.5598852584152162,
149: (16)                            },
150: (16)                            0.2: {
151: (20)                                0: 0.5454545454545454,
152: (20)                                1: 0.6363636363636365,
153: (20)                                2: 0.5454545454545454,
154: (16)                            },
155: (16)                            0.3: {
156: (20)                                0: 0.5333964609104419,
157: (20)                                1: 0.6564879518897746,
158: (20)                                2: 0.5333964609104419,
159: (16)                            },
160: (16)                            0.4: {
161: (20)                                0: 0.5232045649263551,
162: (20)                                1: 0.6726915834767423,
163: (20)                                2: 0.5232045649263551,
164: (16)                            },
165: (16)                            0.5: {
166: (20)                                0: 0.5144957746691622,
167: (20)                                1: 0.6859943117075809,
168: (20)                                2: 0.5144957746691622,
169: (16)                            },
170: (16)                            0.6: {
171: (20)                                0: 0.5069794004195823,
172: (20)                                1: 0.6970966755769258,
173: (20)                                2: 0.5069794004195823,
174: (16)                            },
175: (12)                        }
176: (12)                        G = nx.path_graph(3)
177: (12)                        b = nx.katz_centrality_numpy(G, alpha)
178: (12)                        for n in sorted(G):
179: (16)                            assert b[n] == pytest.approx(b_answer[alpha][n], abs=1e-4)
180: (4)                def test_multigraph(self):
181: (8)                    with pytest.raises(nx.NetworkXException):
182: (12)                       nx.katz_centrality(nx.MultiGraph(), 0.1)
183: (4)                def test_empty(self):
184: (8)                    e = nx.katz_centrality(nx.Graph(), 0.1)
185: (8)                    assert e == {}
186: (4)                def test_bad_beta(self):
187: (8)                    with pytest.raises(nx.NetworkXException):
188: (12)                       G = nx.Graph([(0, 1)])
189: (12)                       beta = {0: 77}
190: (12)                       nx.katz_centrality_numpy(G, 0.1, beta=beta)
191: (4)                def test_bad_beta_numbe(self):
192: (8)                    with pytest.raises(nx.NetworkXException):
193: (12)                       G = nx.Graph([(0, 1)])
194: (12)                       nx.katz_centrality_numpy(G, 0.1, beta="foo")
195: (4)                def test_K5_unweighted(self):
196: (8)                    """Katz centrality: K5"""
197: (8)                    G = nx.complete_graph(5)
198: (8)                    alpha = 0.1
199: (8)                    b = nx.katz_centrality(G, alpha, weight=None)
200: (8)                    v = math.sqrt(1 / 5.0)
201: (8)                    b_answer = dict.fromkeys(G, v)
202: (8)                    for n in sorted(G):
203: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-7)
204: (8)                    b = nx.eigenvector_centrality_numpy(G, weight=None)
205: (8)                    for n in sorted(G):
206: (12)                       assert b[n] == pytest.approx(b_answer[n], abs=1e-3)
207: (4)                def test_P3_unweighted(self):
208: (8)                    """Katz centrality: P3"""
209: (8)                    alpha = 0.1
210: (8)                    G = nx.path_graph(3)
211: (8)                    b_answer = {0: 0.5598852584152165, 1: 0.6107839182711449, 2:
0.5598852584152162}
212: (8)                    b = nx.katz_centrality_numpy(G, alpha, weight=None)
```

```
(8)                     for n in sorted(G):
(12)                        assert b[n] == pytest.approx(b_answer[n], abs=1e-4)
(0)             class TestKatzCentralityDirected:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     G = nx.DiGraph()
(8)                     edges = [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (2, 4),
(12)                        (3, 2),
(12)                        (3, 5),
(12)                        (4, 2),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (5, 6),
(12)                        (5, 7),
(12)                        (5, 8),
(12)                        (6, 8),
(12)                        (7, 1),
(12)                        (7, 5),
(12)                        (7, 8),
(12)                        (8, 6),
(12)                        (8, 7),
(8)                     ]
(8)                     G.add_edges_from(edges, weight=2.0)
(8)                     cls.G = G.reverse()
(8)                     cls.G.alpha = 0.1
(8)                     cls.G.evc = [
(12)                        0.3289589783189635,
(12)                        0.2832077296243516,
(12)                        0.3425906003685471,
(12)                        0.3970420865198392,
(12)                        0.41074871061646284,
(12)                        0.272257430756461,
(12)                        0.4201989685435462,
(12)                        0.34229059218038554,
(8)                     ]
(8)                     H = nx.DiGraph(edges)
(8)                     cls.H = G.reverse()
(8)                     cls.H.alpha = 0.1
(8)                     cls.H.evc = [
(12)                        0.3289589783189635,
(12)                        0.2832077296243516,
(12)                        0.3425906003685471,
(12)                        0.3970420865198392,
(12)                        0.41074871061646284,
(12)                        0.272257430756461,
(12)                        0.4201989685435462,
(12)                        0.34229059218038554,
(8)                     ]
(4)                 def test_katz_centrality_weighted(self):
(8)                     G = self.G
(8)                     alpha = self.G.alpha
(8)                     p = nx.katz_centrality(G, alpha, weight="weight")
(8)                     for a, b in zip(list(p.values()), self.G.evc):
(12)                        assert a == pytest.approx(b, abs=1e-7)
(4)                 def test_katz_centrality_unweighted(self):
(8)                     H = self.H
(8)                     alpha = self.H.alpha
(8)                     p = nx.katz_centrality(H, alpha, weight="weight")
(8)                     for a, b in zip(list(p.values()), self.H.evc):
(12)                        assert a == pytest.approx(b, abs=1e-7)
(0)             class TestKatzCentralityDirectedNumpy(TestKatzCentralityDirected):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     global np
(8)                     np = pytest.importorskip("numpy")
(8)                     pytest.importorskip("scipy")
```

```
282: (8)                    super().setup_class()
283: (4)                def test_katz_centrality_weighted(self):
284: (8)                    G = self.G
285: (8)                    alpha = self.G.alpha
286: (8)                    p = nx.katz_centrality_numpy(G, alpha, weight="weight")
287: (8)                    for a, b in zip(list(p.values()), self.G.evc):
288: (12)                       assert a == pytest.approx(b, abs=1e-7)
289: (4)                def test_katz_centrality_unweighted(self):
290: (8)                    H = self.H
291: (8)                    alpha = self.H.alpha
292: (8)                    p = nx.katz_centrality_numpy(H, alpha, weight="weight")
293: (8)                    for a, b in zip(list(p.values()), self.H.evc):
294: (12)                       assert a == pytest.approx(b, abs=1e-7)
295: (0)            class TestKatzEigenvectorVKatz:
296: (4)                @classmethod
297: (4)                def setup_class(cls):
298: (8)                    global np
299: (8)                    np = pytest.importorskip("numpy")
300: (8)                    pytest.importorskip("scipy")
301: (4)                def test_eigenvector_v_katz_random(self):
302: (8)                    G = nx.gnp_random_graph(10, 0.5, seed=1234)
303: (8)                    l = max(np.linalg.eigvals(nx.adjacency_matrix(G).todense()))
304: (8)                    e = nx.eigenvector_centrality_numpy(G)
305: (8)                    k = nx.katz_centrality_numpy(G, 1.0 / l)
306: (8)                    for n in G:
307: (12)                       assert e[n] == pytest.approx(k[n], abs=1e-7)

----------------------------------------

File 165 - test_harmonic_centrality.py:

1: (0)              """
2: (0)              Tests for degree centrality.
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.centrality import harmonic_centrality
7: (0)              class TestClosenessCentrality:
8: (4)                  @classmethod
9: (4)                  def setup_class(cls):
10: (8)                     cls.P3 = nx.path_graph(3)
11: (8)                     cls.P4 = nx.path_graph(4)
12: (8)                     cls.K5 = nx.complete_graph(5)
13: (8)                     cls.C4 = nx.cycle_graph(4)
14: (8)                     cls.C4_directed = nx.cycle_graph(4, create_using=nx.DiGraph)
15: (8)                     cls.C5 = nx.cycle_graph(5)
16: (8)                     cls.T = nx.balanced_tree(r=2, h=2)
17: (8)                     cls.Gb = nx.DiGraph()
18: (8)                     cls.Gb.add_edges_from([(0, 1), (0, 2), (0, 4), (2, 1), (2, 3), (4,
3)])
19: (4)                 def test_p3_harmonic(self):
20: (8)                     c = harmonic_centrality(self.P3)
21: (8)                     d = {0: 1.5, 1: 2, 2: 1.5}
22: (8)                     for n in sorted(self.P3):
23: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
24: (4)                 def test_p4_harmonic(self):
25: (8)                     c = harmonic_centrality(self.P4)
26: (8)                     d = {0: 1.8333333, 1: 2.5, 2: 2.5, 3: 1.8333333}
27: (8)                     for n in sorted(self.P4):
28: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
29: (4)                 def test_clique_complete(self):
30: (8)                     c = harmonic_centrality(self.K5)
31: (8)                     d = {0: 4, 1: 4, 2: 4, 3: 4, 4: 4}
32: (8)                     for n in sorted(self.P3):
33: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
34: (4)                 def test_cycle_C4(self):
35: (8)                     c = harmonic_centrality(self.C4)
36: (8)                     d = {0: 2.5, 1: 2.5, 2: 2.5, 3: 2.5}
37: (8)                     for n in sorted(self.C4):
```

```
38: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
39: (4)                 def test_cycle_C5(self):
40: (8)                     c = harmonic_centrality(self.C5)
41: (8)                     d = {0: 3, 1: 3, 2: 3, 3: 3, 4: 3, 5: 4}
42: (8)                     for n in sorted(self.C5):
43: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
44: (4)                 def test_bal_tree(self):
45: (8)                     c = harmonic_centrality(self.T)
46: (8)                     d = {0: 4.0, 1: 4.1666, 2: 4.1666, 3: 2.8333, 4: 2.8333, 5: 2.8333, 6:
2.8333}
47: (8)                     for n in sorted(self.T):
48: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
49: (4)                 def test_exampleGraph(self):
50: (8)                     c = harmonic_centrality(self.Gb)
51: (8)                     d = {0: 0, 1: 2, 2: 1, 3: 2.5, 4: 1}
52: (8)                     for n in sorted(self.Gb):
53: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
54: (4)                 def test_weighted_harmonic(self):
55: (8)                     XG = nx.DiGraph()
56: (8)                     XG.add_weighted_edges_from(
57: (12)                        [
58: (16)                            ("a", "b", 10),
59: (16)                            ("d", "c", 5),
60: (16)                            ("a", "c", 1),
61: (16)                            ("e", "f", 2),
62: (16)                            ("f", "c", 1),
63: (16)                            ("a", "f", 3),
64: (12)                        ]
65: (8)                     )
66: (8)                     c = harmonic_centrality(XG, distance="weight")
67: (8)                     d = {"a": 0, "b": 0.1, "c": 2.533, "d": 0, "e": 0, "f": 0.83333}
68: (8)                     for n in sorted(XG):
69: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
70: (4)                 def test_empty(self):
71: (8)                     G = nx.DiGraph()
72: (8)                     c = harmonic_centrality(G, distance="weight")
73: (8)                     d = {}
74: (8)                     assert c == d
75: (4)                 def test_singleton(self):
76: (8)                     G = nx.DiGraph()
77: (8)                     G.add_node(0)
78: (8)                     c = harmonic_centrality(G, distance="weight")
79: (8)                     d = {0: 0}
80: (8)                     assert c == d
81: (4)                 def test_cycle_c4_directed(self):
82: (8)                     c = harmonic_centrality(self.C4_directed, nbunch=[0, 1], sources=[1,
2])
83: (8)                     d = {0: 0.833, 1: 0.333}
84: (8)                     for n in [0, 1]:
85: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
86: (4)                 def test_p3_harmonic_subset(self):
87: (8)                     c = harmonic_centrality(self.P3, sources=[0, 1])
88: (8)                     d = {0: 1, 1: 1, 2: 1.5}
89: (8)                     for n in self.P3:
90: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
91: (4)                 def test_p4_harmonic_subset(self):
92: (8)                     c = harmonic_centrality(self.P4, nbunch=[2, 3], sources=[0, 1])
93: (8)                     d = {2: 1.5, 3: 0.8333333}
94: (8)                     for n in [2, 3]:
95: (12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)

----------------------------------------

File 166 - test_load_centrality.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestLoadCentrality:
4: (4)                  @classmethod
```

```
(4)     def setup_class(cls):
(8)         G = nx.Graph()
(8)         G.add_edge(0, 1, weight=3)
(8)         G.add_edge(0, 2, weight=2)
(8)         G.add_edge(0, 3, weight=6)
(8)         G.add_edge(0, 4, weight=4)
(8)         G.add_edge(1, 3, weight=5)
(8)         G.add_edge(1, 5, weight=5)
(8)         G.add_edge(2, 4, weight=1)
(8)         G.add_edge(3, 4, weight=2)
(8)         G.add_edge(3, 5, weight=1)
(8)         G.add_edge(4, 5, weight=4)
(8)         cls.G = G
(8)         cls.exact_weighted = {0: 4.0, 1: 0.0, 2: 8.0, 3: 6.0, 4: 8.0, 5: 0.0}
(8)         cls.K = nx.krackhardt_kite_graph()
(8)         cls.P3 = nx.path_graph(3)
(8)         cls.P4 = nx.path_graph(4)
(8)         cls.K5 = nx.complete_graph(5)
(8)         cls.P2 = nx.path_graph(2)
(8)         cls.C4 = nx.cycle_graph(4)
(8)         cls.T = nx.balanced_tree(r=2, h=2)
(8)         cls.Gb = nx.Graph()
(8)         cls.Gb.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (2, 4), (4, 5),
(3, 5)])
(8)         cls.F = nx.florentine_families_graph()
(8)         cls.LM = nx.les_miserables_graph()
(8)         cls.D = nx.cycle_graph(3, create_using=nx.DiGraph())
(8)         cls.D.add_edges_from([(3, 0), (4, 3)])
(4)     def test_not_strongly_connected(self):
(8)         b = nx.load_centrality(self.D)
(8)         result = {0: 5.0 / 12, 1: 1.0 / 4, 2: 1.0 / 12, 3: 1.0 / 4, 4: 0.000}
(8)         for n in sorted(self.D):
(12)            assert result[n] == pytest.approx(b[n], abs=1e-3)
(12)            assert result[n] == pytest.approx(nx.load_centrality(self.D, n),
abs=1e-3)
(4)     def test_P2_normalized_load(self):
(8)         G = self.P2
(8)         c = nx.load_centrality(G, normalized=True)
(8)         d = {0: 0.000, 1: 0.000}
(8)         for n in sorted(G):
(12)            assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)     def test_weighted_load(self):
(8)         b = nx.load_centrality(self.G, weight="weight", normalized=False)
(8)         for n in sorted(self.G):
(12)            assert b[n] == self.exact_weighted[n]
(4)     def test_k5_load(self):
(8)         G = self.K5
(8)         c = nx.load_centrality(G)
(8)         d = {0: 0.000, 1: 0.000, 2: 0.000, 3: 0.000, 4: 0.000}
(8)         for n in sorted(G):
(12)            assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)     def test_p3_load(self):
(8)         G = self.P3
(8)         c = nx.load_centrality(G)
(8)         d = {0: 0.000, 1: 1.000, 2: 0.000}
(8)         for n in sorted(G):
(12)            assert c[n] == pytest.approx(d[n], abs=1e-3)
(8)         c = nx.load_centrality(G, v=1)
(8)         assert c == pytest.approx(1.0, abs=1e-7)
(8)         c = nx.load_centrality(G, v=1, normalized=True)
(8)         assert c == pytest.approx(1.0, abs=1e-7)
(4)     def test_p2_load(self):
(8)         G = nx.path_graph(2)
(8)         c = nx.load_centrality(G)
(8)         d = {0: 0.000, 1: 0.000}
(8)         for n in sorted(G):
(12)            assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)     def test_krackhardt_load(self):
(8)         G = self.K
```

```
(8)                     c = nx.load_centrality(G)
(8)                     d = {
(12)                        0: 0.023,
(12)                        1: 0.023,
(12)                        2: 0.000,
(12)                        3: 0.102,
(12)                        4: 0.000,
(12)                        5: 0.231,
(12)                        6: 0.231,
(12)                        7: 0.389,
(12)                        8: 0.222,
(12)                        9: 0.000,
(8)                     }
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_florentine_families_load(self):
(8)                     G = self.F
(8)                     c = nx.load_centrality(G)
(8)                     d = {
(12)                        "Acciaiuoli": 0.000,
(12)                        "Albizzi": 0.211,
(12)                        "Barbadori": 0.093,
(12)                        "Bischeri": 0.104,
(12)                        "Castellani": 0.055,
(12)                        "Ginori": 0.000,
(12)                        "Guadagni": 0.251,
(12)                        "Lamberteschi": 0.000,
(12)                        "Medici": 0.522,
(12)                        "Pazzi": 0.000,
(12)                        "Peruzzi": 0.022,
(12)                        "Ridolfi": 0.117,
(12)                        "Salviati": 0.143,
(12)                        "Strozzi": 0.106,
(12)                        "Tornabuoni": 0.090,
(8)                     }
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_les_miserables_load(self):
(8)                     G = self.LM
(8)                     c = nx.load_centrality(G)
(8)                     d = {
(12)                        "Napoleon": 0.000,
(12)                        "Myriel": 0.177,
(12)                        "MlleBaptistine": 0.000,
(12)                        "MmeMagloire": 0.000,
(12)                        "CountessDeLo": 0.000,
(12)                        "Geborand": 0.000,
(12)                        "Champtercier": 0.000,
(12)                        "Cravatte": 0.000,
(12)                        "Count": 0.000,
(12)                        "OldMan": 0.000,
(12)                        "Valjean": 0.567,
(12)                        "Labarre": 0.000,
(12)                        "Marguerite": 0.000,
(12)                        "MmeDeR": 0.000,
(12)                        "Isabeau": 0.000,
(12)                        "Gervais": 0.000,
(12)                        "Listolier": 0.000,
(12)                        "Tholomyes": 0.043,
(12)                        "Fameuil": 0.000,
(12)                        "Blacheville": 0.000,
(12)                        "Favourite": 0.000,
(12)                        "Dahlia": 0.000,
(12)                        "Zephine": 0.000,
(12)                        "Fantine": 0.128,
(12)                        "MmeThenardier": 0.029,
(12)                        "Thenardier": 0.075,
(12)                        "Cosette": 0.024,
(12)                        "Javert": 0.054,
```

```
(12)                        "Fauchelevent": 0.026,
(12)                        "Bamatabois": 0.008,
(12)                        "Perpetue": 0.000,
(12)                        "Simplice": 0.009,
(12)                        "Scaufflaire": 0.000,
(12)                        "Woman1": 0.000,
(12)                        "Judge": 0.000,
(12)                        "Champmathieu": 0.000,
(12)                        "Brevet": 0.000,
(12)                        "Chenildieu": 0.000,
(12)                        "Cochepaille": 0.000,
(12)                        "Pontmercy": 0.007,
(12)                        "Boulatruelle": 0.000,
(12)                        "Eponine": 0.012,
(12)                        "Anzelma": 0.000,
(12)                        "Woman2": 0.000,
(12)                        "MotherInnocent": 0.000,
(12)                        "Gribier": 0.000,
(12)                        "MmeBurgon": 0.026,
(12)                        "Jondrette": 0.000,
(12)                        "Gavroche": 0.164,
(12)                        "Gillenormand": 0.021,
(12)                        "Magnon": 0.000,
(12)                        "MlleGillenormand": 0.047,
(12)                        "MmePontmercy": 0.000,
(12)                        "MlleVaubois": 0.000,
(12)                        "LtGillenormand": 0.000,
(12)                        "Marius": 0.133,
(12)                        "BaronessT": 0.000,
(12)                        "Mabeuf": 0.028,
(12)                        "Enjolras": 0.041,
(12)                        "Combeferre": 0.001,
(12)                        "Prouvaire": 0.000,
(12)                        "Feuilly": 0.001,
(12)                        "Courfeyrac": 0.006,
(12)                        "Bahorel": 0.002,
(12)                        "Bossuet": 0.032,
(12)                        "Joly": 0.002,
(12)                        "Grantaire": 0.000,
(12)                        "MotherPlutarch": 0.000,
(12)                        "Gueulemer": 0.005,
(12)                        "Babet": 0.005,
(12)                        "Claquesous": 0.005,
(12)                        "Montparnasse": 0.004,
(12)                        "Toussaint": 0.000,
(12)                        "Child1": 0.000,
(12)                        "Child2": 0.000,
(12)                        "Brujon": 0.000,
(12)                        "MmeHucheloup": 0.000,
(8)                     }
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_unnormalized_k5_load(self):
(8)                     G = self.K5
(8)                     c = nx.load_centrality(G, normalized=False)
(8)                     d = {0: 0.000, 1: 0.000, 2: 0.000, 3: 0.000, 4: 0.000}
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_unnormalized_p3_load(self):
(8)                     G = self.P3
(8)                     c = nx.load_centrality(G, normalized=False)
(8)                     d = {0: 0.000, 1: 2.000, 2: 0.000}
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_unnormalized_krackhardt_load(self):
(8)                     G = self.K
(8)                     c = nx.load_centrality(G, normalized=False)
(8)                     d = {
(12)                        0: 1.667,
```

```
(12)                        1: 1.667,
(12)                        2: 0.000,
(12)                        3: 7.333,
(12)                        4: 0.000,
(12)                        5: 16.667,
(12)                        6: 16.667,
(12)                        7: 28.000,
(12)                        8: 16.000,
(12)                        9: 0.000,
(8)                     }
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_unnormalized_florentine_families_load(self):
(8)                     G = self.F
(8)                     c = nx.load_centrality(G, normalized=False)
(8)                     d = {
(12)                        "Acciaiuoli": 0.000,
(12)                        "Albizzi": 38.333,
(12)                        "Barbadori": 17.000,
(12)                        "Bischeri": 19.000,
(12)                        "Castellani": 10.000,
(12)                        "Ginori": 0.000,
(12)                        "Guadagni": 45.667,
(12)                        "Lamberteschi": 0.000,
(12)                        "Medici": 95.000,
(12)                        "Pazzi": 0.000,
(12)                        "Peruzzi": 4.000,
(12)                        "Ridolfi": 21.333,
(12)                        "Salviati": 26.000,
(12)                        "Strozzi": 19.333,
(12)                        "Tornabuoni": 16.333,
(8)                     }
(8)                     for n in sorted(G):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_load_betweenness_difference(self):
(8)                     B = nx.Graph()  # ladder_graph(3)
(8)                     B.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (2, 4), (4, 5), (3,
5)])
(8)                     c = nx.load_centrality(B, normalized=False)
(8)                     d = {0: 1.750, 1: 1.750, 2: 6.500, 3: 6.500, 4: 1.750, 5: 1.750}
(8)                     for n in sorted(B):
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_c4_edge_load(self):
(8)                     G = self.C4
(8)                     c = nx.edge_load_centrality(G)
(8)                     d = {(0, 1): 6.000, (0, 3): 6.000, (1, 2): 6.000, (2, 3): 6.000}
(8)                     for n in G.edges():
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_p4_edge_load(self):
(8)                     G = self.P4
(8)                     c = nx.edge_load_centrality(G)
(8)                     d = {(0, 1): 6.000, (1, 2): 8.000, (2, 3): 6.000}
(8)                     for n in G.edges():
(12)                        assert c[n] == pytest.approx(d[n], abs=1e-3)
(4)                 def test_k5_edge_load(self):
(8)                     G = self.K5
(8)                     c = nx.edge_load_centrality(G)
(8)                     d = {
(12)                        (0, 1): 5.000,
(12)                        (0, 2): 5.000,
(12)                        (0, 3): 5.000,
(12)                        (0, 4): 5.000,
(12)                        (1, 2): 5.000,
(12)                        (1, 3): 5.000,
(12)                        (1, 4): 5.000,
(12)                        (2, 3): 5.000,
(12)                        (2, 4): 5.000,
(12)                        (3, 4): 5.000,
(8)                     }
```

```
278: (8)                    for n in G.edges():
279: (12)                       assert c[n] == pytest.approx(d[n], abs=1e-3)
280: (4)                def test_tree_edge_load(self):
281: (8)                    G = self.T
282: (8)                    c = nx.edge_load_centrality(G)
283: (8)                    d = {
284: (12)                       (0, 1): 24.000,
285: (12)                       (0, 2): 24.000,
286: (12)                       (1, 3): 12.000,
287: (12)                       (1, 4): 12.000,
288: (12)                       (2, 5): 12.000,
289: (12)                       (2, 6): 12.000,
290: (8)                    }
291: (8)                    for n in G.edges():
292: (12)                       assert c[n] == pytest.approx(d[n], abs=1e-3)
```

File 167 - test_laplacian_centrality.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              np = pytest.importorskip("numpy")
4: (0)              sp = pytest.importorskip("scipy")
5: (0)              def test_laplacian_centrality_null_graph():
6: (4)                  G = nx.Graph()
7: (4)                  with pytest.raises(nx.NetworkXPointlessConcept):
8: (8)                      d = nx.laplacian_centrality(G, normalized=False)
9: (0)              def test_laplacian_centrality_single_node():
10: (4)                 """See gh-6571"""
11: (4)                 G = nx.empty_graph(1)
12: (4)                 assert nx.laplacian_centrality(G, normalized=False) == {0: 0}
13: (4)                 with pytest.raises(ZeroDivisionError):
14: (8)                     nx.laplacian_centrality(G, normalized=True)
15: (0)             def test_laplacian_centrality_unconnected_nodes():
16: (4)                 """laplacian_centrality on a unconnected node graph should return 0
17: (4)                 For graphs without edges, the Laplacian energy is 0 and is unchanged with
18: (4)                 node removal, so::
19: (8)                     LC(v) = LE(G) - LE(G - v) = 0 - 0 = 0
20: (4)                 """
21: (4)                 G = nx.empty_graph(3)
22: (4)                 assert nx.laplacian_centrality(G, normalized=False) == {0: 0, 1: 0, 2: 0}
23: (0)             def test_laplacian_centrality_empty_graph():
24: (4)                 G = nx.empty_graph(3)
25: (4)                 with pytest.raises(ZeroDivisionError):
26: (8)                     d = nx.laplacian_centrality(G, normalized=True)
27: (0)             def test_laplacian_centrality_E():
28: (4)                 E = nx.Graph()
29: (4)                 E.add_weighted_edges_from(
30: (8)                     [(0, 1, 4), (4, 5, 1), (0, 2, 2), (2, 1, 1), (1, 3, 2), (1, 4, 2)]
31: (4)                 )
32: (4)                 d = nx.laplacian_centrality(E)
33: (4)                 exact = {
34: (8)                     0: 0.700000,
35: (8)                     1: 0.900000,
36: (8)                     2: 0.280000,
37: (8)                     3: 0.220000,
38: (8)                     4: 0.260000,
39: (8)                     5: 0.040000,
40: (4)                 }
41: (4)                 for n, dc in d.items():
42: (8)                     assert exact[n] == pytest.approx(dc, abs=1e-7)
43: (4)                 full_energy = 200
44: (4)                 dnn = nx.laplacian_centrality(E, normalized=False)
45: (4)                 for n, dc in dnn.items():
46: (8)                     assert exact[n] * full_energy == pytest.approx(dc, abs=1e-7)
47: (4)                 duw_nn = nx.laplacian_centrality(E, normalized=False, weight=None)
48: (4)                 print(duw_nn)
49: (4)                 exact_uw_nn = {
```

```
50: (8)                     0: 18,
51: (8)                     1: 34,
52: (8)                     2: 18,
53: (8)                     3: 10,
54: (8)                     4: 16,
55: (8)                     5: 6,
56: (4)                 }
57: (4)                 for n, dc in duw_nn.items():
58: (8)                     assert exact_uw_nn[n] == pytest.approx(dc, abs=1e-7)
59: (4)                 duw = nx.laplacian_centrality(E, weight=None)
60: (4)                 full_energy = 42
61: (4)                 for n, dc in duw.items():
62: (8)                     assert exact_uw_nn[n] / full_energy == pytest.approx(dc, abs=1e-7)
63: (0)             def test_laplacian_centrality_KC():
64: (4)                 KC = nx.karate_club_graph()
65: (4)                 d = nx.laplacian_centrality(KC)
66: (4)                 exact = {
67: (8)                     0: 0.2543593,
68: (8)                     1: 0.1724524,
69: (8)                     2: 0.2166053,
70: (8)                     3: 0.0964646,
71: (8)                     4: 0.0350344,
72: (8)                     5: 0.0571109,
73: (8)                     6: 0.0540713,
74: (8)                     7: 0.0788674,
75: (8)                     8: 0.1222204,
76: (8)                     9: 0.0217565,
77: (8)                     10: 0.0308751,
78: (8)                     11: 0.0215965,
79: (8)                     12: 0.0174372,
80: (8)                     13: 0.118861,
81: (8)                     14: 0.0366341,
82: (8)                     15: 0.0548712,
83: (8)                     16: 0.0172772,
84: (8)                     17: 0.0191969,
85: (8)                     18: 0.0225564,
86: (8)                     19: 0.0331147,
87: (8)                     20: 0.0279955,
88: (8)                     21: 0.0246361,
89: (8)                     22: 0.0382339,
90: (8)                     23: 0.1294193,
91: (8)                     24: 0.0227164,
92: (8)                     25: 0.0644697,
93: (8)                     26: 0.0281555,
94: (8)                     27: 0.075188,
95: (8)                     28: 0.0364742,
96: (8)                     29: 0.0707087,
97: (8)                     30: 0.0708687,
98: (8)                     31: 0.131019,
99: (8)                     32: 0.2370821,
100: (8)                    33: 0.3066709,
101: (4)                }
102: (4)                for n, dc in d.items():
103: (8)                    assert exact[n] == pytest.approx(dc, abs=1e-7)
104: (4)                full_energy = 12502
105: (4)                dnn = nx.laplacian_centrality(KC, normalized=False)
106: (4)                for n, dc in dnn.items():
107: (8)                    assert exact[n] * full_energy == pytest.approx(dc, abs=1e-3)
108: (0)            def test_laplacian_centrality_K():
109: (4)                K = nx.krackhardt_kite_graph()
110: (4)                d = nx.laplacian_centrality(K)
111: (4)                exact = {
112: (8)                    0: 0.3010753,
113: (8)                    1: 0.3010753,
114: (8)                    2: 0.2258065,
115: (8)                    3: 0.483871,
116: (8)                    4: 0.2258065,
117: (8)                    5: 0.3870968,
118: (8)                    6: 0.3870968,
```

```
(8)                     7: 0.1935484,
(8)                     8: 0.0752688,
(8)                     9: 0.0322581,
(4)                 }
(4)                 for n, dc in d.items():
(8)                     assert exact[n] == pytest.approx(dc, abs=1e-7)
(4)                 full_energy = 186
(4)                 dnn = nx.laplacian_centrality(K, normalized=False)
(4)                 for n, dc in dnn.items():
(8)                     assert exact[n] * full_energy == pytest.approx(dc, abs=1e-3)
(0)             def test_laplacian_centrality_P3():
(4)                 P3 = nx.path_graph(3)
(4)                 d = nx.laplacian_centrality(P3)
(4)                 exact = {0: 0.6, 1: 1.0, 2: 0.6}
(4)                 for n, dc in d.items():
(8)                     assert exact[n] == pytest.approx(dc, abs=1e-7)
(0)             def test_laplacian_centrality_K5():
(4)                 K5 = nx.complete_graph(5)
(4)                 d = nx.laplacian_centrality(K5)
(4)                 exact = {0: 0.52, 1: 0.52, 2: 0.52, 3: 0.52, 4: 0.52}
(4)                 for n, dc in d.items():
(8)                     assert exact[n] == pytest.approx(dc, abs=1e-7)
(0)             def test_laplacian_centrality_FF():
(4)                 FF = nx.florentine_families_graph()
(4)                 d = nx.laplacian_centrality(FF)
(4)                 exact = {
(8)                     "Acciaiuoli": 0.0804598,
(8)                     "Medici": 0.4022989,
(8)                     "Castellani": 0.1724138,
(8)                     "Peruzzi": 0.183908,
(8)                     "Strozzi": 0.2528736,
(8)                     "Barbadori": 0.137931,
(8)                     "Ridolfi": 0.2183908,
(8)                     "Tornabuoni": 0.2183908,
(8)                     "Albizzi": 0.1954023,
(8)                     "Salviati": 0.1149425,
(8)                     "Pazzi": 0.0344828,
(8)                     "Bischeri": 0.1954023,
(8)                     "Guadagni": 0.2298851,
(8)                     "Ginori": 0.045977,
(8)                     "Lamberteschi": 0.0574713,
(4)                 }
(4)                 for n, dc in d.items():
(8)                     assert exact[n] == pytest.approx(dc, abs=1e-7)
(0)             def test_laplacian_centrality_DG():
(4)                 DG = nx.DiGraph([(0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 6), (5, 7),
(5, 8)])
(4)                 d = nx.laplacian_centrality(DG)
(4)                 exact = {
(8)                     0: 0.2123352,
(8)                     5: 0.515391,
(8)                     1: 0.2123352,
(8)                     2: 0.2123352,
(8)                     3: 0.2123352,
(8)                     4: 0.2123352,
(8)                     6: 0.2952031,
(8)                     7: 0.2952031,
(8)                     8: 0.2952031,
(4)                 }
(4)                 for n, dc in d.items():
(8)                     assert exact[n] == pytest.approx(dc, abs=1e-7)
(4)                 full_energy = 9.50704
(4)                 dnn = nx.laplacian_centrality(DG, normalized=False)
(4)                 for n, dc in dnn.items():
(8)                     assert exact[n] * full_energy == pytest.approx(dc, abs=1e-4)

----------------------------------------

File 168 - test_reaching.py:
```

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.centrality.reaching`
module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestGlobalReachingCentrality:
5: (4)                  """Unit tests for the global reaching centrality function."""
6: (4)                  def test_non_positive_weights(self):
7: (8)                      with pytest.raises(nx.NetworkXError):
8: (12)                         G = nx.DiGraph()
9: (12)                         nx.global_reaching_centrality(G, weight="weight")
10: (4)                 def test_negatively_weighted(self):
11: (8)                     with pytest.raises(nx.NetworkXError):
12: (12)                        G = nx.Graph()
13: (12)                        G.add_weighted_edges_from([(0, 1, -2), (1, 2, +1)])
14: (12)                        nx.global_reaching_centrality(G, weight="weight")
15: (4)                 def test_directed_star(self):
16: (8)                     G = nx.DiGraph()
17: (8)                     G.add_weighted_edges_from([(1, 2, 0.5), (1, 3, 0.5)])
18: (8)                     grc = nx.global_reaching_centrality
19: (8)                     assert grc(G, normalized=False, weight="weight") == 0.5
20: (8)                     assert grc(G) == 1
21: (4)                 def test_undirected_unweighted_star(self):
22: (8)                     G = nx.star_graph(2)
23: (8)                     grc = nx.global_reaching_centrality
24: (8)                     assert grc(G, normalized=False, weight=None) == 0.25
25: (4)                 def test_undirected_weighted_star(self):
26: (8)                     G = nx.Graph()
27: (8)                     G.add_weighted_edges_from([(1, 2, 1), (1, 3, 2)])
28: (8)                     grc = nx.global_reaching_centrality
29: (8)                     assert grc(G, normalized=False, weight="weight") == 0.375
30: (4)                 def test_cycle_directed_unweighted(self):
31: (8)                     G = nx.DiGraph()
32: (8)                     G.add_edge(1, 2)
33: (8)                     G.add_edge(2, 1)
34: (8)                     assert nx.global_reaching_centrality(G, weight=None) == 0
35: (4)                 def test_cycle_undirected_unweighted(self):
36: (8)                     G = nx.Graph()
37: (8)                     G.add_edge(1, 2)
38: (8)                     assert nx.global_reaching_centrality(G, weight=None) == 0
39: (4)                 def test_cycle_directed_weighted(self):
40: (8)                     G = nx.DiGraph()
41: (8)                     G.add_weighted_edges_from([(1, 2, 1), (2, 1, 1)])
42: (8)                     assert nx.global_reaching_centrality(G) == 0
43: (4)                 def test_cycle_undirected_weighted(self):
44: (8)                     G = nx.Graph()
45: (8)                     G.add_edge(1, 2, weight=1)
46: (8)                     grc = nx.global_reaching_centrality
47: (8)                     assert grc(G, normalized=False) == 0
48: (4)                 def test_directed_weighted(self):
49: (8)                     G = nx.DiGraph()
50: (8)                     G.add_edge("A", "B", weight=5)
51: (8)                     G.add_edge("B", "C", weight=1)
52: (8)                     G.add_edge("B", "D", weight=0.25)
53: (8)                     G.add_edge("D", "E", weight=1)
54: (8)                     denom = len(G) - 1
55: (8)                     A_local = sum([5, 3, 2.625, 2.0833333333333]) / denom
56: (8)                     B_local = sum([1, 0.25, 0.625]) / denom
57: (8)                     C_local = 0
58: (8)                     D_local = sum([1]) / denom
59: (8)                     E_local = 0
60: (8)                     local_reach_ctrs = [A_local, C_local, B_local, D_local, E_local]
61: (8)                     max_local = max(local_reach_ctrs)
62: (8)                     expected = sum(max_local - lrc for lrc in local_reach_ctrs) / denom
63: (8)                     grc = nx.global_reaching_centrality
64: (8)                     actual = grc(G, normalized=False, weight="weight")
65: (8)                     assert expected == pytest.approx(actual, abs=1e-7)
66: (0)             class TestLocalReachingCentrality:
67: (4)                 """Unit tests for the local reaching centrality function."""
```

```
68: (4)                 def test_non_positive_weights(self):
69: (8)                     with pytest.raises(nx.NetworkXError):
70: (12)                        G = nx.DiGraph()
71: (12)                        G.add_weighted_edges_from([(0, 1, 0)])
72: (12)                        nx.local_reaching_centrality(G, 0, weight="weight")
73: (4)                 def test_negatively_weighted(self):
74: (8)                     with pytest.raises(nx.NetworkXError):
75: (12)                        G = nx.Graph()
76: (12)                        G.add_weighted_edges_from([(0, 1, -2), (1, 2, +1)])
77: (12)                        nx.local_reaching_centrality(G, 0, weight="weight")
78: (4)                 def test_undirected_unweighted_star(self):
79: (8)                     G = nx.star_graph(2)
80: (8)                     grc = nx.local_reaching_centrality
81: (8)                     assert grc(G, 1, weight=None, normalized=False) == 0.75
82: (4)                 def test_undirected_weighted_star(self):
83: (8)                     G = nx.Graph()
84: (8)                     G.add_weighted_edges_from([(1, 2, 1), (1, 3, 2)])
85: (8)                     centrality = nx.local_reaching_centrality(
86: (12)                        G, 1, normalized=False, weight="weight"
87: (8)                     )
88: (8)                     assert centrality == 1.5
89: (4)                 def test_undirected_weighted_normalized(self):
90: (8)                     G = nx.Graph()
91: (8)                     G.add_weighted_edges_from([(1, 2, 1), (1, 3, 2)])
92: (8)                     centrality = nx.local_reaching_centrality(
93: (12)                        G, 1, normalized=True, weight="weight"
94: (8)                     )
95: (8)                     assert centrality == 1.0

----------------------------------------

File 169 - test_percolation_centrality.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def example1a_G():
4: (4)                  G = nx.Graph()
5: (4)                  G.add_node(1, percolation=0.1)
6: (4)                  G.add_node(2, percolation=0.2)
7: (4)                  G.add_node(3, percolation=0.2)
8: (4)                  G.add_node(4, percolation=0.2)
9: (4)                  G.add_node(5, percolation=0.3)
10: (4)                 G.add_node(6, percolation=0.2)
11: (4)                 G.add_node(7, percolation=0.5)
12: (4)                 G.add_node(8, percolation=0.5)
13: (4)                 G.add_edges_from([(1, 4), (2, 4), (3, 4), (4, 5), (5, 6), (6, 7), (6, 8)])
14: (4)                 return G
15: (0)             def example1b_G():
16: (4)                 G = nx.Graph()
17: (4)                 G.add_node(1, percolation=0.3)
18: (4)                 G.add_node(2, percolation=0.5)
19: (4)                 G.add_node(3, percolation=0.5)
20: (4)                 G.add_node(4, percolation=0.2)
21: (4)                 G.add_node(5, percolation=0.3)
22: (4)                 G.add_node(6, percolation=0.2)
23: (4)                 G.add_node(7, percolation=0.1)
24: (4)                 G.add_node(8, percolation=0.1)
25: (4)                 G.add_edges_from([(1, 4), (2, 4), (3, 4), (4, 5), (5, 6), (6, 7), (6, 8)])
26: (4)                 return G
27: (0)             def test_percolation_example1a():
28: (4)                 """percolation centrality: example 1a"""
29: (4)                 G = example1a_G()
30: (4)                 p = nx.percolation_centrality(G)
31: (4)                 p_answer = {4: 0.625, 6: 0.667}
32: (4)                 for n, k in p_answer.items():
33: (8)                     assert p[n] == pytest.approx(k, abs=1e-3)
34: (0)             def test_percolation_example1b():
35: (4)                 """percolation centrality: example 1a"""
36: (4)                 G = example1b_G()
```

```
37: (4)                 p = nx.percolation_centrality(G)
38: (4)                 p_answer = {4: 0.825, 6: 0.4}
39: (4)                 for n, k in p_answer.items():
40: (8)                     assert p[n] == pytest.approx(k, abs=1e-3)
41: (0)             def test_converge_to_betweenness():
42: (4)                 """percolation centrality: should converge to betweenness
43: (4)                 centrality when all nodes are percolated the same"""
44: (4)                 G = nx.florentine_families_graph()
45: (4)                 b_answer = {
46: (8)                     "Acciaiuoli": 0.000,
47: (8)                     "Albizzi": 0.212,
48: (8)                     "Barbadori": 0.093,
49: (8)                     "Bischeri": 0.104,
50: (8)                     "Castellani": 0.055,
51: (8)                     "Ginori": 0.000,
52: (8)                     "Guadagni": 0.255,
53: (8)                     "Lamberteschi": 0.000,
54: (8)                     "Medici": 0.522,
55: (8)                     "Pazzi": 0.000,
56: (8)                     "Peruzzi": 0.022,
57: (8)                     "Ridolfi": 0.114,
58: (8)                     "Salviati": 0.143,
59: (8)                     "Strozzi": 0.103,
60: (8)                     "Tornabuoni": 0.092,
61: (4)                 }
62: (4)                 p_answer = nx.percolation_centrality(G)
63: (4)                 assert p_answer == pytest.approx(b_answer, abs=1e-3)
64: (4)                 p_states = {k: 0.3 for k, v in b_answer.items()}
65: (4)                 p_answer = nx.percolation_centrality(G, states=p_states)
66: (4)                 assert p_answer == pytest.approx(b_answer, abs=1e-3)
67: (0)             def test_default_percolation():
68: (4)                 G = nx.erdos_renyi_graph(42, 0.42, seed=42)
69: (4)                 assert nx.percolation_centrality(G) ==
pytest.approx(nx.betweenness_centrality(G))

----------------------------------------

File 170 - test_subgraph.py:

1: (0)              import pytest
2: (0)              pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.centrality.subgraph_alg import (
6: (4)                  communicability_betweenness_centrality,
7: (4)                  estrada_index,
8: (4)                  subgraph_centrality,
9: (4)                  subgraph_centrality_exp,
10: (0)             )
11: (0)             class TestSubgraph:
12: (4)                 def test_subgraph_centrality(self):
13: (8)                     answer = {0: 1.5430806348152433, 1: 1.5430806348152433}
14: (8)                     result = subgraph_centrality(nx.path_graph(2))
15: (8)                     for k, v in result.items():
16: (12)                        assert answer[k] == pytest.approx(v, abs=1e-7)
17: (8)                     answer1 = {
18: (12)                        "1": 1.6445956054135658,
19: (12)                        "Albert": 2.4368257358712189,
20: (12)                        "Aric": 2.4368257358712193,
21: (12)                        "Dan": 3.1306328496328168,
22: (12)                        "Franck": 2.3876142275231915,
23: (8)                     }
24: (8)                     G1 = nx.Graph(
25: (12)                        [
26: (16)                            ("Franck", "Aric"),
27: (16)                            ("Aric", "Dan"),
28: (16)                            ("Dan", "Albert"),
29: (16)                            ("Albert", "Franck"),
30: (16)                            ("Dan", "1"),
```

```
(16)                            ("Franck", "Albert"),
(12)                        ]
(8)                     )
(8)                     result1 = subgraph_centrality(G1)
(8)                     for k, v in result1.items():
(12)                        assert answer1[k] == pytest.approx(v, abs=1e-7)
(8)                     result1 = subgraph_centrality_exp(G1)
(8)                     for k, v in result1.items():
(12)                        assert answer1[k] == pytest.approx(v, abs=1e-7)
(4)                 def test_subgraph_centrality_big_graph(self):
(8)                     g199 = nx.complete_graph(199)
(8)                     g200 = nx.complete_graph(200)
(8)                     comm199 = nx.subgraph_centrality(g199)
(8)                     comm199_exp = nx.subgraph_centrality_exp(g199)
(8)                     comm200 = nx.subgraph_centrality(g200)
(8)                     comm200_exp = nx.subgraph_centrality_exp(g200)
(4)                 def test_communicability_betweenness_centrality_small(self):
(8)                     result = communicability_betweenness_centrality(nx.path_graph(2))
(8)                     assert result == {0: 0, 1: 0}
(8)                     result = communicability_betweenness_centrality(nx.path_graph(1))
(8)                     assert result == {0: 0}
(8)                     result = communicability_betweenness_centrality(nx.path_graph(0))
(8)                     assert result == {}
(8)                     answer = {0: 0.1411224421177313, 1: 1.0, 2: 0.1411224421177313}
(8)                     result = communicability_betweenness_centrality(nx.path_graph(3))
(8)                     for k, v in result.items():
(12)                        assert answer[k] == pytest.approx(v, abs=1e-7)
(8)                     result = communicability_betweenness_centrality(nx.complete_graph(3))
(8)                     for k, v in result.items():
(12)                        assert 0.49786143366223296 == pytest.approx(v, abs=1e-7)
(4)                 def test_communicability_betweenness_centrality(self):
(8)                     answer = {
(12)                        0: 0.07017447951484615,
(12)                        1: 0.71565598701107991,
(12)                        2: 0.71565598701107991,
(12)                        3: 0.07017447951484615,
(8)                     }
(8)                     result = communicability_betweenness_centrality(nx.path_graph(4))
(8)                     for k, v in result.items():
(12)                        assert answer[k] == pytest.approx(v, abs=1e-7)
(8)                     answer1 = {
(12)                        "1": 0.060039074193949521,
(12)                        "Albert": 0.315470761661372,
(12)                        "Aric": 0.31547076166137211,
(12)                        "Dan": 0.68297778678316201,
(12)                        "Franck": 0.21977926617449497,
(8)                     }
(8)                     G1 = nx.Graph(
(12)                        [
(16)                            ("Franck", "Aric"),
(16)                            ("Aric", "Dan"),
(16)                            ("Dan", "Albert"),
(16)                            ("Albert", "Franck"),
(16)                            ("Dan", "1"),
(16)                            ("Franck", "Albert"),
(12)                        ]
(8)                     )
(8)                     result1 = communicability_betweenness_centrality(G1)
(8)                     for k, v in result1.items():
(12)                        assert answer1[k] == pytest.approx(v, abs=1e-7)
(4)                 def test_estrada_index(self):
(8)                     answer = 1041.2470334195475
(8)                     result = estrada_index(nx.karate_club_graph())
(8)                     assert answer == pytest.approx(result, abs=1e-7)

----------------------------------------

File 171 - test_second_order_centrality.py:
```

```
1: (0)              """
2: (0)              Tests for second order centrality.
3: (0)              """
4: (0)              import pytest
5: (0)              pytest.importorskip("numpy")
6: (0)              pytest.importorskip("scipy")
7: (0)              import networkx as nx
8: (0)              def test_empty():
9: (4)                  with pytest.raises(nx.NetworkXException):
10: (8)                     G = nx.empty_graph()
11: (8)                     nx.second_order_centrality(G)
12: (0)             def test_non_connected():
13: (4)                 with pytest.raises(nx.NetworkXException):
14: (8)                     G = nx.Graph()
15: (8)                     G.add_node(0)
16: (8)                     G.add_node(1)
17: (8)                     nx.second_order_centrality(G)
18: (0)             def test_non_negative_edge_weights():
19: (4)                 with pytest.raises(nx.NetworkXException):
20: (8)                     G = nx.path_graph(2)
21: (8)                     G.add_edge(0, 1, weight=-1)
22: (8)                     nx.second_order_centrality(G)
23: (0)             def test_weight_attribute():
24: (4)                 G = nx.Graph()
25: (4)                 G.add_weighted_edges_from([(0, 1, 1.0), (1, 2, 3.5)], weight="w")
26: (4)                 expected = {0: 3.431, 1: 3.082, 2: 5.612}
27: (4)                 b = nx.second_order_centrality(G, weight="w")
28: (4)                 for n in sorted(G):
29: (8)                     assert b[n] == pytest.approx(expected[n], abs=1e-2)
30: (0)             def test_one_node_graph():
31: (4)                 """Second order centrality: single node"""
32: (4)                 G = nx.Graph()
33: (4)                 G.add_node(0)
34: (4)                 G.add_edge(0, 0)
35: (4)                 assert nx.second_order_centrality(G)[0] == 0
36: (0)             def test_P3():
37: (4)                 """Second order centrality: line graph, as defined in paper"""
38: (4)                 G = nx.path_graph(3)
39: (4)                 b_answer = {0: 3.741, 1: 1.414, 2: 3.741}
40: (4)                 b = nx.second_order_centrality(G)
41: (4)                 for n in sorted(G):
42: (8)                     assert b[n] == pytest.approx(b_answer[n], abs=1e-2)
43: (0)             def test_K3():
44: (4)                 """Second order centrality: complete graph, as defined in paper"""
45: (4)                 G = nx.complete_graph(3)
46: (4)                 b_answer = {0: 1.414, 1: 1.414, 2: 1.414}
47: (4)                 b = nx.second_order_centrality(G)
48: (4)                 for n in sorted(G):
49: (8)                     assert b[n] == pytest.approx(b_answer[n], abs=1e-2)
50: (0)             def test_ring_graph():
51: (4)                 """Second order centrality: ring graph, as defined in paper"""
52: (4)                 G = nx.cycle_graph(5)
53: (4)                 b_answer = {0: 4.472, 1: 4.472, 2: 4.472, 3: 4.472, 4: 4.472}
54: (4)                 b = nx.second_order_centrality(G)
55: (4)                 for n in sorted(G):
56: (8)                     assert b[n] == pytest.approx(b_answer[n], abs=1e-2)
```

File 172 - test_trophic.py:

```
1: (0)              """Test trophic levels, trophic differences and trophic coherence
2: (0)              """
3: (0)              import pytest
4: (0)              np = pytest.importorskip("numpy")
5: (0)              pytest.importorskip("scipy")
6: (0)              import networkx as nx
7: (0)              def test_trophic_levels():
8: (4)                  """Trivial example"""
```

```
 9: (4)                 G = nx.DiGraph()
10: (4)                 G.add_edge("a", "b")
11: (4)                 G.add_edge("b", "c")
12: (4)                 d = nx.trophic_levels(G)
13: (4)                 assert d == {"a": 1, "b": 2, "c": 3}
14: (0)             def test_trophic_levels_levine():
15: (4)                 """Example from Figure 5 in Stephen Levine (1980) J. theor. Biol. 83,
16: (4)                 195-207
17: (4)                 """
18: (4)                 S = nx.DiGraph()
19: (4)                 S.add_edge(1, 2, weight=1.0)
20: (4)                 S.add_edge(1, 3, weight=0.2)
21: (4)                 S.add_edge(1, 4, weight=0.8)
22: (4)                 S.add_edge(2, 3, weight=0.2)
23: (4)                 S.add_edge(2, 5, weight=0.3)
24: (4)                 S.add_edge(4, 3, weight=0.6)
25: (4)                 S.add_edge(4, 5, weight=0.7)
26: (4)                 S.add_edge(5, 4, weight=0.2)
27: (4)                 S2 = S.copy()
28: (4)                 z = [nid for nid, d in S.in_degree if d == 0]
29: (4)                 for nid in z:
30: (8)                     S.remove_node(nid)
31: (4)                 q = nx.linalg.graphmatrix.adjacency_matrix(S).T
32: (4)                 expected_q = np.array([
33: (8)                     [0, 0, 0., 0],
34: (8)                     [0.2, 0, 0.6, 0],
35: (8)                     [0, 0, 0, 0.2],
36: (8)                     [0.3, 0, 0.7, 0]
37: (4)                 ])
38: (4)                 assert np.array_equal(q.todense(), expected_q)
39: (4)                 assert len(q.shape) == 2
40: (4)                 assert q.shape[0] == q.shape[1]
41: (4)                 assert q.shape[0] == len(S)
42: (4)                 nn = q.shape[0]
43: (4)                 i = np.eye(nn)
44: (4)                 n = np.linalg.inv(i - q)
45: (4)                 y = np.asarray(n) @ np.ones(nn)
46: (4)                 expected_y = np.array([1, 2.07906977, 1.46511628, 2.3255814])
47: (4)                 assert np.allclose(y, expected_y)
48: (4)                 expected_d = {1: 1, 2: 2, 3: 3.07906977, 4: 2.46511628, 5: 3.3255814}
49: (4)                 d = nx.trophic_levels(S2)
50: (4)                 for nid, level in d.items():
51: (8)                     expected_level = expected_d[nid]
52: (8)                     assert expected_level == pytest.approx(level, abs=1e-7)
53: (0)             def test_trophic_levels_simple():
54: (4)                 matrix_a = np.array([[0, 0], [1, 0]])
55: (4)                 G = nx.from_numpy_array(matrix_a, create_using=nx.DiGraph)
56: (4)                 d = nx.trophic_levels(G)
57: (4)                 assert d[0] == pytest.approx(2, abs=1e-7)
58: (4)                 assert d[1] == pytest.approx(1, abs=1e-7)
59: (0)             def test_trophic_levels_more_complex():
60: (4)                 matrix = np.array([
61: (8)                     [0, 1, 0, 0],
62: (8)                     [0, 0, 1, 0],
63: (8)                     [0, 0, 0, 1],
64: (8)                     [0, 0, 0, 0]
65: (4)                 ])
66: (4)                 G = nx.from_numpy_array(matrix, create_using=nx.DiGraph)
67: (4)                 d = nx.trophic_levels(G)
68: (4)                 expected_result = [1, 2, 3, 4]
69: (4)                 for ind in range(4):
70: (8)                     assert d[ind] == pytest.approx(expected_result[ind], abs=1e-7)
71: (4)                 matrix = np.array([
72: (8)                     [0, 1, 1, 0],
73: (8)                     [0, 0, 1, 1],
74: (8)                     [0, 0, 0, 1],
75: (8)                     [0, 0, 0, 0]
76: (4)                 ])
77: (4)                 G = nx.from_numpy_array(matrix, create_using=nx.DiGraph)
```

```
(4)                 d = nx.trophic_levels(G)
(4)                 expected_result = [1, 2, 2.5, 3.25]
(4)                 print("Calculated result: ", d)
(4)                 print("Expected Result: ", expected_result)
(4)                 for ind in range(4):
(8)                     assert d[ind] == pytest.approx(expected_result[ind], abs=1e-7)
(0)             def test_trophic_levels_even_more_complex():
(4)                 matrix = np.array([
(8)                     [0, 0, 0, 0, 0],
(8)                     [0, 1, 0, 1, 0],
(8)                     [1, 0, 0, 0, 0],
(8)                     [0, 1, 0, 0, 0],
(8)                     [0, 0, 0, 1, 0]
(4)                 ])
(4)                 K = np.array([
(8)                     [1, 0, -1, 0, 0],
(8)                     [0, 0.5, 0, -0.5, 0],
(8)                     [0, 0, 1, 0, 0],
(8)                     [0, -0.5, 0, 1, -0.5],
(8)                     [0, 0, 0, 0, 1],
(4)                 ])
(4)                 result_1 = np.ravel(np.linalg.inv(K) @ np.ones(5))
(4)                 G = nx.from_numpy_array(matrix, create_using=nx.DiGraph)
(4)                 result_2 = nx.trophic_levels(G)
(4)                 for ind in range(5):
(8)                     assert result_1[ind] == pytest.approx(result_2[ind], abs=1e-7)
(0)             def test_trophic_levels_singular_matrix():
(4)                 """Should raise an error with graphs with only non-basal nodes"""
(4)                 matrix = np.identity(4)
(4)                 G = nx.from_numpy_array(matrix, create_using=nx.DiGraph)
(4)                 with pytest.raises(nx.NetworkXError) as e:
(8)                     nx.trophic_levels(G)
(4)                 msg = (
(8)                     "Trophic levels are only defined for graphs where every node "
(8)                     + "has a path from a basal node (basal nodes are nodes with no "
(8)                     + "incoming edges)."
(4)                 )
(4)                 assert msg in str(e.value)
(0)             def test_trophic_levels_singular_with_basal():
(4)                 """Should fail to compute if there are any parts of the graph which are
not
(4)                 reachable from any basal node (with in-degree zero).
(4)                 """
(4)                 G = nx.DiGraph()
(4)                 G.add_edge("a", "b")
(4)                 G.add_edge("c", "b")
(4)                 G.add_edge("d", "b")
(4)                 G.add_edge("c", "d")
(4)                 G.add_edge("d", "c")
(4)                 with pytest.raises(nx.NetworkXError) as e:
(8)                     nx.trophic_levels(G)
(4)                 msg = (
(8)                     "Trophic levels are only defined for graphs where every node "
(8)                     + "has a path from a basal node (basal nodes are nodes with no "
(8)                     + "incoming edges)."
(4)                 )
(4)                 assert msg in str(e.value)
(4)                 G = nx.DiGraph()
(4)                 G.add_edge("a", "b")  # a has in-degree zero
(4)                 G.add_edge("c", "b")  # b is one level above a and c
(4)                 G.add_edge("c", "c")  # c has a self-loop
(4)                 with pytest.raises(nx.NetworkXError) as e:
(8)                     nx.trophic_levels(G)
(4)                 msg = (
(8)                     "Trophic levels are only defined for graphs where every node "
(8)                     + "has a path from a basal node (basal nodes are nodes with no "
(8)                     + "incoming edges)."
(4)                 )
(4)                 assert msg in str(e.value)
```

```
(0)  def test_trophic_differences():
(4)      matrix_a = np.array([[0, 1], [0, 0]])
(4)      G = nx.from_numpy_array(matrix_a, create_using=nx.DiGraph)
(4)      diffs = nx.trophic_differences(G)
(4)      assert diffs[(0, 1)] == pytest.approx(1, abs=1e-7)
(4)      matrix_b = np.array([
(8)          [0, 1, 1, 0],
(8)          [0, 0, 1, 1],
(8)          [0, 0, 0, 1],
(8)          [0, 0, 0, 0]
(4)      ])
(4)      G = nx.from_numpy_array(matrix_b, create_using=nx.DiGraph)
(4)      diffs = nx.trophic_differences(G)
(4)      assert diffs[(0, 1)] == pytest.approx(1, abs=1e-7)
(4)      assert diffs[(0, 2)] == pytest.approx(1.5, abs=1e-7)
(4)      assert diffs[(1, 2)] == pytest.approx(0.5, abs=1e-7)
(4)      assert diffs[(1, 3)] == pytest.approx(1.25, abs=1e-7)
(4)      assert diffs[(2, 3)] == pytest.approx(0.75, abs=1e-7)
(0)  def test_trophic_incoherence_parameter_no_cannibalism():
(4)      matrix_a = np.array([[0, 1], [0, 0]])
(4)      G = nx.from_numpy_array(matrix_a, create_using=nx.DiGraph)
(4)      q = nx.trophic_incoherence_parameter(G, cannibalism=False)
(4)      assert q == pytest.approx(0, abs=1e-7)
(4)      matrix_b = np.array([
(8)          [0, 1, 1, 0],
(8)          [0, 0, 1, 1],
(8)          [0, 0, 0, 1],
(8)          [0, 0, 0, 0]
(4)      ])
(4)      G = nx.from_numpy_array(matrix_b, create_using=nx.DiGraph)
(4)      q = nx.trophic_incoherence_parameter(G, cannibalism=False)
(4)      assert q == pytest.approx(np.std([1, 1.5, 0.5, 0.75, 1.25]), abs=1e-7)
(4)      matrix_c = np.array([
(8)          [0, 1, 1, 0],
(8)          [0, 1, 1, 1],
(8)          [0, 0, 0, 1],
(8)          [0, 0, 0, 1]
(4)      ])
(4)      G = nx.from_numpy_array(matrix_c, create_using=nx.DiGraph)
(4)      q = nx.trophic_incoherence_parameter(G, cannibalism=False)
(4)      assert q == pytest.approx(np.std([1, 1.5, 0.5, 0.75, 1.25]), abs=1e-7)
(4)      matrix_d = np.array([
(8)          [0, 1, 1, 0],
(8)          [0, 0, 1, 1],
(8)          [0, 0, 0, 1],
(8)          [0, 0, 0, 0]
(4)      ])
(4)      G = nx.from_numpy_array(matrix_d, create_using=nx.DiGraph)
(4)      q = nx.trophic_incoherence_parameter(G, cannibalism=False)
(4)      assert q == pytest.approx(np.std([1, 1.5, 0.5, 0.75, 1.25]), abs=1e-7)
(0)  def test_trophic_incoherence_parameter_cannibalism():
(4)      matrix_a = np.array([[0, 1], [0, 0]])
(4)      G = nx.from_numpy_array(matrix_a, create_using=nx.DiGraph)
(4)      q = nx.trophic_incoherence_parameter(G, cannibalism=True)
(4)      assert q == pytest.approx(0, abs=1e-7)
(4)      matrix_b = np.array([
(8)          [0, 0, 0, 0, 0],
(8)          [0, 1, 0, 1, 0],
(8)          [1, 0, 0, 0, 0],
(8)          [0, 1, 0, 0, 0],
(8)          [0, 0, 0, 1, 0]
(4)      ])
(4)      G = nx.from_numpy_array(matrix_b, create_using=nx.DiGraph)
(4)      q = nx.trophic_incoherence_parameter(G, cannibalism=True)
(4)      assert q == pytest.approx(2, abs=1e-7)
(4)      matrix_c = np.array([
(8)          [0, 1, 1, 0],
(8)          [0, 0, 1, 1],
(8)          [0, 0, 0, 1],
```

```
215: (8)                    [0, 0, 0, 0]
216: (4)                ])
217: (4)                G = nx.from_numpy_array(matrix_c, create_using=nx.DiGraph)
218: (4)                q = nx.trophic_incoherence_parameter(G, cannibalism=True)
219: (4)                assert q == pytest.approx(np.std([1, 1.5, 0.5, 0.75, 1.25]), abs=1e-7)
```

File 173 - test_voterank.py:

```
1: (0)              """
2: (4)                  Unit tests for VoteRank.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              class TestVoteRankCentrality:
6: (4)                  def test_voterank_centrality_1(self):
7: (8)                      G = nx.Graph()
8: (8)                      G.add_edges_from(
9: (12)                         [
10: (16)                            (7, 8),
11: (16)                            (7, 5),
12: (16)                            (7, 9),
13: (16)                            (5, 0),
14: (16)                            (0, 1),
15: (16)                            (0, 2),
16: (16)                            (0, 3),
17: (16)                            (0, 4),
18: (16)                            (1, 6),
19: (16)                            (2, 6),
20: (16)                            (3, 6),
21: (16)                            (4, 6),
22: (12)                        ]
23: (8)                     )
24: (8)                     assert [0, 7, 6] == nx.voterank(G)
25: (4)                 def test_voterank_emptygraph(self):
26: (8)                     G = nx.Graph()
27: (8)                     assert [] == nx.voterank(G)
28: (4)                 def test_voterank_centrality_2(self):
29: (8)                     G = nx.florentine_families_graph()
30: (8)                     d = nx.voterank(G, 4)
31: (8)                     exact = ["Medici", "Strozzi", "Guadagni", "Castellani"]
32: (8)                     assert exact == d
33: (4)                 def test_voterank_centrality_3(self):
34: (8)                     G = nx.gnc_graph(10, seed=7)
35: (8)                     d = nx.voterank(G, 4)
36: (8)                     exact = [3, 6, 8]
37: (8)                     assert exact == d
38: (4)                 def test_voterank_centrality_4(self):
39: (8)                     G = nx.MultiGraph()
40: (8)                     G.add_edges_from(
41: (12)                        [(0, 1), (0, 1), (1, 2), (2, 5), (2, 5), (5, 6), (5, 6), (2, 4),
(4, 3)]
42: (8)                     )
43: (8)                     exact = [2, 1, 5, 4]
44: (8)                     assert exact == nx.voterank(G)
45: (4)                 def test_voterank_centrality_5(self):
46: (8)                     G = nx.MultiDiGraph()
47: (8)                     G.add_edges_from(
48: (12)                        [(0, 1), (0, 1), (1, 2), (2, 5), (2, 5), (5, 6), (5, 6), (2, 4),
(4, 3)]
49: (8)                     )
50: (8)                     exact = [2, 0, 5, 4]
51: (8)                     assert exact == nx.voterank(G)
```

File 174 - __init__.py:

```
1: (0)              from networkx.algorithms.coloring.greedy_coloring import *
```

```
2: (0)              from networkx.algorithms.coloring.equitable_coloring import equitable_color
3: (0)              __all__ = ["greedy_color", "equitable_color"]
```

File 175 - equitable_coloring.py:

```
1: (0)              """
2: (0)              Equitable coloring of graphs with bounded degree.
3: (0)              """
4: (0)              from collections import defaultdict
5: (0)              import networkx as nx
6: (0)              __all__ = ["equitable_color"]
7: (0)              @nx._dispatchable
8: (0)              def is_coloring(G, coloring):
9: (4)                  """Determine if the coloring is a valid coloring for the graph G."""
10: (4)                 return all(coloring[s] != coloring[d] for s, d in G.edges)
11: (0)             @nx._dispatchable
12: (0)             def is_equitable(G, coloring, num_colors=None):
13: (4)                 """Determines if the coloring is valid and equitable for the graph G."""
14: (4)                 if not is_coloring(G, coloring):
15: (8)                     return False
16: (4)                 color_set_size = defaultdict(int)
17: (4)                 for color in coloring.values():
18: (8)                     color_set_size[color] += 1
19: (4)                 if num_colors is not None:
20: (8)                     for color in range(num_colors):
21: (12)                        if color not in color_set_size:
22: (16)                            color_set_size[color] = 0
23: (4)                 all_set_sizes = set(color_set_size.values())
24: (4)                 if len(all_set_sizes) == 0 and num_colors is None:  # Was an empty graph
25: (8)                     return True
26: (4)                 elif len(all_set_sizes) == 1:
27: (8)                     return True
28: (4)                 elif len(all_set_sizes) == 2:
29: (8)                     a, b = list(all_set_sizes)
30: (8)                     return abs(a - b) <= 1
31: (4)                 else:  # len(all_set_sizes) > 2:
32: (8)                     return False
33: (0)             def make_C_from_F(F):
34: (4)                 C = defaultdict(list)
35: (4)                 for node, color in F.items():
36: (8)                     C[color].append(node)
37: (4)                 return C
38: (0)             def make_N_from_L_C(L, C):
39: (4)                 nodes = L.keys()
40: (4)                 colors = C.keys()
41: (4)                 return {
42: (8)                     (node, color): sum(1 for v in L[node] if v in C[color])
43: (8)                     for node in nodes
44: (8)                     for color in colors
45: (4)                 }
46: (0)             def make_H_from_C_N(C, N):
47: (4)                 return {
48: (8)                     (c1, c2): sum(1 for node in C[c1] if N[(node, c2)] == 0) for c1 in C
for c2 in C
49: (4)                 }
50: (0)             def change_color(u, X, Y, N, H, F, C, L):
51: (4)                 """Change the color of 'u' from X to Y and update N, H, F, C."""
52: (4)                 assert F[u] == X and X != Y
53: (4)                 F[u] = Y
54: (4)                 for k in C:
55: (8)                     if N[u, k] == 0:
56: (12)                        H[(X, k)] -= 1
57: (12)                        H[(Y, k)] += 1
58: (4)                 for v in L[u]:
59: (8)                     N[(v, X)] -= 1
60: (8)                     N[(v, Y)] += 1
61: (8)                     if N[(v, X)] == 0:
```

```
62: (12)                        H[(F[v], X)] += 1
63: (8)                     if N[(v, Y)] == 1:
64: (12)                        H[(F[v], Y)] -= 1
65: (4)                 C[X].remove(u)
66: (4)                 C[Y].append(u)
67: (0)             def move_witnesses(src_color, dst_color, N, H, F, C, T_cal, L):
68: (4)                 """Move witness along a path from src_color to dst_color."""
69: (4)                 X = src_color
70: (4)                 while X != dst_color:
71: (8)                     Y = T_cal[X]
72: (8)                     w = next(x for x in C[X] if N[(x, Y)] == 0)
73: (8)                     change_color(w, X, Y, N=N, H=H, F=F, C=C, L=L)
74: (8)                     X = Y
75: (0)             @nx._dispatchable(mutates_input=True)
76: (0)             def pad_graph(G, num_colors):
77: (4)                 """Add a disconnected complete clique K_p such that the number of nodes in
78: (4)                 the graph becomes a multiple of `num_colors`.
79: (4)                 Assumes that the graph's nodes are labelled using integers.
80: (4)                 Returns the number of nodes with each color.
81: (4)                 """
82: (4)                 n_ = len(G)
83: (4)                 r = num_colors - 1
84: (4)                 s = n_ // (r + 1)
85: (4)                 if n_ != s * (r + 1):
86: (8)                     p = (r + 1) - n_ % (r + 1)
87: (8)                     s += 1
88: (8)                     K = nx.relabel_nodes(nx.complete_graph(p), {idx: idx + n_ for idx in
range(p)})
89: (8)                     G.add_edges_from(K.edges)
90: (4)                 return s
91: (0)             def procedure_P(V_minus, V_plus, N, H, F, C, L, excluded_colors=None):
92: (4)                 """Procedure P as described in the paper."""
93: (4)                 if excluded_colors is None:
94: (8)                     excluded_colors = set()
95: (4)                 A_cal = set()
96: (4)                 T_cal = {}
97: (4)                 R_cal = []
98: (4)                 reachable = [V_minus]
99: (4)                 marked = set(reachable)
100: (4)                idx = 0
101: (4)                while idx < len(reachable):
102: (8)                    pop = reachable[idx]
103: (8)                    idx += 1
104: (8)                    A_cal.add(pop)
105: (8)                    R_cal.append(pop)
106: (8)                    next_layer = []
107: (8)                    for k in C:
108: (12)                       if (
109: (16)                           H[(k, pop)] > 0
110: (16)                           and k not in A_cal
111: (16)                           and k not in excluded_colors
112: (16)                           and k not in marked
113: (12)                       ):
114: (16)                           next_layer.append(k)
115: (8)                    for dst in next_layer:
116: (12)                       T_cal[dst] = pop
117: (8)                    marked.update(next_layer)
118: (8)                    reachable.extend(next_layer)
119: (4)                b = len(C) - len(A_cal)
120: (4)                if V_plus in A_cal:
121: (8)                    move_witnesses(V_plus, V_minus, N=N, H=H, F=F, C=C, T_cal=T_cal, L=L)
122: (4)                else:
123: (8)                    A_0 = set()
124: (8)                    A_cal_0 = set()
125: (8)                    num_terminal_sets_found = 0
126: (8)                    made_equitable = False
127: (8)                    for W_1 in R_cal[::-1]:
128: (12)                       for v in C[W_1]:
129: (16)                           X = None
```

```
130: (16)                            for U in C:
131: (20)                                if N[(v, U)] == 0 and U in A_cal and U != W_1:
132: (24)                                    X = U
133: (16)                            if X is None:
134: (20)                                continue
135: (16)                            for U in C:
136: (20)                                if N[(v, U)] >= 1 and U not in A_cal:
137: (24)                                    X_prime = U
138: (24)                                    w = v
139: (24)                                    try:
140: (28)                                        y = next(
141: (32)                                            node
142: (32)                                            for node in L[w]
143: (32)                                            if F[node] == X_prime and N[(node, W_1)] == 1
144: (28)                                        )
145: (24)                                    except StopIteration:
146: (28)                                        pass
147: (24)                                    else:
148: (28)                                        W = W_1
149: (28)                                        change_color(w, W, X, N=N, H=H, F=F, C=C, L=L)
150: (28)                                        move_witnesses(
151: (32)                                            src_color=X,
152: (32)                                            dst_color=V_minus,
153: (32)                                            N=N,
154: (32)                                            H=H,
155: (32)                                            F=F,
156: (32)                                            C=C,
157: (32)                                            T_cal=T_cal,
158: (32)                                            L=L,
159: (28)                                        )
160: (28)                                        change_color(y, X_prime, W, N=N, H=H, F=F, C=C,
L=L)
161: (28)                                        procedure_P(
162: (32)                                            V_minus=X_prime,
163: (32)                                            V_plus=V_plus,
164: (32)                                            N=N,
165: (32)                                            H=H,
166: (32)                                            C=C,
167: (32)                                            F=F,
168: (32)                                            L=L,
169: (32)                                            excluded_colors=excluded_colors.union(A_cal),
170: (28)                                        )
171: (28)                                        made_equitable = True
172: (28)                                        break
173: (16)                            if made_equitable:
174: (20)                                break
175: (12)                        else:
176: (16)                            A_cal_0.add(W_1)
177: (16)                            A_0.update(C[W_1])
178: (16)                            num_terminal_sets_found += 1
179: (12)                        if num_terminal_sets_found == b:
180: (16)                            B_cal_prime = set()
181: (16)                            T_cal_prime = {}
182: (16)                            reachable = [V_plus]
183: (16)                            marked = set(reachable)
184: (16)                            idx = 0
185: (16)                            while idx < len(reachable):
186: (20)                                pop = reachable[idx]
187: (20)                                idx += 1
188: (20)                                B_cal_prime.add(pop)
189: (20)                                next_layer = [
190: (24)                                    k
191: (24)                                    for k in C
192: (24)                                    if H[(pop, k)] > 0 and k not in B_cal_prime and k not
in marked
193: (20)                                ]
194: (20)                                for dst in next_layer:
195: (24)                                    T_cal_prime[pop] = dst
196: (20)                                marked.update(next_layer)
```

```
197: (20)                                reachable.extend(next_layer)
198: (16)                            I_set = set()
199: (16)                            I_covered = set()
200: (16)                            W_covering = {}
201: (16)                            B_prime = [node for k in B_cal_prime for node in C[k]]
202: (16)                            for z in C[V_plus] + B_prime:
203: (20)                                if z in I_covered or F[z] not in B_cal_prime:
204: (24)                                    continue
205: (20)                                I_set.add(z)
206: (20)                                I_covered.add(z)
207: (20)                                I_covered.update(list(L[z]))
208: (20)                                for w in L[z]:
209: (24)                                    if F[w] in A_cal_0 and N[(z, F[w])] == 1:
210: (28)                                        if w not in W_covering:
211: (32)                                            W_covering[w] = z
212: (28)                                        else:
213: (32)                                            z_1 = W_covering[w]
214: (32)                                            Z = F[z_1]
215: (32)                                            W = F[w]
216: (32)                                            move_witnesses(
217: (36)                                                W, V_minus, N=N, H=H, F=F, C=C,
T_cal=T_cal, L=L
218: (32)                                            )
219: (32)                                            move_witnesses(
220: (36)                                                V_plus,
221: (36)                                                Z,
222: (36)                                                N=N,
223: (36)                                                H=H,
224: (36)                                                F=F,
225: (36)                                                C=C,
226: (36)                                                T_cal=T_cal_prime,
227: (36)                                                L=L,
228: (32)                                            )
229: (32)                                            change_color(z_1, Z, W, N=N, H=H, F=F, C=C,
L=L)
230: (32)                                            W_plus = next(
231: (36)                                                k for k in C if N[(w, k)] == 0 and k not
in A_cal
232: (32)                                            )
233: (32)                                            change_color(w, W, W_plus, N=N, H=H, F=F, C=C,
L=L)
234: (32)                                            excluded_colors.update(
235: (36)                                                [k for k in C if k != W and k not in
B_cal_prime]
236: (32)                                            )
237: (32)                                            procedure_P(
238: (36)                                                V_minus=W,
239: (36)                                                V_plus=W_plus,
240: (36)                                                N=N,
241: (36)                                                H=H,
242: (36)                                                C=C,
243: (36)                                                F=F,
244: (36)                                                L=L,
245: (36)                                                excluded_colors=excluded_colors,
246: (32)                                            )
247: (32)                                            made_equitable = True
248: (32)                                            break
249: (20)                                if made_equitable:
250: (24)                                    break
251: (16)                            else:
252: (20)                                assert False, (
253: (24)                                    "Must find a w which is the solo neighbor "
254: (24)                                    "of two vertices in B_cal_prime."
255: (20)                                )
256: (12)                        if made_equitable:
257: (16)                            break
258: (0)            @nx._dispatchable
259: (0)            def equitable_color(G, num_colors):
260: (4)                """Provides an equitable coloring for nodes of `G`.
```

```
    Attempts to color a graph using `num_colors` colors, where no neighbors of
    a node can have same color as the node itself and the number of nodes with
    each color differ by at most 1. `num_colors` must be greater than the
    maximum degree of `G`. The algorithm is described in [1]_ and has
    complexity O(num_colors * n**2).
    Parameters
    ----------
    G : networkX graph
       The nodes of this graph will be colored.
    num_colors : number of colors to use
       This number must be at least one more than the maximum degree of nodes
       in the graph.
    Returns
    -------
    A dictionary with keys representing nodes and values representing
    corresponding coloring.
    Examples
    --------
    >>> G = nx.cycle_graph(4)
    >>> nx.coloring.equitable_color(G, num_colors=3)  # doctest: +SKIP
    {0: 2, 1: 1, 2: 2, 3: 0}
    Raises
    ------
    NetworkXAlgorithmError
        If `num_colors` is not at least the maximum degree of the graph `G`
    References
    ----------
    .. [1] Kierstead, H. A., Kostochka, A. V., Mydlarz, M., & Szemerédi, E.
        (2010). A fast algorithm for equitable coloring. Combinatorica, 30(2),
        217-224.
    """
    nodes_to_int = {}
    int_to_nodes = {}
    for idx, node in enumerate(G.nodes):
        nodes_to_int[node] = idx
        int_to_nodes[idx] = node
    G = nx.relabel_nodes(G, nodes_to_int, copy=True)
    if len(G.nodes) > 0:
        r_ = max(G.degree(node) for node in G.nodes)
    else:
        r_ = 0
    if r_ >= num_colors:
        raise nx.NetworkXAlgorithmError(
            f"Graph has maximum degree {r_}, needs "
            f"{r_ + 1} (> {num_colors}) colors for guaranteed coloring."
        )
    pad_graph(G, num_colors)
    L_ = {node: [] for node in G.nodes}
    F = {node: idx % num_colors for idx, node in enumerate(G.nodes)}
    C = make_C_from_F(F)
    N = make_N_from_L_C(L_, C)
    H = make_H_from_C_N(C, N)
    edges_seen = set()
    for u in sorted(G.nodes):
        for v in sorted(G.neighbors(u)):
            if (v, u) in edges_seen:
                continue
            edges_seen.add((u, v))
            L_[u].append(v)
            L_[v].append(u)
            N[(u, F[v])] += 1
            N[(v, F[u])] += 1
            if F[u] != F[v]:
                if N[(u, F[v])] == 1:
                    H[F[u], F[v]] -= 1  # u cannot witness an edge between F[u], F[v]
                if N[(v, F[u])] == 1:
                    H[F[v], F[u]] -= 1  # v cannot witness an edge between F[v], F[u]
```

```
328: (8)                    if N[(u, F[u])] != 0:
329: (12)                       Y = next(k for k in C if N[(u, k)] == 0)
330: (12)                       X = F[u]
331: (12)                       change_color(u, X, Y, N=N, H=H, F=F, C=C, L=L_)
332: (12)                       procedure_P(V_minus=X, V_plus=Y, N=N, H=H, F=F, C=C, L=L_)
333: (4)                return {int_to_nodes[x]: F[x] for x in int_to_nodes}
```

File 176 - __init__.py:

```
1: (0)
```

File 177 - greedy_coloring.py:

```
1: (0)              """
2: (0)              Greedy graph coloring using various strategies.
3: (0)              """
4: (0)              import itertools
5: (0)              from collections import defaultdict, deque
6: (0)              import networkx as nx
7: (0)              from networkx.utils import arbitrary_element, py_random_state
8: (0)              __all__ = [
9: (4)                  "greedy_color",
10: (4)                 "strategy_connected_sequential",
11: (4)                 "strategy_connected_sequential_bfs",
12: (4)                 "strategy_connected_sequential_dfs",
13: (4)                 "strategy_independent_set",
14: (4)                 "strategy_largest_first",
15: (4)                 "strategy_random_sequential",
16: (4)                 "strategy_saturation_largest_first",
17: (4)                 "strategy_smallest_last",
18: (0)             ]
19: (0)             def strategy_largest_first(G, colors):
20: (4)                 """Returns a list of the nodes of ``G`` in decreasing order by
21: (4)                 degree.
22: (4)                 ``G`` is a NetworkX graph. ``colors`` is ignored.
23: (4)                 """
24: (4)                 return sorted(G, key=G.degree, reverse=True)
25: (0)             @py_random_state(2)
26: (0)             def strategy_random_sequential(G, colors, seed=None):
27: (4)                 """Returns a random permutation of the nodes of ``G`` as a list.
28: (4)                 ``G`` is a NetworkX graph. ``colors`` is ignored.
29: (4)                 seed : integer, random_state, or None (default)
30: (8)                     Indicator of random number generation state.
31: (8)                     See :ref:`Randomness<randomness>`.
32: (4)                 """
33: (4)                 nodes = list(G)
34: (4)                 seed.shuffle(nodes)
35: (4)                 return nodes
36: (0)             def strategy_smallest_last(G, colors):
37: (4)                 """Returns a deque of the nodes of ``G``, "smallest" last.
38: (4)                 Specifically, the degrees of each node are tracked in a bucket queue.
39: (4)                 From this, the node of minimum degree is repeatedly popped from the
40: (4)                 graph, updating its neighbors' degrees.
41: (4)                 ``G`` is a NetworkX graph. ``colors`` is ignored.
42: (4)                 This implementation of the strategy runs in $O(n + m)$ time
43: (4)                 (ignoring polylogarithmic factors), where $n$ is the number of nodes
44: (4)                 and $m$ is the number of edges.
45: (4)                 This strategy is related to :func:`strategy_independent_set`: if we
46: (4)                 interpret each node removed as an independent set of size one, then
47: (4)                 this strategy chooses an independent set of size one instead of a
48: (4)                 maximal independent set.
49: (4)                 """
50: (4)                 H = G.copy()
51: (4)                 result = deque()
52: (4)                 degrees = defaultdict(set)  # set(), for fast random-access removals
```

```
53: (4)                 lbound = float("inf")
54: (4)                 for node, d in H.degree():
55: (8)                     degrees[d].add(node)
56: (8)                     lbound = min(lbound, d)  # Lower bound on min-degree.
57: (4)                 def find_min_degree():
58: (8)                     return next(d for d in itertools.count(lbound) if d in degrees)
59: (4)                 for _ in G:
60: (8)                     min_degree = find_min_degree()
61: (8)                     u = degrees[min_degree].pop()
62: (8)                     if not degrees[min_degree]:  # Clean up the degree list.
63: (12)                        del degrees[min_degree]
64: (8)                     result.appendleft(u)
65: (8)                     for v in H[u]:
66: (12)                        degree = H.degree(v)
67: (12)                        degrees[degree].remove(v)
68: (12)                        if not degrees[degree]:  # Clean up the degree list.
69: (16)                            del degrees[degree]
70: (12)                        degrees[degree - 1].add(v)
71: (8)                     H.remove_node(u)
72: (8)                     lbound = min_degree - 1  # Subtract 1 in case of tied neighbors.
73: (4)                 return result
74: (0)             def _maximal_independent_set(G):
75: (4)                 """Returns a maximal independent set of nodes in ``G`` by repeatedly
76: (4)                 choosing an independent node of minimum degree (with respect to the
77: (4)                 subgraph of unchosen nodes).
78: (4)                 """
79: (4)                 result = set()
80: (4)                 remaining = set(G)
81: (4)                 while remaining:
82: (8)                     G = G.subgraph(remaining)
83: (8)                     v = min(remaining, key=G.degree)
84: (8)                     result.add(v)
85: (8)                     remaining -= set(G[v]) | {v}
86: (4)                 return result
87: (0)             def strategy_independent_set(G, colors):
88: (4)                 """Uses a greedy independent set removal strategy to determine the
89: (4)                 colors.
90: (4)                 This function updates ``colors`` **in-place** and return ``None``,
91: (4)                 unlike the other strategy functions in this module.
92: (4)                 This algorithm repeatedly finds and removes a maximal independent
93: (4)                 set, assigning each node in the set an unused color.
94: (4)                 ``G`` is a NetworkX graph.
95: (4)                 This strategy is related to :func:`strategy_smallest_last`: in that
96: (4)                 strategy, an independent set of size one is chosen at each step
97: (4)                 instead of a maximal independent set.
98: (4)                 """
99: (4)                 remaining_nodes = set(G)
100: (4)                while len(remaining_nodes) > 0:
101: (8)                    nodes = _maximal_independent_set(G.subgraph(remaining_nodes))
102: (8)                    remaining_nodes -= nodes
103: (8)                    yield from nodes
104: (0)            def strategy_connected_sequential_bfs(G, colors):
105: (4)                """Returns an iterable over nodes in ``G`` in the order given by a
106: (4)                breadth-first traversal.
107: (4)                The generated sequence has the property that for each node except
108: (4)                the first, at least one neighbor appeared earlier in the sequence.
109: (4)                ``G`` is a NetworkX graph. ``colors`` is ignored.
110: (4)                """
111: (4)                return strategy_connected_sequential(G, colors, "bfs")
112: (0)            def strategy_connected_sequential_dfs(G, colors):
113: (4)                """Returns an iterable over nodes in ``G`` in the order given by a
114: (4)                depth-first traversal.
115: (4)                The generated sequence has the property that for each node except
116: (4)                the first, at least one neighbor appeared earlier in the sequence.
117: (4)                ``G`` is a NetworkX graph. ``colors`` is ignored.
118: (4)                """
119: (4)                return strategy_connected_sequential(G, colors, "dfs")
120: (0)            def strategy_connected_sequential(G, colors, traversal="bfs"):
121: (4)                """Returns an iterable over nodes in ``G`` in the order given by a
```

```
(4)                 breadth-first or depth-first traversal.
(4)                 ``traversal`` must be one of the strings ``'dfs'`` or ``'bfs'``,
(4)                 representing depth-first traversal or breadth-first traversal,
(4)                 respectively.
(4)                 The generated sequence has the property that for each node except
(4)                 the first, at least one neighbor appeared earlier in the sequence.
(4)                 ``G`` is a NetworkX graph. ``colors`` is ignored.
(4)                 """
(4)                 if traversal == "bfs":
(8)                     traverse = nx.bfs_edges
(4)                 elif traversal == "dfs":
(8)                     traverse = nx.dfs_edges
(4)                 else:
(8)                     raise nx.NetworkXError(
(12)                        "Please specify one of the strings 'bfs' or"
(12)                        " 'dfs' for connected sequential ordering"
(8)                     )
(4)                 for component in nx.connected_components(G):
(8)                     source = arbitrary_element(component)
(8)                     yield source
(8)                     for _, end in traverse(G.subgraph(component), source):
(12)                        yield end
(0)             def strategy_saturation_largest_first(G, colors):
(4)                 """Iterates over all the nodes of ``G`` in "saturation order" (also
(4)                 known as "DSATUR").
(4)                 ``G`` is a NetworkX graph. ``colors`` is a dictionary mapping nodes of
(4)                 ``G`` to colors, for those nodes that have already been colored.
(4)                 """
(4)                 distinct_colors = {v: set() for v in G}
(4)                 for node, color in colors.items():
(8)                     for neighbor in G[node]:
(12)                        distinct_colors[neighbor].add(color)
(4)                 if len(colors) >= 2:
(8)                     for node, color in colors.items():
(12)                        if color in distinct_colors[node]:
(16)                            raise nx.NetworkXError("Neighboring nodes must have different
colors")
(4)                 if not colors:
(8)                     node = max(G, key=G.degree)
(8)                     yield node
(8)                     for v in G[node]:
(12)                        distinct_colors[v].add(0)
(4)                 while len(G) != len(colors):
(8)                     for node, color in colors.items():
(12)                        for neighbor in G[node]:
(16)                            distinct_colors[neighbor].add(color)
(8)                     saturation = {v: len(c) for v, c in distinct_colors.items() if v not
in colors}
(8)                     node = max(saturation, key=lambda v: (saturation[v], G.degree(v)))
(8)                     yield node
(0)             STRATEGIES = {
(4)                 "largest_first": strategy_largest_first,
(4)                 "random_sequential": strategy_random_sequential,
(4)                 "smallest_last": strategy_smallest_last,
(4)                 "independent_set": strategy_independent_set,
(4)                 "connected_sequential_bfs": strategy_connected_sequential_bfs,
(4)                 "connected_sequential_dfs": strategy_connected_sequential_dfs,
(4)                 "connected_sequential": strategy_connected_sequential,
(4)                 "saturation_largest_first": strategy_saturation_largest_first,
(4)                 "DSATUR": strategy_saturation_largest_first,
(0)             }
(0)             @nx._dispatchable
(0)             def greedy_color(G, strategy="largest_first", interchange=False):
(4)                 """Color a graph using various strategies of greedy graph coloring.
(4)                 Attempts to color a graph using as few colors as possible, where no
(4)                 neighbors of a node can have same color as the node itself. The
(4)                 given strategy determines the order in which nodes are colored.
(4)                 The strategies are described in [1]_, and smallest-last is based on
(4)                 [2]_.
```

```
    Parameters
    ----------
    G : NetworkX graph
    strategy : string or function(G, colors)
       A function (or a string representing a function) that provides
       the coloring strategy, by returning nodes in the ordering they
       should be colored. ``G`` is the graph, and ``colors`` is a
       dictionary of the currently assigned colors, keyed by nodes. The
       function must return an iterable over all the nodes in ``G``.
       If the strategy function is an iterator generator (that is, a
       function with ``yield`` statements), keep in mind that the
       ``colors`` dictionary will be updated after each ``yield``, since
       this function chooses colors greedily.
       If ``strategy`` is a string, it must be one of the following,
       each of which represents one of the built-in strategy functions.
       * ```'largest_first'```
       * ```'random_sequential'```
       * ```'smallest_last'```
       * ```'independent_set'```
       * ```'connected_sequential_bfs'```
       * ```'connected_sequential_dfs'```
       * ```'connected_sequential'``` (alias for the previous strategy)
       * ```'saturation_largest_first'```
       * ```'DSATUR'``` (alias for the previous strategy)
    interchange: bool
       Will use the color interchange algorithm described by [3]_ if set
       to ``True``.
       Note that ``saturation_largest_first`` and ``independent_set``
       do not work with interchange. Furthermore, if you use
       interchange with your own strategy function, you cannot rely
       on the values in the ``colors`` argument.
    Returns
    -------
    A dictionary with keys representing nodes and values representing
    corresponding coloring.
    Examples
    --------
    >>> G = nx.cycle_graph(4)
    >>> d = nx.coloring.greedy_color(G, strategy="largest_first")
    >>> d in [{0: 0, 1: 1, 2: 0, 3: 1}, {0: 1, 1: 0, 2: 1, 3: 0}]
    True
    Raises
    ------
    NetworkXPointlessConcept
        If ``strategy`` is ``saturation_largest_first`` or
        ``independent_set`` and ``interchange`` is ``True``.
    References
    ----------
    .. [1] Adrian Kosowski, and Krzysztof Manuszewski,
       Classical Coloring of Graphs, Graph Colorings, 2-19, 2004.
       ISBN 0-8218-3458-4.
    .. [2] David W. Matula, and Leland L. Beck, "Smallest-last
       ordering and clustering and graph coloring algorithms." *J. ACM* 30,
       3 (July 1983), 417–427. <https://doi.org/10.1145/2402.322385>
    .. [3] Maciej M. Sysło, Narsingh Deo, Janusz S. Kowalik,
       Discrete Optimization Algorithms with Pascal Programs, 415-424, 1983.
       ISBN 0-486-45353-7.
    """
    if len(G) == 0:
        return {}
    strategy = STRATEGIES.get(strategy, strategy)
    if not callable(strategy):
        raise nx.NetworkXError(
            f"strategy must be callable or a valid string. {strategy} not valid."
        )
    if interchange:
        if strategy is strategy_independent_set:
            msg = "interchange cannot be used with independent_set"
```

```
(12)                        raise nx.NetworkXPointlessConcept(msg)
(8)                     if strategy is strategy_saturation_largest_first:
(12)                        msg = "interchange cannot be used with" "
saturation_largest_first"
(12)                        raise nx.NetworkXPointlessConcept(msg)
(4)                 colors = {}
(4)                 nodes = strategy(G, colors)
(4)                 if interchange:
(8)                     return _greedy_coloring_with_interchange(G, nodes)
(4)                 for u in nodes:
(8)                     nbr_colors = {colors[v] for v in G[u] if v in colors}
(8)                     for color in itertools.count():
(12)                        if color not in nbr_colors:
(16)                            break
(8)                     colors[u] = color
(4)                 return colors
(0)             class _Node:
(4)                 __slots__ = ["node_id", "color", "adj_list", "adj_color"]
(4)                 def __init__(self, node_id, n):
(8)                     self.node_id = node_id
(8)                     self.color = -1
(8)                     self.adj_list = None
(8)                     self.adj_color = [None for _ in range(n)]
(4)                 def __repr__(self):
(8)                     return (
(12)                        f"Node_id: {self.node_id}, Color: {self.color}, "
(12)                        f"Adj_list: ({self.adj_list}), adj_color: ({self.adj_color})"
(8)                     )
(4)                 def assign_color(self, adj_entry, color):
(8)                     adj_entry.col_prev = None
(8)                     adj_entry.col_next = self.adj_color[color]
(8)                     self.adj_color[color] = adj_entry
(8)                     if adj_entry.col_next is not None:
(12)                        adj_entry.col_next.col_prev = adj_entry
(4)                 def clear_color(self, adj_entry, color):
(8)                     if adj_entry.col_prev is None:
(12)                        self.adj_color[color] = adj_entry.col_next
(8)                     else:
(12)                        adj_entry.col_prev.col_next = adj_entry.col_next
(8)                     if adj_entry.col_next is not None:
(12)                        adj_entry.col_next.col_prev = adj_entry.col_prev
(4)                 def iter_neighbors(self):
(8)                     adj_node = self.adj_list
(8)                     while adj_node is not None:
(12)                        yield adj_node
(12)                        adj_node = adj_node.next
(4)                 def iter_neighbors_color(self, color):
(8)                     adj_color_node = self.adj_color[color]
(8)                     while adj_color_node is not None:
(12)                        yield adj_color_node.node_id
(12)                        adj_color_node = adj_color_node.col_next
(0)             class _AdjEntry:
(4)                 __slots__ = ["node_id", "next", "mate", "col_next", "col_prev"]
(4)                 def __init__(self, node_id):
(8)                     self.node_id = node_id
(8)                     self.next = None
(8)                     self.mate = None
(8)                     self.col_next = None
(8)                     self.col_prev = None
(4)                 def __repr__(self):
(8)                     col_next = None if self.col_next is None else self.col_next.node_id
(8)                     col_prev = None if self.col_prev is None else self.col_prev.node_id
(8)                     return (
(12)                        f"Node_id: {self.node_id}, Next: ({self.next}), "
(12)                        f"Mate: ({self.mate.node_id}), "
(12)                        f"col_next: ({col_next}), col_prev: ({col_prev})"
(8)                     )
(0)             def _greedy_coloring_with_interchange(G, nodes):
(4)                 """Return a coloring for `original_graph` using interchange approach
```

```
    This procedure is an adaption of the algorithm described by [1]_,
    and is an implementation of coloring with interchange. Please be
    advised, that the datastructures used are rather complex because
    they are optimized to minimize the time spent identifying
    subcomponents of the graph, which are possible candidates for color
    interchange.
    Parameters
    ----------
    G : NetworkX graph
        The graph to be colored
    nodes : list
        nodes ordered using the strategy of choice
    Returns
    -------
    dict :
        A dictionary keyed by node to a color value
    References
    ----------
    .. [1] Maciej M. Syslo, Narsingh Deo, Janusz S. Kowalik,
       Discrete Optimization Algorithms with Pascal Programs, 415-424, 1983.
       ISBN 0-486-45353-7.
    """
    n = len(G)
    graph = {node: _Node(node, n) for node in G}
    for node1, node2 in G.edges():
        adj_entry1 = _AdjEntry(node2)
        adj_entry2 = _AdjEntry(node1)
        adj_entry1.mate = adj_entry2
        adj_entry2.mate = adj_entry1
        node1_head = graph[node1].adj_list
        adj_entry1.next = node1_head
        graph[node1].adj_list = adj_entry1
        node2_head = graph[node2].adj_list
        adj_entry2.next = node2_head
        graph[node2].adj_list = adj_entry2
    k = 0
    for node in nodes:
        neighbors = graph[node].iter_neighbors()
        col_used = {graph[adj_node.node_id].color for adj_node in neighbors}
        col_used.discard(-1)
        k1 = next(itertools.dropwhile(lambda x: x in col_used, itertools.count()))
        if k1 > k:
            connected = True
            visited = set()
            col1 = -1
            col2 = -1
            while connected and col1 < k:
                col1 += 1
                neighbor_cols = graph[node].iter_neighbors_color(col1)
                col1_adj = list(neighbor_cols)
                col2 = col1
                while connected and col2 < k:
                    col2 += 1
                    visited = set(col1_adj)
                    frontier = list(col1_adj)
                    i = 0
                    while i < len(frontier):
                        search_node = frontier[i]
                        i += 1
                        col_opp = col2 if graph[search_node].color == col1 else col1
                        neighbor_cols = graph[search_node].iter_neighbors_color(col_opp)
                        for neighbor in neighbor_cols:
                            if neighbor not in visited:
                                visited.add(neighbor)
                                frontier.append(neighbor)
                    connected = (
```

```
391: (24)                                   len(
392: (28)
visited.intersection(graph[node].iter_neighbors_color(col2))
393: (24)                                   )
394: (24)                                   > 0
395: (20)                               )
396: (12)                       if not connected:
397: (16)                           for search_node in visited:
398: (20)                               graph[search_node].color = (
399: (24)                                   col2 if graph[search_node].color == col1 else col1
400: (20)                               )
401: (20)                               col2_adj = graph[search_node].adj_color[col2]
402: (20)                               graph[search_node].adj_color[col2] =
graph[search_node].adj_color[
403: (24)                                   col1
404: (20)                               ]
405: (20)                               graph[search_node].adj_color[col1] = col2_adj
406: (16)                           for search_node in visited:
407: (20)                               col = graph[search_node].color
408: (20)                               col_opp = col1 if col == col2 else col2
409: (20)                               for adj_node in graph[search_node].iter_neighbors():
410: (24)                                   if graph[adj_node.node_id].color != col_opp:
411: (28)                                       adj_mate = adj_node.mate
412: (28)                                       graph[adj_node.node_id].clear_color(adj_mate,
col_opp)
413: (28)                                       graph[adj_node.node_id].assign_color(adj_mate,
col)
414: (16)                           k1 = col1
415: (8)                    graph[node].color = k1
416: (8)                    k = max(k1, k)
417: (8)                    for adj_node in graph[node].iter_neighbors():
418: (12)                       adj_mate = adj_node.mate
419: (12)                       graph[adj_node.node_id].assign_color(adj_mate, k1)
420: (4)                return {node.node_id: node.color for node in graph.values()}
```

File 178 - test_coloring.py:

```
1: (0)              """Greedy coloring test suite.
2: (0)              """
3: (0)              import itertools
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              is_coloring = nx.algorithms.coloring.equitable_coloring.is_coloring
7: (0)              is_equitable = nx.algorithms.coloring.equitable_coloring.is_equitable
8: (0)              ALL_STRATEGIES = [
9: (4)                  "largest_first",
10: (4)                 "random_sequential",
11: (4)                 "smallest_last",
12: (4)                 "independent_set",
13: (4)                 "connected_sequential_bfs",
14: (4)                 "connected_sequential_dfs",
15: (4)                 "connected_sequential",
16: (4)                 "saturation_largest_first",
17: (4)                 "DSATUR",
18: (0)             ]
19: (0)             INTERCHANGE_INVALID = ["independent_set", "saturation_largest_first",
"DSATUR"]
20: (0)             class TestColoring:
21: (4)                 def test_basic_cases(self):
22: (8)                     def check_basic_case(graph_func, n_nodes, strategy, interchange):
23: (12)                        graph = graph_func()
24: (12)                        coloring = nx.coloring.greedy_color(
25: (16)                            graph, strategy=strategy, interchange=interchange
26: (12)                        )
27: (12)                        assert verify_length(coloring, n_nodes)
28: (12)                        assert verify_coloring(graph, coloring)
29: (8)                     for graph_func, n_nodes in BASIC_TEST_CASES.items():
```

```
30: (12)                        for interchange in [True, False]:
31: (16)                            for strategy in ALL_STRATEGIES:
32: (20)                                check_basic_case(graph_func, n_nodes, strategy, False)
33: (20)                                if strategy not in INTERCHANGE_INVALID:
34: (24)                                    check_basic_case(graph_func, n_nodes, strategy, True)
35: (4)                 def test_special_cases(self):
36: (8)                     def check_special_case(strategy, graph_func, interchange, colors):
37: (12)                        graph = graph_func()
38: (12)                        coloring = nx.coloring.greedy_color(
39: (16)                            graph, strategy=strategy, interchange=interchange
40: (12)                        )
41: (12)                        if not hasattr(colors, "__len__"):
42: (16)                            colors = [colors]
43: (12)                        assert any(verify_length(coloring, n_colors) for n_colors in
colors)
44: (12)                        assert verify_coloring(graph, coloring)
45: (8)                     for strategy, arglist in SPECIAL_TEST_CASES.items():
46: (12)                        for args in arglist:
47: (16)                            check_special_case(strategy, args[0], args[1], args[2])
48: (4)                 def test_interchange_invalid(self):
49: (8)                     graph = one_node_graph()
50: (8)                     for strategy in INTERCHANGE_INVALID:
51: (12)                        pytest.raises(
52: (16)                            nx.NetworkXPointlessConcept,
53: (16)                            nx.coloring.greedy_color,
54: (16)                            graph,
55: (16)                            strategy=strategy,
56: (16)                            interchange=True,
57: (12)                        )
58: (4)                 def test_bad_inputs(self):
59: (8)                     graph = one_node_graph()
60: (8)                     pytest.raises(
61: (12)                        nx.NetworkXError,
62: (12)                        nx.coloring.greedy_color,
63: (12)                        graph,
64: (12)                        strategy="invalid strategy",
65: (8)                     )
66: (4)                 def test_strategy_as_function(self):
67: (8)                     graph = lf_shc()
68: (8)                     colors_1 = nx.coloring.greedy_color(graph, "largest_first")
69: (8)                     colors_2 = nx.coloring.greedy_color(graph,
nx.coloring.strategy_largest_first)
70: (8)                     assert colors_1 == colors_2
71: (4)                 def test_seed_argument(self):
72: (8)                     graph = lf_shc()
73: (8)                     rs = nx.coloring.strategy_random_sequential
74: (8)                     c1 = nx.coloring.greedy_color(graph, lambda g, c: rs(g, c, seed=1))
75: (8)                     for u, v in graph.edges:
76: (12)                        assert c1[u] != c1[v]
77: (4)                 def test_is_coloring(self):
78: (8)                     G = nx.Graph()
79: (8)                     G.add_edges_from([(0, 1), (1, 2)])
80: (8)                     coloring = {0: 0, 1: 1, 2: 0}
81: (8)                     assert is_coloring(G, coloring)
82: (8)                     coloring[0] = 1
83: (8)                     assert not is_coloring(G, coloring)
84: (8)                     assert not is_equitable(G, coloring)
85: (4)                 def test_is_equitable(self):
86: (8)                     G = nx.Graph()
87: (8)                     G.add_edges_from([(0, 1), (1, 2)])
88: (8)                     coloring = {0: 0, 1: 1, 2: 0}
89: (8)                     assert is_equitable(G, coloring)
90: (8)                     G.add_edges_from([(2, 3), (2, 4), (2, 5)])
91: (8)                     coloring[3] = 1
92: (8)                     coloring[4] = 1
93: (8)                     coloring[5] = 1
94: (8)                     assert is_coloring(G, coloring)
95: (8)                     assert not is_equitable(G, coloring)
96: (4)                 def test_num_colors(self):
```

```
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (0, 3)])
(8)                     pytest.raises(nx.NetworkXAlgorithmError, nx.coloring.equitable_color,
G, 2)
(4)                 def test_equitable_color(self):
(8)                     G = nx.fast_gnp_random_graph(n=10, p=0.2, seed=42)
(8)                     coloring = nx.coloring.equitable_color(G, max_degree(G) + 1)
(8)                     assert is_equitable(G, coloring)
(4)                 def test_equitable_color_empty(self):
(8)                     G = nx.empty_graph()
(8)                     coloring = nx.coloring.equitable_color(G, max_degree(G) + 1)
(8)                     assert is_equitable(G, coloring)
(4)                 def test_equitable_color_large(self):
(8)                     G = nx.fast_gnp_random_graph(100, 0.1, seed=42)
(8)                     coloring = nx.coloring.equitable_color(G, max_degree(G) + 1)
(8)                     assert is_equitable(G, coloring, num_colors=max_degree(G) + 1)
(4)                 def test_case_V_plus_not_in_A_cal(self):
(8)                     L = {
(12)                        0: [2, 5],
(12)                        1: [3, 4],
(12)                        2: [0, 8],
(12)                        3: [1, 7],
(12)                        4: [1, 6],
(12)                        5: [0, 6],
(12)                        6: [4, 5],
(12)                        7: [3],
(12)                        8: [2],
(8)                     }
(8)                     F = {
(12)                        0: 0,
(12)                        1: 0,
(12)                        2: 1,
(12)                        3: 1,
(12)                        4: 1,
(12)                        5: 1,
(12)                        6: 2,
(12)                        7: 2,
(12)                        8: 2,
(8)                     }
(8)                     C = nx.algorithms.coloring.equitable_coloring.make_C_from_F(F)
(8)                     N = nx.algorithms.coloring.equitable_coloring.make_N_from_L_C(L, C)
(8)                     H = nx.algorithms.coloring.equitable_coloring.make_H_from_C_N(C, N)
(8)                     nx.algorithms.coloring.equitable_coloring.procedure_P(
(12)                        V_minus=0, V_plus=1, N=N, H=H, F=F, C=C, L=L
(8)                     )
(8)                     check_state(L=L, N=N, H=H, F=F, C=C)
(4)                 def test_cast_no_solo(self):
(8)                     L = {
(12)                        0: [8, 9],
(12)                        1: [10, 11],
(12)                        2: [8],
(12)                        3: [9],
(12)                        4: [10, 11],
(12)                        5: [8],
(12)                        6: [9],
(12)                        7: [10, 11],
(12)                        8: [0, 2, 5],
(12)                        9: [0, 3, 6],
(12)                        10: [1, 4, 7],
(12)                        11: [1, 4, 7],
(8)                     }
(8)                     F = {0: 0, 1: 0, 2: 2, 3: 2, 4: 2, 5: 3, 6: 3, 7: 3, 8: 1, 9: 1, 10:
1, 11: 1}
(8)                     C = nx.algorithms.coloring.equitable_coloring.make_C_from_F(F)
(8)                     N = nx.algorithms.coloring.equitable_coloring.make_N_from_L_C(L, C)
(8)                     H = nx.algorithms.coloring.equitable_coloring.make_H_from_C_N(C, N)
(8)                     nx.algorithms.coloring.equitable_coloring.procedure_P(
(12)                        V_minus=0, V_plus=1, N=N, H=H, F=F, C=C, L=L
(8)                     )
```

```
(8)                 check_state(L=L, N=N, H=H, F=F, C=C)
(4)             def test_hard_prob(self):
(8)                 num_colors, s = 5, 5
(8)                 G = nx.Graph()
(8)                 G.add_edges_from(
(12)                    [
(16)                        (0, 10),
(16)                        (0, 11),
(16)                        (0, 12),
(16)                        (0, 23),
(16)                        (10, 4),
(16)                        (10, 9),
(16)                        (10, 20),
(16)                        (11, 4),
(16)                        (11, 8),
(16)                        (11, 16),
(16)                        (12, 9),
(16)                        (12, 22),
(16)                        (12, 23),
(16)                        (23, 7),
(16)                        (1, 17),
(16)                        (1, 18),
(16)                        (1, 19),
(16)                        (1, 24),
(16)                        (17, 5),
(16)                        (17, 13),
(16)                        (17, 22),
(16)                        (18, 5),
(16)                        (19, 5),
(16)                        (19, 6),
(16)                        (19, 8),
(16)                        (24, 7),
(16)                        (24, 16),
(16)                        (2, 4),
(16)                        (2, 13),
(16)                        (2, 14),
(16)                        (2, 15),
(16)                        (4, 6),
(16)                        (13, 5),
(16)                        (13, 21),
(16)                        (14, 6),
(16)                        (14, 15),
(16)                        (15, 6),
(16)                        (15, 21),
(16)                        (3, 16),
(16)                        (3, 20),
(16)                        (3, 21),
(16)                        (3, 22),
(16)                        (16, 8),
(16)                        (20, 8),
(16)                        (21, 9),
(16)                        (22, 7),
(12)                    ]
(8)                 )
(8)                 F = {node: node // s for node in range(num_colors * s)}
(8)                 F[s - 1] = num_colors - 1
(8)                 params = make_params_from_graph(G=G, F=F)
(8)                 nx.algorithms.coloring.equitable_coloring.procedure_P(
(12)                    V_minus=0, V_plus=num_colors - 1, **params
(8)                 )
(8)                 check_state(**params)
(4)             def test_hardest_prob(self):
(8)                 num_colors, s = 10, 4
(8)                 G = nx.Graph()
(8)                 G.add_edges_from(
(12)                    [
(16)                        (0, 19),
(16)                        (0, 24),
(16)                        (0, 29),
```

```
233: (16)                           (0, 30),
234: (16)                           (0, 35),
235: (16)                           (19, 3),
236: (16)                           (19, 7),
237: (16)                           (19, 9),
238: (16)                           (19, 15),
239: (16)                           (19, 21),
240: (16)                           (19, 24),
241: (16)                           (19, 30),
242: (16)                           (19, 38),
243: (16)                           (24, 5),
244: (16)                           (24, 11),
245: (16)                           (24, 13),
246: (16)                           (24, 20),
247: (16)                           (24, 30),
248: (16)                           (24, 37),
249: (16)                           (24, 38),
250: (16)                           (29, 6),
251: (16)                           (29, 10),
252: (16)                           (29, 13),
253: (16)                           (29, 15),
254: (16)                           (29, 16),
255: (16)                           (29, 17),
256: (16)                           (29, 20),
257: (16)                           (29, 26),
258: (16)                           (30, 6),
259: (16)                           (30, 10),
260: (16)                           (30, 15),
261: (16)                           (30, 22),
262: (16)                           (30, 23),
263: (16)                           (30, 39),
264: (16)                           (35, 6),
265: (16)                           (35, 9),
266: (16)                           (35, 14),
267: (16)                           (35, 18),
268: (16)                           (35, 22),
269: (16)                           (35, 23),
270: (16)                           (35, 25),
271: (16)                           (35, 27),
272: (16)                           (1, 20),
273: (16)                           (1, 26),
274: (16)                           (1, 31),
275: (16)                           (1, 34),
276: (16)                           (1, 38),
277: (16)                           (20, 4),
278: (16)                           (20, 8),
279: (16)                           (20, 14),
280: (16)                           (20, 18),
281: (16)                           (20, 28),
282: (16)                           (20, 33),
283: (16)                           (26, 7),
284: (16)                           (26, 10),
285: (16)                           (26, 14),
286: (16)                           (26, 18),
287: (16)                           (26, 21),
288: (16)                           (26, 32),
289: (16)                           (26, 39),
290: (16)                           (31, 5),
291: (16)                           (31, 8),
292: (16)                           (31, 13),
293: (16)                           (31, 16),
294: (16)                           (31, 17),
295: (16)                           (31, 21),
296: (16)                           (31, 25),
297: (16)                           (31, 27),
298: (16)                           (34, 7),
299: (16)                           (34, 8),
300: (16)                           (34, 13),
301: (16)                           (34, 18),
```

```
302: (16)                           (34, 22),
303: (16)                           (34, 23),
304: (16)                           (34, 25),
305: (16)                           (34, 27),
306: (16)                           (38, 4),
307: (16)                           (38, 9),
308: (16)                           (38, 12),
309: (16)                           (38, 14),
310: (16)                           (38, 21),
311: (16)                           (38, 27),
312: (16)                           (2, 3),
313: (16)                           (2, 18),
314: (16)                           (2, 21),
315: (16)                           (2, 28),
316: (16)                           (2, 32),
317: (16)                           (2, 33),
318: (16)                           (2, 36),
319: (16)                           (2, 37),
320: (16)                           (2, 39),
321: (16)                           (3, 5),
322: (16)                           (3, 9),
323: (16)                           (3, 13),
324: (16)                           (3, 22),
325: (16)                           (3, 23),
326: (16)                           (3, 25),
327: (16)                           (3, 27),
328: (16)                           (18, 6),
329: (16)                           (18, 11),
330: (16)                           (18, 15),
331: (16)                           (18, 39),
332: (16)                           (21, 4),
333: (16)                           (21, 10),
334: (16)                           (21, 14),
335: (16)                           (21, 36),
336: (16)                           (28, 6),
337: (16)                           (28, 10),
338: (16)                           (28, 14),
339: (16)                           (28, 16),
340: (16)                           (28, 17),
341: (16)                           (28, 25),
342: (16)                           (28, 27),
343: (16)                           (32, 5),
344: (16)                           (32, 10),
345: (16)                           (32, 12),
346: (16)                           (32, 16),
347: (16)                           (32, 17),
348: (16)                           (32, 22),
349: (16)                           (32, 23),
350: (16)                           (33, 7),
351: (16)                           (33, 10),
352: (16)                           (33, 12),
353: (16)                           (33, 16),
354: (16)                           (33, 17),
355: (16)                           (33, 25),
356: (16)                           (33, 27),
357: (16)                           (36, 5),
358: (16)                           (36, 8),
359: (16)                           (36, 15),
360: (16)                           (36, 16),
361: (16)                           (36, 17),
362: (16)                           (36, 25),
363: (16)                           (36, 27),
364: (16)                           (37, 5),
365: (16)                           (37, 11),
366: (16)                           (37, 15),
367: (16)                           (37, 16),
368: (16)                           (37, 17),
369: (16)                           (37, 22),
370: (16)                           (37, 23),
```

```
371: (16)                            (39, 7),
372: (16)                            (39, 8),
373: (16)                            (39, 15),
374: (16)                            (39, 22),
375: (16)                            (39, 23),
376: (12)                        ]
377: (8)                    )
378: (8)                    F = {node: node // s for node in range(num_colors * s)}
379: (8)                    F[s - 1] = num_colors - 1  # V- = 0, V+ = num_colors - 1
380: (8)                    params = make_params_from_graph(G=G, F=F)
381: (8)                    nx.algorithms.coloring.equitable_coloring.procedure_P(
382: (12)                        V_minus=0, V_plus=num_colors - 1, **params
383: (8)                    )
384: (8)                    check_state(**params)
385: (4)                def test_strategy_saturation_largest_first(self):
386: (8)                    def color_remaining_nodes(
387: (12)                        G,
388: (12)                        colored_nodes,
389: (12)                        full_color_assignment=None,
390: (12)                        nodes_to_add_between_calls=1,
391: (8)                    ):
392: (12)                        color_assignments = []
393: (12)                        aux_colored_nodes = colored_nodes.copy()
394: (12)                        node_iterator =
nx.algorithms.coloring.greedy_coloring.strategy_saturation_largest_first(
395: (16)                            G, aux_colored_nodes
396: (12)                        )
397: (12)                        for u in node_iterator:
398: (16)                            nbr_colors = {
399: (20)                                aux_colored_nodes[v] for v in G[u] if v in
aux_colored_nodes
400: (16)                            }
401: (16)                            for color in itertools.count():
402: (20)                                if color not in nbr_colors:
403: (24)                                    break
404: (16)                            aux_colored_nodes[u] = color
405: (16)                            color_assignments.append((u, color))
406: (16)                            for i in range(nodes_to_add_between_calls - 1):
407: (20)                                if not len(color_assignments) + len(colored_nodes) >= len(
408: (24)                                    full_color_assignment
409: (20)                                ):
410: (24)                                    full_color_assignment_node, color =
full_color_assignment[
411: (28)                                        len(color_assignments) + len(colored_nodes)
412: (24)                                    ]
413: (24)                                    aux_colored_nodes[full_color_assignment_node] = color
414: (24)                                    color_assignments.append((full_color_assignment_node,
color))
415: (12)                        return color_assignments, aux_colored_nodes
416: (8)                    for G, _, _ in SPECIAL_TEST_CASES["saturation_largest_first"]:
417: (12)                        G = G()
418: (12)                        for nodes_to_add_between_calls in range(1, 5):
419: (16)                            colored_nodes = {}
420: (16)                            full_color_assignment, full_colored_nodes =
color_remaining_nodes(
421: (20)                                G, colored_nodes
422: (16)                            )
423: (16)                            for ind, (node, color) in enumerate(full_color_assignment):
424: (20)                                colored_nodes[node] = color
425: (20)                                (
426: (24)                                    partial_color_assignment,
427: (24)                                    partial_colored_nodes,
428: (20)                                ) = color_remaining_nodes(
429: (24)                                    G,
430: (24)                                    colored_nodes,
431: (24)                                    full_color_assignment=full_color_assignment,
432: (24)                                    nodes_to_add_between_calls=nodes_to_add_between_calls,
433: (20)                                )
434: (20)                                assert full_color_assignment[ind + 1 :] ==
```

```
partial_color_assignment
435: (20)                                assert full_colored_nodes == partial_colored_nodes
436: (0)            def verify_coloring(graph, coloring):
437: (4)                for node in graph.nodes():
438: (8)                    if node not in coloring:
439: (12)                       return False
440: (8)                    color = coloring[node]
441: (8)                    for neighbor in graph.neighbors(node):
442: (12)                       if coloring[neighbor] == color:
443: (16)                           return False
444: (4)                return True
445: (0)            def verify_length(coloring, expected):
446: (4)                coloring = dict_to_sets(coloring)
447: (4)                return len(coloring) == expected
448: (0)            def dict_to_sets(colors):
449: (4)                if len(colors) == 0:
450: (8)                    return []
451: (4)                k = max(colors.values()) + 1
452: (4)                sets = [set() for _ in range(k)]
453: (4)                for node, color in colors.items():
454: (8)                    sets[color].add(node)
455: (4)                return sets
456: (0)            def empty_graph():
457: (4)                return nx.Graph()
458: (0)            def one_node_graph():
459: (4)                graph = nx.Graph()
460: (4)                graph.add_nodes_from([1])
461: (4)                return graph
462: (0)            def two_node_graph():
463: (4)                graph = nx.Graph()
464: (4)                graph.add_nodes_from([1, 2])
465: (4)                graph.add_edges_from([(1, 2)])
466: (4)                return graph
467: (0)            def three_node_clique():
468: (4)                graph = nx.Graph()
469: (4)                graph.add_nodes_from([1, 2, 3])
470: (4)                graph.add_edges_from([(1, 2), (1, 3), (2, 3)])
471: (4)                return graph
472: (0)            def disconnected():
473: (4)                graph = nx.Graph()
474: (4)                graph.add_edges_from([(1, 2), (2, 3), (4, 5), (5, 6)])
475: (4)                return graph
476: (0)            def rs_shc():
477: (4)                graph = nx.Graph()
478: (4)                graph.add_nodes_from([1, 2, 3, 4])
479: (4)                graph.add_edges_from([(1, 2), (2, 3), (3, 4)])
480: (4)                return graph
481: (0)            def slf_shc():
482: (4)                graph = nx.Graph()
483: (4)                graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7])
484: (4)                graph.add_edges_from(
485: (8)                    [(1, 2), (1, 5), (1, 6), (2, 3), (2, 7), (3, 4), (3, 7), (4, 5), (4,
6), (5, 6)]
486: (4)                )
487: (4)                return graph
488: (0)            def slf_hc():
489: (4)                graph = nx.Graph()
490: (4)                graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8])
491: (4)                graph.add_edges_from(
492: (8)                    [
493: (12)                       (1, 2),
494: (12)                       (1, 3),
495: (12)                       (1, 4),
496: (12)                       (1, 5),
497: (12)                       (2, 3),
498: (12)                       (2, 4),
499: (12)                       (2, 6),
500: (12)                       (5, 7),
501: (12)                       (5, 8),
```

```
(12)                        (6, 7),
(12)                        (6, 8),
(12)                        (7, 8),
(8)                     ]
(4)                 )
(4)                 return graph
(0)             def lf_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6])
(4)                 graph.add_edges_from([(6, 1), (1, 4), (4, 3), (3, 2), (2, 5)])
(4)                 return graph
(0)             def lf_hc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7])
(4)                 graph.add_edges_from(
(8)                     [
(12)                        (1, 7),
(12)                        (1, 6),
(12)                        (1, 3),
(12)                        (1, 4),
(12)                        (7, 2),
(12)                        (2, 6),
(12)                        (2, 3),
(12)                        (2, 5),
(12)                        (5, 3),
(12)                        (5, 4),
(12)                        (4, 3),
(8)                     ]
(4)                 )
(4)                 return graph
(0)             def sl_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6])
(4)                 graph.add_edges_from(
(8)                     [(1, 2), (1, 3), (2, 3), (1, 4), (2, 5), (3, 6), (4, 5), (4, 6), (5,
6)]
(4)                 )
(4)                 return graph
(0)             def sl_hc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8])
(4)                 graph.add_edges_from(
(8)                     [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (1, 5),
(12)                        (1, 7),
(12)                        (2, 3),
(12)                        (2, 4),
(12)                        (2, 8),
(12)                        (8, 4),
(12)                        (8, 6),
(12)                        (8, 7),
(12)                        (7, 5),
(12)                        (7, 6),
(12)                        (3, 4),
(12)                        (4, 6),
(12)                        (6, 5),
(12)                        (5, 3),
(8)                     ]
(4)                 )
(4)                 return graph
(0)             def gis_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4])
(4)                 graph.add_edges_from([(1, 2), (2, 3), (3, 4)])
(4)                 return graph
(0)             def gis_hc():
(4)                 graph = nx.Graph()
```

```
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6])
(4)                 graph.add_edges_from([(1, 5), (2, 5), (3, 6), (4, 6), (5, 6)])
(4)                 return graph
(0)             def cs_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5])
(4)                 graph.add_edges_from([(1, 2), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (4,
5)])
(4)                 return graph
(0)             def rsi_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6])
(4)                 graph.add_edges_from(
(8)                     [(1, 2), (1, 5), (1, 6), (2, 3), (3, 4), (4, 5), (4, 6), (5, 6)]
(4)                 )
(4)                 return graph
(0)             def lfi_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7])
(4)                 graph.add_edges_from(
(8)                     [(1, 2), (1, 5), (1, 6), (2, 3), (2, 7), (3, 4), (3, 7), (4, 5), (4,
6), (5, 6)]
(4)                 )
(4)                 return graph
(0)             def lfi_hc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8, 9])
(4)                 graph.add_edges_from(
(8)                     [
(12)                        (1, 2),
(12)                        (1, 5),
(12)                        (1, 6),
(12)                        (1, 7),
(12)                        (2, 3),
(12)                        (2, 8),
(12)                        (2, 9),
(12)                        (3, 4),
(12)                        (3, 8),
(12)                        (3, 9),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (4, 7),
(12)                        (5, 6),
(8)                     ]
(4)                 )
(4)                 return graph
(0)             def sli_shc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7])
(4)                 graph.add_edges_from(
(8)                     [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (1, 5),
(12)                        (1, 7),
(12)                        (2, 3),
(12)                        (2, 6),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (5, 7),
(12)                        (6, 7),
(8)                     ]
(4)                 )
(4)                 return graph
(0)             def sli_hc():
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8, 9])
(4)                 graph.add_edges_from(
```

```
637: (8)                    [
638: (12)                       (1, 2),
639: (12)                       (1, 3),
640: (12)                       (1, 4),
641: (12)                       (1, 5),
642: (12)                       (2, 3),
643: (12)                       (2, 7),
644: (12)                       (2, 8),
645: (12)                       (2, 9),
646: (12)                       (3, 6),
647: (12)                       (3, 7),
648: (12)                       (3, 9),
649: (12)                       (4, 5),
650: (12)                       (4, 6),
651: (12)                       (4, 8),
652: (12)                       (4, 9),
653: (12)                       (5, 6),
654: (12)                       (5, 7),
655: (12)                       (5, 8),
656: (12)                       (6, 7),
657: (12)                       (6, 9),
658: (12)                       (7, 8),
659: (12)                       (8, 9),
660: (8)                    ]
661: (4)                )
662: (4)                return graph
663: (0)            BASIC_TEST_CASES = {
664: (4)                empty_graph: 0,
665: (4)                one_node_graph: 1,
666: (4)                two_node_graph: 2,
667: (4)                disconnected: 2,
668: (4)                three_node_clique: 3,
669: (0)            }
670: (0)            SPECIAL_TEST_CASES = {
671: (4)                "random_sequential": [
672: (8)                    (rs_shc, False, (2, 3)),
673: (8)                    (rs_shc, True, 2),
674: (8)                    (rsi_shc, True, (3, 4)),
675: (4)                ],
676: (4)                "saturation_largest_first": [(slf_shc, False, (3, 4)), (slf_hc, False,
4)],
677: (4)                "largest_first": [
678: (8)                    (lf_shc, False, (2, 3)),
679: (8)                    (lf_hc, False, 4),
680: (8)                    (lf_shc, True, 2),
681: (8)                    (lf_hc, True, 3),
682: (8)                    (lfi_shc, True, (3, 4)),
683: (8)                    (lfi_hc, True, 4),
684: (4)                ],
685: (4)                "smallest_last": [
686: (8)                    (sl_shc, False, (3, 4)),
687: (8)                    (sl_hc, False, 5),
688: (8)                    (sl_shc, True, 3),
689: (8)                    (sl_hc, True, 4),
690: (8)                    (sli_shc, True, (3, 4)),
691: (8)                    (sli_hc, True, 5),
692: (4)                ],
693: (4)                "independent_set": [(gis_shc, False, (2, 3)), (gis_hc, False, 3)],
694: (4)                "connected_sequential": [(cs_shc, False, (3, 4)), (cs_shc, True, 3)],
695: (4)                "connected_sequential_dfs": [(cs_shc, False, (3, 4))],
696: (0)            }
697: (0)            def check_state(L, N, H, F, C):
698: (4)                s = len(C[0])
699: (4)                num_colors = len(C.keys())
700: (4)                assert all(u in L[v] for u in L for v in L[u])
701: (4)                assert all(F[u] != F[v] for u in L for v in L[u])
702: (4)                assert all(len(L[u]) < num_colors for u in L)
703: (4)                assert all(len(C[x]) == s for x in C)
704: (4)                assert all(H[(c1, c2)] >= 0 for c1 in C for c2 in C)
```

```
705: (4)                assert all(N[(u, F[u])] == 0 for u in F)
706: (0)            def max_degree(G):
707: (4)                """Get the maximum degree of any node in G."""
708: (4)                return max(G.degree(node) for node in G.nodes) if len(G.nodes) > 0 else 0
709: (0)            def make_params_from_graph(G, F):
710: (4)                """Returns {N, L, H, C} from the given graph."""
711: (4)                num_nodes = len(G)
712: (4)                L = {u: [] for u in range(num_nodes)}
713: (4)                for u, v in G.edges:
714: (8)                    L[u].append(v)
715: (8)                    L[v].append(u)
716: (4)                C = nx.algorithms.coloring.equitable_coloring.make_C_from_F(F)
717: (4)                N = nx.algorithms.coloring.equitable_coloring.make_N_from_L_C(L, C)
718: (4)                H = nx.algorithms.coloring.equitable_coloring.make_H_from_C_N(C, N)
719: (4)                return {"N": N, "F": F, "C": C, "H": H, "L": L}
```

File 179 - __init__.py:

```
1: (0)              """Functions for computing and measuring community structure.
2: (0)              The ``community`` subpackage can be accessed by using
:mod:`networkx.community`, then accessing the
3: (0)              functions as attributes of ``community``. For example::
4: (4)                  >>> import networkx as nx
5: (4)                  >>> G = nx.barbell_graph(5, 1)
6: (4)                  >>> communities_generator = nx.community.girvan_newman(G)
7: (4)                  >>> top_level_communities = next(communities_generator)
8: (4)                  >>> next_level_communities = next(communities_generator)
9: (4)                  >>> sorted(map(sorted, next_level_communities))
10: (4)                 [[0, 1, 2, 3, 4], [5], [6, 7, 8, 9, 10]]
11: (0)             """
12: (0)             from networkx.algorithms.community.asyn_fluid import *
13: (0)             from networkx.algorithms.community.centrality import *
14: (0)             from networkx.algorithms.community.divisive import *
15: (0)             from networkx.algorithms.community.kclique import *
16: (0)             from networkx.algorithms.community.kernighan_lin import *
17: (0)             from networkx.algorithms.community.label_propagation import *
18: (0)             from networkx.algorithms.community.lukes import *
19: (0)             from networkx.algorithms.community.modularity_max import *
20: (0)             from networkx.algorithms.community.quality import *
21: (0)             from networkx.algorithms.community.community_utils import *
22: (0)             from networkx.algorithms.community.louvain import *
```

File 180 - asyn_fluid.py:

```
1: (0)              """Asynchronous Fluid Communities algorithm for community detection."""
2: (0)              from collections import Counter
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.components import is_connected
5: (0)              from networkx.exception import NetworkXError
6: (0)              from networkx.utils import groups, not_implemented_for, py_random_state
7: (0)              __all__ = ["asyn_fluidc"]
8: (0)              @not_implemented_for("directed")
9: (0)              @not_implemented_for("multigraph")
10: (0)             @py_random_state(3)
11: (0)             @nx._dispatchable
12: (0)             def asyn_fluidc(G, k, max_iter=100, seed=None):
13: (4)                 """Returns communities in `G` as detected by Fluid Communities algorithm.
14: (4)                 The asynchronous fluid communities algorithm is described in
15: (4)                 [1]_. The algorithm is based on the simple idea of fluids interacting
16: (4)                 in an environment, expanding and pushing each other. Its initialization is
17: (4)                 random, so found communities may vary on different executions.
18: (4)                 The algorithm proceeds as follows. First each of the initial k communities
19: (4)                 is initialized in a random vertex in the graph. Then the algorithm
iterates
20: (4)                 over all vertices in a random order, updating the community of each vertex
```

```
 21: (4)                 based on its own community and the communities of its neighbors. This
 22: (4)                 process is performed several times until convergence.
 23: (4)                 At all times, each community has a total density of 1, which is equally
 24: (4)                 distributed among the vertices it contains. If a vertex changes of
 25: (4)                 community, vertex densities of affected communities are adjusted
 26: (4)                 immediately. When a complete iteration over all vertices is done, such
that
 27: (4)                 no vertex changes the community it belongs to, the algorithm has converged
 28: (4)                 and returns.
 29: (4)                 This is the original version of the algorithm described in [1]_.
 30: (4)                 Unfortunately, it does not support weighted graphs yet.
 31: (4)                 Parameters
 32: (4)                 ----------
 33: (4)                 G : NetworkX graph
 34: (8)                     Graph must be simple and undirected.
 35: (4)                 k : integer
 36: (8)                     The number of communities to be found.
 37: (4)                 max_iter : integer
 38: (8)                     The number of maximum iterations allowed. By default 100.
 39: (4)                 seed : integer, random_state, or None (default)
 40: (8)                     Indicator of random number generation state.
 41: (8)                     See :ref:`Randomness<randomness>`.
 42: (4)                 Returns
 43: (4)                 -------
 44: (4)                 communities : iterable
 45: (8)                     Iterable of communities given as sets of nodes.
 46: (4)                 Notes
 47: (4)                 -----
 48: (4)                 k variable is not an optional argument.
 49: (4)                 References
 50: (4)                 ----------
 51: (4)                 .. [1] Parés F., Garcia-Gasulla D. et al. "Fluid Communities: A
 52: (7)                    Competitive and Highly Scalable Community Detection Algorithm".
 53: (7)                    [https://arxiv.org/pdf/1703.09307.pdf].
 54: (4)                 """
 55: (4)                 if not isinstance(k, int):
 56: (8)                     raise NetworkXError("k must be an integer.")
 57: (4)                 if not k > 0:
 58: (8)                     raise NetworkXError("k must be greater than 0.")
 59: (4)                 if not is_connected(G):
 60: (8)                     raise NetworkXError("Fluid Communities require connected Graphs.")
 61: (4)                 if len(G) < k:
 62: (8)                     raise NetworkXError("k cannot be bigger than the number of nodes.")
 63: (4)                 max_density = 1.0
 64: (4)                 vertices = list(G)
 65: (4)                 seed.shuffle(vertices)
 66: (4)                 communities = {n: i for i, n in enumerate(vertices[:k])}
 67: (4)                 density = {}
 68: (4)                 com_to_numvertices = {}
 69: (4)                 for vertex in communities:
 70: (8)                     com_to_numvertices[communities[vertex]] = 1
 71: (8)                     density[communities[vertex]] = max_density
 72: (4)                 iter_count = 0
 73: (4)                 cont = True
 74: (4)                 while cont:
 75: (8)                     cont = False
 76: (8)                     iter_count += 1
 77: (8)                     vertices = list(G)
 78: (8)                     seed.shuffle(vertices)
 79: (8)                     for vertex in vertices:
 80: (12)                        com_counter = Counter()
 81: (12)                        try:
 82: (16)                            com_counter.update({communities[vertex]:
density[communities[vertex]]})
 83: (12)                        except KeyError:
 84: (16)                            pass
 85: (12)                        for v in G[vertex]:
 86: (16)                            try:
 87: (20)                                com_counter.update({communities[v]:
```

```
density[communities[v]]})
88: (16)                            except KeyError:
89: (20)                                continue
90: (12)                        new_com = -1
91: (12)                        if len(com_counter.keys()) > 0:
92: (16)                            max_freq = max(com_counter.values())
93: (16)                            best_communities = [
94: (20)                                com
95: (20)                                for com, freq in com_counter.items()
96: (20)                                if (max_freq - freq) < 0.0001
97: (16)                            ]
98: (16)                            try:
99: (20)                                if communities[vertex] in best_communities:
100: (24)                                   new_com = communities[vertex]
101: (16)                           except KeyError:
102: (20)                               pass
103: (16)                           if new_com == -1:
104: (20)                               cont = True
105: (20)                               new_com = seed.choice(best_communities)
106: (20)                               try:
107: (24)                                   com_to_numvertices[communities[vertex]] -= 1
108: (24)                                   density[communities[vertex]] = (
109: (28)                                       max_density /
com_to_numvertices[communities[vertex]]
110: (24)                                   )
111: (20)                               except KeyError:
112: (24)                                   pass
113: (20)                               communities[vertex] = new_com
114: (20)                               com_to_numvertices[communities[vertex]] += 1
115: (20)                               density[communities[vertex]] = (
116: (24)                                   max_density / com_to_numvertices[communities[vertex]]
117: (20)                               )
118: (8)                    if iter_count > max_iter:
119: (12)                       break
120: (4)                return iter(groups(communities).values())

----------------------------------------

File 181 - centrality.py:

1: (0)              """Functions for computing communities based on centrality notions."""
2: (0)              import networkx as nx
3: (0)              __all__ = ["girvan_newman"]
4: (0)              @nx._dispatchable(preserve_edge_attrs="most_valuable_edge")
5: (0)              def girvan_newman(G, most_valuable_edge=None):
6: (4)                  """Finds communities in a graph using the Girvan–Newman method.
7: (4)                  Parameters
8: (4)                  ----------
9: (4)                  G : NetworkX graph
10: (4)                 most_valuable_edge : function
11: (8)                     Function that takes a graph as input and outputs an edge. The
12: (8)                     edge returned by this function will be recomputed and removed at
13: (8)                     each iteration of the algorithm.
14: (8)                     If not specified, the edge with the highest
15: (8)                     :func:`networkx.edge_betweenness_centrality` will be used.
16: (4)                 Returns
17: (4)                 -------
18: (4)                 iterator
19: (8)                     Iterator over tuples of sets of nodes in `G`. Each set of node
20: (8)                     is a community, each tuple is a sequence of communities at a
21: (8)                     particular level of the algorithm.
22: (4)                 Examples
23: (4)                 --------
24: (4)                 To get the first pair of communities::
25: (8)                     >>> G = nx.path_graph(10)
26: (8)                     >>> comp = nx.community.girvan_newman(G)
27: (8)                     >>> tuple(sorted(c) for c in next(comp))
28: (8)                     ([0, 1, 2, 3, 4], [5, 6, 7, 8, 9])
29: (4)                 To get only the first *k* tuples of communities, use
```

```
 30: (4)                 :func:`itertools.islice`::
 31: (8)                     >>> import itertools
 32: (8)                     >>> G = nx.path_graph(8)
 33: (8)                     >>> k = 2
 34: (8)                     >>> comp = nx.community.girvan_newman(G)
 35: (8)                     >>> for communities in itertools.islice(comp, k):
 36: (8)                     ...     print(tuple(sorted(c) for c in communities))
 37: (8)                     ...
 38: (8)                     ([0, 1, 2, 3], [4, 5, 6, 7])
 39: (8)                     ([0, 1], [2, 3], [4, 5, 6, 7])
 40: (4)                 To stop getting tuples of communities once the number of communities
 41: (4)                 is greater than *k*, use :func:`itertools.takewhile`::
 42: (8)                     >>> import itertools
 43: (8)                     >>> G = nx.path_graph(8)
 44: (8)                     >>> k = 4
 45: (8)                     >>> comp = nx.community.girvan_newman(G)
 46: (8)                     >>> limited = itertools.takewhile(lambda c: len(c) <= k, comp)
 47: (8)                     >>> for communities in limited:
 48: (8)                     ...     print(tuple(sorted(c) for c in communities))
 49: (8)                     ...
 50: (8)                     ([0, 1, 2, 3], [4, 5, 6, 7])
 51: (8)                     ([0, 1], [2, 3], [4, 5, 6, 7])
 52: (8)                     ([0, 1], [2, 3], [4, 5], [6, 7])
 53: (4)                 To just choose an edge to remove based on the weight::
 54: (8)                     >>> from operator import itemgetter
 55: (8)                     >>> G = nx.path_graph(10)
 56: (8)                     >>> edges = G.edges()
 57: (8)                     >>> nx.set_edge_attributes(G, {(u, v): v for u, v in edges}, "weight")
 58: (8)                     >>> def heaviest(G):
 59: (8)                     ...     u, v, w = max(G.edges(data="weight"), key=itemgetter(2))
 60: (8)                     ...     return (u, v)
 61: (8)                     ...
 62: (8)                     >>> comp = nx.community.girvan_newman(G, most_valuable_edge=heaviest)
 63: (8)                     >>> tuple(sorted(c) for c in next(comp))
 64: (8)                     ([0, 1, 2, 3, 4, 5, 6, 7, 8], [9])
 65: (4)                 To utilize edge weights when choosing an edge with, for example, the
 66: (4)                 highest betweenness centrality::
 67: (8)                     >>> from networkx import edge_betweenness_centrality as betweenness
 68: (8)                     >>> def most_central_edge(G):
 69: (8)                     ...     centrality = betweenness(G, weight="weight")
 70: (8)                     ...     return max(centrality, key=centrality.get)
 71: (8)                     ...
 72: (8)                     >>> G = nx.path_graph(10)
 73: (8)                     >>> comp = nx.community.girvan_newman(G,
most_valuable_edge=most_central_edge)
 74: (8)                     >>> tuple(sorted(c) for c in next(comp))
 75: (8)                     ([0, 1, 2, 3, 4], [5, 6, 7, 8, 9])
 76: (4)                 To specify a different ranking algorithm for edges, use the
 77: (4)                 `most_valuable_edge` keyword argument::
 78: (8)                     >>> from networkx import edge_betweenness_centrality
 79: (8)                     >>> from random import random
 80: (8)                     >>> def most_central_edge(G):
 81: (8)                     ...     centrality = edge_betweenness_centrality(G)
 82: (8)                     ...     max_cent = max(centrality.values())
 83: (8)                     ...     # Scale the centrality values so they are between 0 and 1,
 84: (8)                     ...     # and add some random noise.
 85: (8)                     ...     centrality = {e: c / max_cent for e, c in centrality.items()}
 86: (8)                     ...     # Add some random noise.
 87: (8)                     ...     centrality = {e: c + random() for e, c in centrality.items()}
 88: (8)                     ...     return max(centrality, key=centrality.get)
 89: (8)                     ...
 90: (8)                     >>> G = nx.path_graph(10)
 91: (8)                     >>> comp = nx.community.girvan_newman(G,
most_valuable_edge=most_central_edge)
 92: (4)                 Notes
 93: (4)                 -----
 94: (4)                 The Girvan–Newman algorithm detects communities by progressively
 95: (4)                 removing edges from the original graph. The algorithm removes the
 96: (4)                 "most valuable" edge, traditionally the edge with the highest
```

```
97: (4)                 betweenness centrality, at each step. As the graph breaks down into
98: (4)                 pieces, the tightly knit community structure is exposed and the
99: (4)                 result can be depicted as a dendrogram.
100: (4)                """
101: (4)                if G.number_of_edges() == 0:
102: (8)                    yield tuple(nx.connected_components(G))
103: (8)                    return
104: (4)                if most_valuable_edge is None:
105: (8)                    def most_valuable_edge(G):
106: (12)                       """Returns the edge with the highest betweenness centrality
107: (12)                       in the graph `G`.
108: (12)                       """
109: (12)                       betweenness = nx.edge_betweenness_centrality(G)
110: (12)                       return max(betweenness, key=betweenness.get)
111: (4)                g = G.copy().to_undirected()
112: (4)                g.remove_edges_from(nx.selfloop_edges(g))
113: (4)                while g.number_of_edges() > 0:
114: (8)                    yield _without_most_central_edges(g, most_valuable_edge)
115: (0)            def _without_most_central_edges(G, most_valuable_edge):
116: (4)                """Returns the connected components of the graph that results from
117: (4)                repeatedly removing the most "valuable" edge in the graph.
118: (4)                `G` must be a non-empty graph. This function modifies the graph `G`
119: (4)                in-place; that is, it removes edges on the graph `G`.
120: (4)                `most_valuable_edge` is a function that takes the graph `G` as input
121: (4)                (or a subgraph with one or more edges of `G` removed) and returns an
122: (4)                edge. That edge will be removed and this process will be repeated
123: (4)                until the number of connected components in the graph increases.
124: (4)                """
125: (4)                original_num_components = nx.number_connected_components(G)
126: (4)                num_new_components = original_num_components
127: (4)                while num_new_components <= original_num_components:
128: (8)                    edge = most_valuable_edge(G)
129: (8)                    G.remove_edge(*edge)
130: (8)                    new_components = tuple(nx.connected_components(G))
131: (8)                    num_new_components = len(new_components)
132: (4)                return new_components
```

File 182 - community_utils.py:

```
1: (0)              """Helper functions for community-finding algorithms."""
2: (0)              import networkx as nx
3: (0)              __all__ = ["is_partition"]
4: (0)              @nx._dispatchable
5: (0)              def is_partition(G, communities):
6: (4)                  """Returns *True* if `communities` is a partition of the nodes of `G`.
7: (4)                  A partition of a universe set is a family of pairwise disjoint sets
8: (4)                  whose union is the entire universe set.
9: (4)                  Parameters
10: (4)                 ----------
11: (4)                 G : NetworkX graph.
12: (4)                 communities : list or iterable of sets of nodes
13: (8)                     If not a list, the iterable is converted internally to a list.
14: (8)                     If it is an iterator it is exhausted.
15: (4)                 """
16: (4)                 if not isinstance(communities, list):
17: (8)                     communities = list(communities)
18: (4)                 nodes = {n for c in communities for n in c if n in G}
19: (4)                 return len(G) == len(nodes) == sum(len(c) for c in communities)
```

File 183 - kclique.py:

```
1: (0)              from collections import defaultdict
2: (0)              import networkx as nx
3: (0)              __all__ = ["k_clique_communities"]
4: (0)              @nx._dispatchable
```

```
5: (0)              def k_clique_communities(G, k, cliques=None):
6: (4)                  """Find k-clique communities in graph using the percolation method.
7: (4)                  A k-clique community is the union of all cliques of size k that
8: (4)                  can be reached through adjacent (sharing k-1 nodes) k-cliques.
9: (4)                  Parameters
10: (4)                 ----------
11: (4)                 G : NetworkX graph
12: (4)                 k : int
13: (7)                    Size of smallest clique
14: (4)                 cliques: list or generator
15: (7)                    Precomputed cliques (use networkx.find_cliques(G))
16: (4)                 Returns
17: (4)                 -------
18: (4)                 Yields sets of nodes, one for each k-clique community.
19: (4)                 Examples
20: (4)                 --------
21: (4)                 >>> G = nx.complete_graph(5)
22: (4)                 >>> K5 = nx.convert_node_labels_to_integers(G, first_label=2)
23: (4)                 >>> G.add_edges_from(K5.edges())
24: (4)                 >>> c = list(nx.community.k_clique_communities(G, 4))
25: (4)                 >>> sorted(list(c[0]))
26: (4)                 [0, 1, 2, 3, 4, 5, 6]
27: (4)                 >>> list(nx.community.k_clique_communities(G, 6))
28: (4)                 []
29: (4)                 References
30: (4)                 ----------
31: (4)                 .. [1] Gergely Palla, Imre Derényi, Illés Farkas1, and Tamás Vicsek,
32: (7)                    Uncovering the overlapping community structure of complex networks
33: (7)                    in nature and society Nature 435, 814-818, 2005,
34: (7)                    doi:10.1038/nature03607
35: (4)                 """
36: (4)                 if k < 2:
37: (8)                     raise nx.NetworkXError(f"k={k}, k must be greater than 1.")
38: (4)                 if cliques is None:
39: (8)                     cliques = nx.find_cliques(G)
40: (4)                 cliques = [frozenset(c) for c in cliques if len(c) >= k]
41: (4)                 membership_dict = defaultdict(list)
42: (4)                 for clique in cliques:
43: (8)                     for node in clique:
44: (12)                        membership_dict[node].append(clique)
45: (4)                 perc_graph = nx.Graph()
46: (4)                 perc_graph.add_nodes_from(cliques)
47: (4)                 for clique in cliques:
48: (8)                     for adj_clique in _get_adjacent_cliques(clique, membership_dict):
49: (12)                        if len(clique.intersection(adj_clique)) >= (k - 1):
50: (16)                            perc_graph.add_edge(clique, adj_clique)
51: (4)                 for component in nx.connected_components(perc_graph):
52: (8)                     yield (frozenset.union(*component))
53: (0)             def _get_adjacent_cliques(clique, membership_dict):
54: (4)                 adjacent_cliques = set()
55: (4)                 for n in clique:
56: (8)                     for adj_clique in membership_dict[n]:
57: (12)                        if clique != adj_clique:
58: (16)                            adjacent_cliques.add(adj_clique)
59: (4)                 return adjacent_cliques
```

File 184 - divisive.py:

```
1: (0)              import functools
2: (0)              import networkx as nx
3: (0)              __all__ = [
4: (4)                  "edge_betweenness_partition",
5: (4)                  "edge_current_flow_betweenness_partition",
6: (0)              ]
7: (0)              @nx._dispatchable(edge_attrs="weight")
8: (0)              def edge_betweenness_partition(G, number_of_sets, *, weight=None):
9: (4)                  """Partition created by iteratively removing the highest edge betweenness
```

```
edge.
    This algorithm works by calculating the edge betweenness for all
    edges and removing the edge with the highest value. It is then
    determined whether the graph has been broken into at least
    `number_of_sets` connected components.
    If not the process is repeated.
    Parameters
    ----------
    G : NetworkX Graph, DiGraph or MultiGraph
      Graph to be partitioned
    number_of_sets : int
      Number of sets in the desired partition of the graph
    weight : key, optional, default=None
      The key to use if using weights for edge betweenness calculation
    Returns
    -------
    C : list of sets
      Partition of the nodes of G
    Raises
    ------
    NetworkXError
      If number_of_sets is <= 0 or if number_of_sets > len(G)
    Examples
    --------
    >>> G = nx.karate_club_graph()
    >>> part = nx.community.edge_betweenness_partition(G, 2)
    >>> {0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21} in part
    True
    >>> {2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33} in part
    True
    See Also
    --------
    edge_current_flow_betweenness_partition
    Notes
    -----
    This algorithm is fairly slow, as both the calculation of connected
    components and edge betweenness relies on all pairs shortest
    path algorithms. They could potentially be combined to cut down
    on overall computation time.
    References
    ----------
    .. [1] Santo Fortunato 'Community Detection in Graphs' Physical Reports
       Volume 486, Issue 3-5 p. 75-174
       http://arxiv.org/abs/0906.0612
    """
    if number_of_sets <= 0:
        raise nx.NetworkXError("number_of_sets must be >0")
    if number_of_sets == 1:
        return [set(G)]
    if number_of_sets == len(G):
        return [{n} for n in G]
    if number_of_sets > len(G):
        raise nx.NetworkXError("number_of_sets must be <= len(G)")
    H = G.copy()
    partition = list(nx.connected_components(H))
    while len(partition) < number_of_sets:
        ranking = nx.edge_betweenness_centrality(H, weight=weight)
        edge = max(ranking, key=ranking.get)
        H.remove_edge(*edge)
        partition = list(nx.connected_components(H))
    return partition
@nx._dispatchable(edge_attrs="weight")
def edge_current_flow_betweenness_partition(G, number_of_sets, *,
weight=None):
    """Partition created by removing the highest edge current flow betweenness
edge.
    This algorithm works by calculating the edge current flow
    betweenness for all edges and removing the edge with the
```

```
(4)                 highest value. It is then determined whether the graph has
(4)                 been broken into at least `number_of_sets` connected
(4)                 components. If not the process is repeated.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph, DiGraph or MultiGraph
(6)                   Graph to be partitioned
(4)                 number_of_sets : int
(6)                   Number of sets in the desired partition of the graph
(4)                 weight : key, optional (default=None)
(6)                   The edge attribute key to use as weights for
(6)                   edge current flow betweenness calculations
(4)                 Returns
(4)                 -------
(4)                 C : list of sets
(6)                   Partition of G
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(6)                   If number_of_sets is <= 0 or number_of_sets > len(G)
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.karate_club_graph()
(4)                 >>> part = nx.community.edge_current_flow_betweenness_partition(G, 2)
(4)                 >>> {0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 16, 17, 19, 21} in part
(4)                 True
(4)                 >>> {8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33} in
part
(4)                 True
(4)                 See Also
(4)                 --------
(4)                 edge_betweenness_partition
(4)                 Notes
(4)                 -----
(4)                 This algorithm is extremely slow, as the recalculation of the edge
(4)                 current flow betweenness is extremely slow.
(4)                 References
(4)                 ----------
(4)                 .. [1] Santo Fortunato 'Community Detection in Graphs' Physical Reports
(7)                    Volume 486, Issue 3-5 p. 75-174
(7)                    http://arxiv.org/abs/0906.0612
(4)                 """
(4)                 if number_of_sets <= 0:
(8)                     raise nx.NetworkXError("number_of_sets must be >0")
(4)                 elif number_of_sets == 1:
(8)                     return [set(G)]
(4)                 elif number_of_sets == len(G):
(8)                     return [{n} for n in G]
(4)                 elif number_of_sets > len(G):
(8)                     raise nx.NetworkXError("number_of_sets must be <= len(G)")
(4)                 rank = functools.partial(
(8)                     nx.edge_current_flow_betweenness_centrality, normalized=False,
weight=weight
(4)                 )
(4)                 H = G.copy()
(4)                 partition = list(nx.connected_components(H))
(4)                 if len(partition) > 1:
(8)                     Hcc_subgraphs = [H.subgraph(cc).copy() for cc in partition]
(4)                 else:
(8)                     Hcc_subgraphs = [H]
(4)                 ranking = {}
(4)                 for Hcc in Hcc_subgraphs:
(8)                     ranking.update(rank(Hcc))
(4)                 while len(partition) < number_of_sets:
(8)                     edge = max(ranking, key=ranking.get)
(8)                     for cc, Hcc in zip(partition, Hcc_subgraphs):
(12)                        if edge[0] in cc:
(16)                            Hcc.remove_edge(*edge)
(16)                            del ranking[edge]
```

```
142: (16)                           splitcc_list = list(nx.connected_components(Hcc))
143: (16)                           if len(splitcc_list) > 1:
144: (20)                               cc_new = min(splitcc_list, key=len)
145: (20)                               Hcc_new = Hcc.subgraph(cc_new).copy()
146: (20)                               newranks = rank(Hcc_new)
147: (20)                               for e, r in newranks.items():
148: (24)                                   ranking[e if e in ranking else e[::-1]] = r
149: (20)                               partition.append(cc_new)
150: (20)                               Hcc_subgraphs.append(Hcc_new)
151: (20)                               Hcc.remove_nodes_from(cc_new)
152: (20)                               cc.difference_update(cc_new)
153: (16)                           newranks = rank(Hcc)
154: (16)                           for e, r in newranks.items():
155: (20)                               ranking[e if e in ranking else e[::-1]] = r
156: (16)                           break
157: (4)                return partition
```

File 185 - kernighan_lin.py:

```
1: (0)              """Functions for computing the Kernighan–Lin bipartition algorithm."""
2: (0)              from itertools import count
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.community.community_utils import is_partition
5: (0)              from networkx.utils import BinaryHeap, not_implemented_for, py_random_state
6: (0)              __all__ = ["kernighan_lin_bisection"]
7: (0)              def _kernighan_lin_sweep(edges, side):
8: (4)                  """
9: (4)                  This is a modified form of Kernighan-Lin, which moves single nodes at a
10: (4)                 time, alternating between sides to keep the bisection balanced.  We keep
11: (4)                 two min-heaps of swap costs to make optimal-next-move selection fast.
12: (4)                 """
13: (4)                 costs0, costs1 = costs = BinaryHeap(), BinaryHeap()
14: (4)                 for u, side_u, edges_u in zip(count(), side, edges):
15: (8)                     cost_u = sum(w if side[v] else -w for v, w in edges_u)
16: (8)                     costs[side_u].insert(u, cost_u if side_u else -cost_u)
17: (4)                 def _update_costs(costs_x, x):
18: (8)                     for y, w in edges[x]:
19: (12)                        costs_y = costs[side[y]]
20: (12)                        cost_y = costs_y.get(y)
21: (12)                        if cost_y is not None:
22: (16)                            cost_y += 2 * (-w if costs_x is costs_y else w)
23: (16)                            costs_y.insert(y, cost_y, True)
24: (4)                 i = 0
25: (4)                 totcost = 0
26: (4)                 while costs0 and costs1:
27: (8)                     u, cost_u = costs0.pop()
28: (8)                     _update_costs(costs0, u)
29: (8)                     v, cost_v = costs1.pop()
30: (8)                     _update_costs(costs1, v)
31: (8)                     totcost += cost_u + cost_v
32: (8)                     i += 1
33: (8)                     yield totcost, i, (u, v)
34: (0)             @not_implemented_for("directed")
35: (0)             @py_random_state(4)
36: (0)             @nx._dispatchable(edge_attrs="weight")
37: (0)             def kernighan_lin_bisection(G, partition=None, max_iter=10, weight="weight",
seed=None):
38: (4)                 """Partition a graph into two blocks using the Kernighan–Lin
39: (4)                 algorithm.
40: (4)                 This algorithm partitions a network into two sets by iteratively
41: (4)                 swapping pairs of nodes to reduce the edge cut between the two sets.  The
42: (4)                 pairs are chosen according to a modified form of Kernighan-Lin [1]_, which
43: (4)                 moves node individually, alternating between sides to keep the bisection
44: (4)                 balanced.
45: (4)                 Parameters
46: (4)                 ----------
47: (4)                 G : NetworkX graph
```

```
    Graph must be undirected.
partition : tuple
    Pair of iterables containing an initial partition. If not
    specified, a random balanced partition is used.
max_iter : int
    Maximum number of times to attempt swaps to find an
    improvement before giving up.
weight : key
    Edge data key to use as weight. If None, the weights are all
    set to one.
seed : integer, random_state, or None (default)
    Indicator of random number generation state.
    See :ref:`Randomness<randomness>`.
    Only used if partition is None
Returns
-------
partition : tuple
    A pair of sets of nodes representing the bipartition.
Raises
------
NetworkXError
    If partition is not a valid partition of the nodes of the graph.
References
----------
.. [1] Kernighan, B. W.; Lin, Shen (1970).
   "An efficient heuristic procedure for partitioning graphs."
   *Bell Systems Technical Journal* 49: 291--307.
   Oxford University Press 2011.
"""
n = len(G)
labels = list(G)
seed.shuffle(labels)
index = {v: i for i, v in enumerate(labels)}
if partition is None:
    side = [0] * (n // 2) + [1] * ((n + 1) // 2)
else:
    try:
        A, B = partition
    except (TypeError, ValueError) as err:
        raise nx.NetworkXError("partition must be two sets") from err
    if not is_partition(G, (A, B)):
        raise nx.NetworkXError("partition invalid")
    side = [0] * n
    for a in A:
        side[index[a]] = 1
if G.is_multigraph():
    edges = [
        [
            (index[u], sum(e.get(weight, 1) for e in d.values()))
            for u, d in G[v].items()
        ]
        for v in labels
    ]
else:
    edges = [
        [(index[u], e.get(weight, 1)) for u, e in G[v].items()] for v in labels
    ]
for i in range(max_iter):
    costs = list(_kernighan_lin_sweep(edges, side))
    min_cost, min_i, _ = min(costs)
    if min_cost >= 0:
        break
    for _, _, (u, v) in costs[:min_i]:
        side[u] = 1
        side[v] = 0
A = {u for u, s in zip(labels, side) if s == 0}
B = {u for u, s in zip(labels, side) if s == 1}
return A, B
```

```
----------------------------------------

File 186 - label_propagation.py:

1: (0)              """
2: (0)              Label propagation community detection algorithms.
3: (0)              """
4: (0)              from collections import Counter, defaultdict, deque
5: (0)              import networkx as nx
6: (0)              from networkx.utils import groups, not_implemented_for, py_random_state
7: (0)              __all__ = [
8: (4)                  "label_propagation_communities",
9: (4)                  "asyn_lpa_communities",
10: (4)                 "fast_label_propagation_communities",
11: (0)             ]
12: (0)             @py_random_state("seed")
13: (0)             @nx._dispatchable(edge_attrs="weight")
14: (0)             def fast_label_propagation_communities(G, *, weight=None, seed=None):
15: (4)                 """Returns communities in `G` as detected by fast label propagation.
16: (4)                 The fast label propagation algorithm is described in [1]_. The algorithm
is
17: (4)                 probabilistic and the found communities may vary in different executions.
18: (4)                 The algorithm operates as follows. First, the community label of each node
is
19: (4)                 set to a unique label. The algorithm then repeatedly updates the labels of
20: (4)                 the nodes to the most frequent label in their neighborhood. In case of
ties,
21: (4)                 a random label is chosen from the most frequent labels.
22: (4)                 The algorithm maintains a queue of nodes that still need to be processed.
23: (4)                 Initially, all nodes are added to the queue in a random order. Then the
nodes
24: (4)                 are removed from the queue one by one and processed. If a node updates its
label,
25: (4)                 all its neighbors that have a different label are added to the queue (if
not
26: (4)                 already in the queue). The algorithm stops when the queue is empty.
27: (4)                 Parameters
28: (4)                 ----------
29: (4)                 G : Graph, DiGraph, MultiGraph, or MultiDiGraph
30: (8)                     Any NetworkX graph.
31: (4)                 weight : string, or None (default)
32: (8)                     The edge attribute representing a non-negative weight of an edge. If
None,
33: (8)                     each edge is assumed to have weight one. The weight of an edge is used
in
34: (8)                     determining the frequency with which a label appears among the
neighbors of
35: (8)                     a node (edge with weight `w` is equivalent to `w` unweighted edges).
36: (4)                 seed : integer, random_state, or None (default)
37: (8)                     Indicator of random number generation state. See
:ref:`Randomness<randomness>`.
38: (4)                 Returns
39: (4)                 -------
40: (4)                 communities : iterable
41: (8)                     Iterable of communities given as sets of nodes.
42: (4)                 Notes
43: (4)                 -----
44: (4)                 Edge directions are ignored for directed graphs.
45: (4)                 Edge weights must be non-negative numbers.
46: (4)                 References
47: (4)                 ----------
48: (4)                 .. [1] Vincent A. Traag & Lovro Šubelj. "Large network community detection
by
49: (7)                    fast label propagation." Scientific Reports 13 (2023): 2701.
50: (7)                    https://doi.org/10.1038/s41598-023-29610-z
51: (4)                 """
52: (4)                 nodes_queue = deque(G)
53: (4)                 seed.shuffle(nodes_queue)
```

```
54: (4)                 nodes_set = set(G)
55: (4)                 comms = {node: i for i, node in enumerate(G)}
56: (4)                 while nodes_queue:
57: (8)                     node = nodes_queue.popleft()
58: (8)                     nodes_set.remove(node)
59: (8)                     if G.degree(node) > 0:
60: (12)                        label_freqs = _fast_label_count(G, comms, node, weight)
61: (12)                        max_freq = max(label_freqs.values())
62: (12)                        comm = seed.choice(
63: (16)                            [comm for comm in label_freqs if label_freqs[comm] ==
max_freq]
64: (12)                        )
65: (12)                        if comms[node] != comm:
66: (16)                            comms[node] = comm
67: (16)                            for nbr in nx.all_neighbors(G, node):
68: (20)                                if comms[nbr] != comm and nbr not in nodes_set:
69: (24)                                    nodes_queue.append(nbr)
70: (24)                                    nodes_set.add(nbr)
71: (4)                 yield from groups(comms).values()
72: (0)             def _fast_label_count(G, comms, node, weight=None):
73: (4)                 """Computes the frequency of labels in the neighborhood of a node.
74: (4)                 Returns a dictionary keyed by label to the frequency of that label.
75: (4)                 """
76: (4)                 if weight is None:
77: (8)                     if not G.is_multigraph():
78: (12)                        label_freqs = Counter(map(comms.get, nx.all_neighbors(G, node)))
79: (8)                     else:
80: (12)                        label_freqs = defaultdict(int)
81: (12)                        for nbr in G[node]:
82: (16)                            label_freqs[comms[nbr]] += len(G[node][nbr])
83: (12)                        if G.is_directed():
84: (16)                            for nbr in G.pred[node]:
85: (20)                                label_freqs[comms[nbr]] += len(G.pred[node][nbr])
86: (4)                 else:
87: (8)                     label_freqs = defaultdict(float)
88: (8)                     for _, nbr, w in G.edges(node, data=weight, default=1):
89: (12)                        label_freqs[comms[nbr]] += w
90: (8)                     if G.is_directed():
91: (12)                        for nbr, _, w in G.in_edges(node, data=weight, default=1):
92: (16)                            label_freqs[comms[nbr]] += w
93: (4)                 return label_freqs
94: (0)             @py_random_state(2)
95: (0)             @nx._dispatchable(edge_attrs="weight")
96: (0)             def asyn_lpa_communities(G, weight=None, seed=None):
97: (4)                 """Returns communities in `G` as detected by asynchronous label
98: (4)                 propagation.
99: (4)                 The asynchronous label propagation algorithm is described in
100: (4)                [1]_. The algorithm is probabilistic and the found communities may
101: (4)                vary on different executions.
102: (4)                The algorithm proceeds as follows. After initializing each node with
103: (4)                a unique label, the algorithm repeatedly sets the label of a node to
104: (4)                be the label that appears most frequently among that nodes
105: (4)                neighbors. The algorithm halts when each node has the label that
106: (4)                appears most frequently among its neighbors. The algorithm is
107: (4)                asynchronous because each node is updated without waiting for
108: (4)                updates on the remaining nodes.
109: (4)                This generalized version of the algorithm in [1]_ accepts edge
110: (4)                weights.
111: (4)                Parameters
112: (4)                ----------
113: (4)                G : Graph
114: (4)                weight : string
115: (8)                    The edge attribute representing the weight of an edge.
116: (8)                    If None, each edge is assumed to have weight one. In this
117: (8)                    algorithm, the weight of an edge is used in determining the
118: (8)                    frequency with which a label appears among the neighbors of a
119: (8)                    node: a higher weight means the label appears more often.
120: (4)                seed : integer, random_state, or None (default)
121: (8)                    Indicator of random number generation state.
```

```
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             communities : iterable
(8)                 Iterable of communities given as sets of nodes.
(4)             Notes
(4)             -----
(4)             Edge weight attributes must be numerical.
(4)             References
(4)             ----------
(4)             .. [1] Raghavan, Usha Nandini, Réka Albert, and Soundar Kumara. "Near
(11)                   linear time algorithm to detect community structures in large-scale
(11)                   networks." Physical Review E 76.3 (2007): 036106.
(4)             """
(4)             labels = {n: i for i, n in enumerate(G)}
(4)             cont = True
(4)             while cont:
(8)                 cont = False
(8)                 nodes = list(G)
(8)                 seed.shuffle(nodes)
(8)                 for node in nodes:
(12)                    if not G[node]:
(16)                        continue
(12)                    if weight is None:
(16)                        label_freq = Counter(map(labels.get, G[node]))
(12)                    else:
(16)                        label_freq = defaultdict(float)
(16)                        for _, v, wt in G.edges(node, data=weight, default=1):
(20)                            label_freq[labels[v]] += wt
(12)                    max_freq = max(label_freq.values())
(12)                    best_labels = [
(16)                        label for label, freq in label_freq.items() if freq == max_freq
(12)                    ]
(12)                    if labels[node] not in best_labels:
(16)                        labels[node] = seed.choice(best_labels)
(16)                        cont = True
(4)             yield from groups(labels).values()
(0)         @not_implemented_for("directed")
(0)         @nx._dispatchable
(0)         def label_propagation_communities(G):
(4)             """Generates community sets determined by label propagation
(4)             Finds communities in `G` using a semi-synchronous label propagation
(4)             method [1]_. This method combines the advantages of both the synchronous
(4)             and asynchronous models. Not implemented for directed graphs.
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 An undirected NetworkX graph.
(4)             Returns
(4)             -------
(4)             communities : iterable
(8)                 A dict_values object that contains a set of nodes for each community.
(4)             Raises
(4)             ------
(4)             NetworkXNotImplemented
(7)                If the graph is directed
(4)             References
(4)             ----------
(4)             .. [1] Cordasco, G., & Gargano, L. (2010, December). Community detection
(7)                via semi-synchronous label propagation algorithms. In Business
(7)                Applications of Social Network Analysis (BASNA), 2010 IEEE International
(7)                Workshop on (pp. 1-8). IEEE.
(4)             """
(4)             coloring = _color_network(G)
(4)             labeling = {v: k for k, v in enumerate(G)}
(4)             while not _labeling_complete(labeling, G):
(8)                 for color, nodes in coloring.items():
```

```
189: (12)                        for n in nodes:
190: (16)                            _update_label(n, labeling, G)
191: (4)                clusters = defaultdict(set)
192: (4)                for node, label in labeling.items():
193: (8)                    clusters[label].add(node)
194: (4)                return clusters.values()
195: (0)            def _color_network(G):
196: (4)                """Colors the network so that neighboring nodes all have distinct colors.
197: (4)                Returns a dict keyed by color to a set of nodes with that color.
198: (4)                """
199: (4)                coloring = {}  # color => set(node)
200: (4)                colors = nx.coloring.greedy_color(G)
201: (4)                for node, color in colors.items():
202: (8)                    if color in coloring:
203: (12)                       coloring[color].add(node)
204: (8)                    else:
205: (12)                       coloring[color] = {node}
206: (4)                return coloring
207: (0)            def _labeling_complete(labeling, G):
208: (4)                """Determines whether or not LPA is done.
209: (4)                Label propagation is complete when all nodes have a label that is
210: (4)                in the set of highest frequency labels amongst its neighbors.
211: (4)                Nodes with no neighbors are considered complete.
212: (4)                """
213: (4)                return all(
214: (8)                    labeling[v] in _most_frequent_labels(v, labeling, G) for v in G if
len(G[v]) > 0
215: (4)                )
216: (0)            def _most_frequent_labels(node, labeling, G):
217: (4)                """Returns a set of all labels with maximum frequency in `labeling`.
218: (4)                Input `labeling` should be a dict keyed by node to labels.
219: (4)                """
220: (4)                if not G[node]:
221: (8)                    return {labeling[node]}
222: (4)                freqs = Counter(labeling[q] for q in G[node])
223: (4)                max_freq = max(freqs.values())
224: (4)                return {label for label, freq in freqs.items() if freq == max_freq}
225: (0)            def _update_label(node, labeling, G):
226: (4)                """Updates the label of a node using the Prec-Max tie breaking algorithm
227: (4)                The algorithm is explained in: 'Community Detection via Semi-Synchronous
228: (4)                Label Propagation Algorithms' Cordasco and Gargano, 2011
229: (4)                """
230: (4)                high_labels = _most_frequent_labels(node, labeling, G)
231: (4)                if len(high_labels) == 1:
232: (8)                    labeling[node] = high_labels.pop()
233: (4)                elif len(high_labels) > 1:
234: (8)                    if labeling[node] not in high_labels:
235: (12)                       labeling[node] = max(high_labels)
```

File 187 - lukes.py:

```
1: (0)              """Lukes Algorithm for exact optimal weighted tree partitioning."""
2: (0)              from copy import deepcopy
3: (0)              from functools import lru_cache
4: (0)              from random import choice
5: (0)              import networkx as nx
6: (0)              from networkx.utils import not_implemented_for
7: (0)              __all__ = ["lukes_partitioning"]
8: (0)              D_EDGE_W = "weight"
9: (0)              D_EDGE_VALUE = 1.0
10: (0)             D_NODE_W = "weight"
11: (0)             D_NODE_VALUE = 1
12: (0)             PKEY = "partitions"
13: (0)             CLUSTER_EVAL_CACHE_SIZE = 2048
14: (0)             def _split_n_from(n, min_size_of_first_part):
15: (4)                 assert n >= min_size_of_first_part
16: (4)                 for p1 in range(min_size_of_first_part, n + 1):
```

```
(8)                 yield p1, n - p1
(0)         @nx._dispatchable(node_attrs="node_weight", edge_attrs="edge_weight")
(0)         def lukes_partitioning(G, max_size, node_weight=None, edge_weight=None):
(4)             """Optimal partitioning of a weighted tree using the Lukes algorithm.
(4)             This algorithm partitions a connected, acyclic graph featuring integer
(4)             node weights and float edge weights. The resulting clusters are such
(4)             that the total weight of the nodes in each cluster does not exceed
(4)             max_size and that the weight of the edges that are cut by the partition
(4)             is minimum. The algorithm is based on [1]_.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             max_size : int
(8)                 Maximum weight a partition can have in terms of sum of
(8)                 node_weight for all nodes in the partition
(4)             edge_weight : key
(8)                 Edge data key to use as weight. If None, the weights are all
(8)                 set to one.
(4)             node_weight : key
(8)                 Node data key to use as weight. If None, the weights are all
(8)                 set to one. The data must be int.
(4)             Returns
(4)             -------
(4)             partition : list
(8)                 A list of sets of nodes representing the clusters of the
(8)                 partition.
(4)             Raises
(4)             ------
(4)             NotATree
(8)                 If G is not a tree.
(4)             TypeError
(8)                 If any of the values of node_weight is not int.
(4)             References
(4)             ----------
(4)             .. [1] Lukes, J. A. (1974).
(7)                "Efficient Algorithm for the Partitioning of Trees."
(7)                IBM Journal of Research and Development, 18(3), 217–224.
(4)             """
(4)             if not nx.is_tree(G):
(8)                 raise nx.NotATree("lukes_partitioning works only on trees")
(4)             else:
(8)                 if nx.is_directed(G):
(12)                    root = [n for n, d in G.in_degree() if d == 0]
(12)                    assert len(root) == 1
(12)                    root = root[0]
(12)                    t_G = deepcopy(G)
(8)                 else:
(12)                    root = choice(list(G.nodes))
(12)                    t_G = nx.dfs_tree(G, root)
(4)             if edge_weight is None or node_weight is None:
(8)                 safe_G = deepcopy(G)
(8)                 if edge_weight is None:
(12)                    nx.set_edge_attributes(safe_G, D_EDGE_VALUE, D_EDGE_W)
(12)                    edge_weight = D_EDGE_W
(8)                 if node_weight is None:
(12)                    nx.set_node_attributes(safe_G, D_NODE_VALUE, D_NODE_W)
(12)                    node_weight = D_NODE_W
(4)             else:
(8)                 safe_G = G
(4)             all_n_attr = nx.get_node_attributes(safe_G, node_weight).values()
(4)             for x in all_n_attr:
(8)                 if not isinstance(x, int):
(12)                    raise TypeError(
(16)                        "lukes_partitioning needs integer "
(16)                        f"values for node_weight ({node_weight})"
(12)                    )
(4)             @not_implemented_for("undirected")
(4)             def _leaves(gr):
(8)                 for x in gr.nodes:
```

```
86: (12)                        if not nx.descendants(gr, x):
87: (16)                            yield x
88: (4)                 @not_implemented_for("undirected")
89: (4)                 def _a_parent_of_leaves_only(gr):
90: (8)                     tleaves = set(_leaves(gr))
91: (8)                     for n in set(gr.nodes) - tleaves:
92: (12)                        if all(x in tleaves for x in nx.descendants(gr, n)):
93: (16)                            return n
94: (4)                 @lru_cache(CLUSTER_EVAL_CACHE_SIZE)
95: (4)                 def _value_of_cluster(cluster):
96: (8)                     valid_edges = [e for e in safe_G.edges if e[0] in cluster and e[1] in
cluster]
97: (8)                     return sum(safe_G.edges[e][edge_weight] for e in valid_edges)
98: (4)                 def _value_of_partition(partition):
99: (8)                     return sum(_value_of_cluster(frozenset(c)) for c in partition)
100: (4)                @lru_cache(CLUSTER_EVAL_CACHE_SIZE)
101: (4)                def _weight_of_cluster(cluster):
102: (8)                    return sum(safe_G.nodes[n][node_weight] for n in cluster)
103: (4)                def _pivot(partition, node):
104: (8)                    ccx = [c for c in partition if node in c]
105: (8)                    assert len(ccx) == 1
106: (8)                    return ccx[0]
107: (4)                def _concatenate_or_merge(partition_1, partition_2, x, i, ref_weight):
108: (8)                    ccx = _pivot(partition_1, x)
109: (8)                    cci = _pivot(partition_2, i)
110: (8)                    merged_xi = ccx.union(cci)
111: (8)                    if _weight_of_cluster(frozenset(merged_xi)) <= ref_weight:
112: (12)                       cp1 = list(filter(lambda x: x != ccx, partition_1))
113: (12)                       cp2 = list(filter(lambda x: x != cci, partition_2))
114: (12)                       option_2 = [merged_xi] + cp1 + cp2
115: (12)                       return option_2, _value_of_partition(option_2)
116: (8)                    else:
117: (12)                       option_1 = partition_1 + partition_2
118: (12)                       return option_1, _value_of_partition(option_1)
119: (4)                leaves = set(_leaves(t_G))
120: (4)                for lv in leaves:
121: (8)                    t_G.nodes[lv][PKEY] = {}
122: (8)                    slot = safe_G.nodes[lv][node_weight]
123: (8)                    t_G.nodes[lv][PKEY][slot] = [{lv}]
124: (8)                    t_G.nodes[lv][PKEY][0] = [{lv}]
125: (4)                for inner in [x for x in t_G.nodes if x not in leaves]:
126: (8)                    t_G.nodes[inner][PKEY] = {}
127: (8)                    slot = safe_G.nodes[inner][node_weight]
128: (8)                    t_G.nodes[inner][PKEY][slot] = [{inner}]
129: (4)                nx._clear_cache(t_G)
130: (4)                while True:
131: (8)                    x_node = _a_parent_of_leaves_only(t_G)
132: (8)                    weight_of_x = safe_G.nodes[x_node][node_weight]
133: (8)                    best_value = 0
134: (8)                    best_partition = None
135: (8)                    bp_buffer = {}
136: (8)                    x_descendants = nx.descendants(t_G, x_node)
137: (8)                    for i_node in x_descendants:
138: (12)                       for j in range(weight_of_x, max_size + 1):
139: (16)                           for a, b in _split_n_from(j, weight_of_x):
140: (20)                               if (
141: (24)                                   a not in t_G.nodes[x_node][PKEY]
142: (24)                                   or b not in t_G.nodes[i_node][PKEY]
143: (20)                               ):
144: (24)                                   continue
145: (20)                               part1 = t_G.nodes[x_node][PKEY][a]
146: (20)                               part2 = t_G.nodes[i_node][PKEY][b]
147: (20)                               part, value = _concatenate_or_merge(part1, part2, x_node,
i_node, j)
148: (20)                               if j not in bp_buffer or bp_buffer[j][1] < value:
149: (24)                                   bp_buffer[j] = part, value
150: (20)                               if best_value <= value:
151: (24)                                   best_value = value
152: (24)                                   best_partition = part
```

```
153: (12)                        for w, (best_part_for_vl, vl) in bp_buffer.items():
154: (16)                            t_G.nodes[x_node][PKEY][w] = best_part_for_vl
155: (12)                        bp_buffer.clear()
156: (8)                    t_G.nodes[x_node][PKEY][0] = best_partition
157: (8)                    t_G.remove_nodes_from(x_descendants)
158: (8)                    if x_node == root:
159: (12)                        return t_G.nodes[root][PKEY][0]

----------------------------------------

File 188 - louvain.py:

1: (0)              """Function for detecting communities based on Louvain Community Detection
2: (0)              Algorithm"""
3: (0)              import itertools
4: (0)              from collections import defaultdict, deque
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.community import modularity
7: (0)              from networkx.utils import py_random_state
8: (0)              __all__ = ["louvain_communities", "louvain_partitions"]
9: (0)              @py_random_state("seed")
10: (0)             @nx._dispatchable(edge_attrs="weight")
11: (0)             def louvain_communities(
12: (4)                 G, weight="weight", resolution=1, threshold=0.0000001, max_level=None,
seed=None
13: (0)             ):
14: (4)                 r"""Find the best partition of a graph using the Louvain Community
Detection
15: (4)                 Algorithm.
16: (4)                 Louvain Community Detection Algorithm is a simple method to extract the
community
17: (4)                 structure of a network. This is a heuristic method based on modularity
optimization. [1]_
18: (4)                 The algorithm works in 2 steps. On the first step it assigns every node to
be
19: (4)                 in its own community and then for each node it tries to find the maximum
positive
20: (4)                 modularity gain by moving each node to all of its neighbor communities. If
no positive
21: (4)                 gain is achieved the node remains in its original community.
22: (4)                 The modularity gain obtained by moving an isolated node $i$ into a
community $C$ can
23: (4)                 easily be calculated by the following formula (combining [1]_ [2]_ and
some algebra):
24: (4)                 .. math::
25: (8)                     \Delta Q = \frac{k_{i,in}}{2m} - \gamma\frac{ \Sigma_{tot} \cdot k_i}
{2m^2}
26: (4)                 where $m$ is the size of the graph, $k_{i,in}$ is the sum of the weights
of the links
27: (4)                 from $i$ to nodes in $C$, $k_i$ is the sum of the weights of the links
incident to node $i$,
28: (4)                 $\Sigma_{tot}$ is the sum of the weights of the links incident to nodes in
$C$ and $\gamma$
29: (4)                 is the resolution parameter.
30: (4)                 For the directed case the modularity gain can be computed using this
formula according to [3]_
31: (4)                 .. math::
32: (8)                     \Delta Q = \frac{k_{i,in}}{m}
33: (8)                     - \gamma\frac{k_i^{out} \cdot\Sigma_{tot}^{in} + k_i^{in} \cdot
\Sigma_{tot}^{out}}{m^2}
34: (4)                 where $k_i^{out}$, $k_i^{in}$ are the outer and inner weighted degrees of
node $i$ and
35: (4)                 $\Sigma_{tot}^{in}$, $\Sigma_{tot}^{out}$ are the sum of in-going and out-
going links incident
36: (4)                 to nodes in $C$.
37: (4)                 The first phase continues until no individual move can improve the
modularity.
38: (4)                 The second phase consists in building a new network whose nodes are now
the communities
```

```
39: (4)                 found in the first phase. To do so, the weights of the links between the
new nodes are given by
40: (4)                 the sum of the weight of the links between nodes in the corresponding two
communities. Once this
41: (4)                 phase is complete it is possible to reapply the first phase creating
bigger communities with
42: (4)                 increased modularity.
43: (4)                 The above two phases are executed until no modularity gain is achieved (or
is less than
44: (4)                 the `threshold`, or until `max_levels` is reached).
45: (4)                 Be careful with self-loops in the input graph. These are treated as
46: (4)                 previously reduced communities -- as if the process had been started
47: (4)                 in the middle of the algorithm. Large self-loop edge weights thus
48: (4)                 represent strong communities and in practice may be hard to add
49: (4)                 other nodes to.  If your input graph edge weights for self-loops
50: (4)                 do not represent already reduced communities you may want to remove
51: (4)                 the self-loops before inputting that graph.
52: (4)                 Parameters
53: (4)                 ----------
54: (4)                 G : NetworkX graph
55: (4)                 weight : string or None, optional (default="weight")
56: (8)                     The name of an edge attribute that holds the numerical value
57: (8)                     used as a weight. If None then each edge has weight 1.
58: (4)                 resolution : float, optional (default=1)
59: (8)                     If resolution is less than 1, the algorithm favors larger communities.
60: (8)                     Greater than 1 favors smaller communities
61: (4)                 threshold : float, optional (default=0.0000001)
62: (8)                     Modularity gain threshold for each level. If the gain of modularity
63: (8)                     between 2 levels of the algorithm is less than the given threshold
64: (8)                     then the algorithm stops and returns the resulting communities.
65: (4)                 max_level : int or None, optional (default=None)
66: (8)                     The maximum number of levels (steps of the algorithm) to compute.
67: (8)                     Must be a positive integer or None. If None, then there is no max
68: (8)                     level and the threshold parameter determines the stopping condition.
69: (4)                 seed : integer, random_state, or None (default)
70: (8)                     Indicator of random number generation state.
71: (8)                     See :ref:`Randomness<randomness>`.
72: (4)                 Returns
73: (4)                 -------
74: (4)                 list
75: (8)                     A list of sets (partition of `G`). Each set represents one community
and contains
76: (8)                     all the nodes that constitute it.
77: (4)                 Examples
78: (4)                 --------
79: (4)                 >>> import networkx as nx
80: (4)                 >>> G = nx.petersen_graph()
81: (4)                 >>> nx.community.louvain_communities(G, seed=123)
82: (4)                 [{0, 4, 5, 7, 9}, {1, 2, 3, 6, 8}]
83: (4)                 Notes
84: (4)                 -----
85: (4)                 The order in which the nodes are considered can affect the final output.
In the algorithm
86: (4)                 the ordering happens using a random shuffle.
87: (4)                 References
88: (4)                 ----------
89: (4)                 .. [1] Blondel, V.D. et al. Fast unfolding of communities in
90: (7)                    large networks. J. Stat. Mech 10008, 1-12(2008).
https://doi.org/10.1088/1742-5468/2008/10/P10008
91: (4)                 .. [2] Traag, V.A., Waltman, L. & van Eck, N.J. From Louvain to Leiden:
guaranteeing
92: (7)                    well-connected communities. Sci Rep 9, 5233 (2019).
https://doi.org/10.1038/s41598-019-41695-z
93: (4)                 .. [3] Nicolas Dugué, Anthony Perez. Directed Louvain : maximizing
modularity in directed networks.
94: (8)                     [Research Report] Université d'Orléans. 2015. hal-01231784.
https://hal.archives-ouvertes.fr/hal-01231784
95: (4)                 See Also
96: (4)                 --------
```

```
97: (4)                 louvain_partitions
98: (4)                 """
99: (4)                 partitions = louvain_partitions(G, weight, resolution, threshold, seed)
100: (4)                if max_level is not None:
101: (8)                    if max_level <= 0:
102: (12)                       raise ValueError("max_level argument must be a positive integer or
None")
103: (8)                    partitions = itertools.islice(partitions, max_level)
104: (4)                final_partition = deque(partitions, maxlen=1)
105: (4)                return final_partition.pop()
106: (0)            @py_random_state("seed")
107: (0)            @nx._dispatchable(edge_attrs="weight")
108: (0)            def louvain_partitions(
109: (4)                G, weight="weight", resolution=1, threshold=0.0000001, seed=None
110: (0)            ):
111: (4)                """Yields partitions for each level of the Louvain Community Detection
Algorithm
112: (4)                Louvain Community Detection Algorithm is a simple method to extract the
community
113: (4)                structure of a network. This is a heuristic method based on modularity
optimization. [1]_
114: (4)                The partitions at each level (step of the algorithm) form a dendrogram of
communities.
115: (4)                A dendrogram is a diagram representing a tree and each level represents
116: (4)                a partition of the G graph. The top level contains the smallest
communities
117: (4)                and as you traverse to the bottom of the tree the communities get bigger
118: (4)                and the overall modularity increases making the partition better.
119: (4)                Each level is generated by executing the two phases of the Louvain
Community
120: (4)                Detection Algorithm.
121: (4)                Be careful with self-loops in the input graph. These are treated as
122: (4)                previously reduced communities -- as if the process had been started
123: (4)                in the middle of the algorithm. Large self-loop edge weights thus
124: (4)                represent strong communities and in practice may be hard to add
125: (4)                other nodes to.  If your input graph edge weights for self-loops
126: (4)                do not represent already reduced communities you may want to remove
127: (4)                the self-loops before inputting that graph.
128: (4)                Parameters
129: (4)                ----------
130: (4)                G : NetworkX graph
131: (4)                weight : string or None, optional (default="weight")
132: (5)                 The name of an edge attribute that holds the numerical value
133: (5)                 used as a weight. If None then each edge has weight 1.
134: (4)                resolution : float, optional (default=1)
135: (8)                    If resolution is less than 1, the algorithm favors larger communities.
136: (8)                    Greater than 1 favors smaller communities
137: (4)                threshold : float, optional (default=0.0000001)
138: (5)                 Modularity gain threshold for each level. If the gain of modularity
139: (5)                 between 2 levels of the algorithm is less than the given threshold
140: (5)                 then the algorithm stops and returns the resulting communities.
141: (4)                seed : integer, random_state, or None (default)
142: (5)                 Indicator of random number generation state.
143: (5)                 See :ref:`Randomness<randomness>`.
144: (4)                Yields
145: (4)                ------
146: (4)                list
147: (8)                    A list of sets (partition of `G`). Each set represents one community
and contains
148: (8)                    all the nodes that constitute it.
149: (4)                References
150: (4)                ----------
151: (4)                .. [1] Blondel, V.D. et al. Fast unfolding of communities in
152: (7)                   large networks. J. Stat. Mech 10008, 1-12(2008)
153: (4)                See Also
154: (4)                --------
155: (4)                louvain_communities
156: (4)                """
157: (4)                partition = [{u} for u in G.nodes()]
```

```
158: (4)                if nx.is_empty(G):
159: (8)                    yield partition
160: (8)                    return
161: (4)                mod = modularity(G, partition, resolution=resolution, weight=weight)
162: (4)                is_directed = G.is_directed()
163: (4)                if G.is_multigraph():
164: (8)                    graph = _convert_multigraph(G, weight, is_directed)
165: (4)                else:
166: (8)                    graph = G.__class__()
167: (8)                    graph.add_nodes_from(G)
168: (8)                    graph.add_weighted_edges_from(G.edges(data=weight, default=1))
169: (4)                m = graph.size(weight="weight")
170: (4)                partition, inner_partition, improvement = _one_level(
171: (8)                    graph, m, partition, resolution, is_directed, seed
172: (4)                )
173: (4)                improvement = True
174: (4)                while improvement:
175: (8)                    yield [s.copy() for s in partition]
176: (8)                    new_mod = modularity(
177: (12)                       graph, inner_partition, resolution=resolution, weight="weight"
178: (8)                    )
179: (8)                    if new_mod - mod <= threshold:
180: (12)                       return
181: (8)                    mod = new_mod
182: (8)                    graph = _gen_graph(graph, inner_partition)
183: (8)                    partition, inner_partition, improvement = _one_level(
184: (12)                       graph, m, partition, resolution, is_directed, seed
185: (8)                    )
186: (0)            def _one_level(G, m, partition, resolution=1, is_directed=False, seed=None):
187: (4)                """Calculate one level of the Louvain partitions tree
188: (4)                Parameters
189: (4)                ----------
190: (4)                G : NetworkX Graph/DiGraph
191: (8)                    The graph from which to detect communities
192: (4)                m : number
193: (8)                    The size of the graph `G`.
194: (4)                partition : list of sets of nodes
195: (8)                    A valid partition of the graph `G`
196: (4)                resolution : positive number
197: (8)                    The resolution parameter for computing the modularity of a partition
198: (4)                is_directed : bool
199: (8)                    True if `G` is a directed graph.
200: (4)                seed : integer, random_state, or None (default)
201: (8)                    Indicator of random number generation state.
202: (8)                    See :ref:`Randomness<randomness>`.
203: (4)                """
204: (4)                node2com = {u: i for i, u in enumerate(G.nodes())}
205: (4)                inner_partition = [{u} for u in G.nodes()]
206: (4)                if is_directed:
207: (8)                    in_degrees = dict(G.in_degree(weight="weight"))
208: (8)                    out_degrees = dict(G.out_degree(weight="weight"))
209: (8)                    Stot_in = list(in_degrees.values())
210: (8)                    Stot_out = list(out_degrees.values())
211: (8)                    nbrs = {}
212: (8)                    for u in G:
213: (12)                       nbrs[u] = defaultdict(float)
214: (12)                       for _, n, wt in G.out_edges(u, data="weight"):
215: (16)                           if u != n:
216: (20)                               nbrs[u][n] += wt
217: (12)                       for n, _, wt in G.in_edges(u, data="weight"):
218: (16)                           if u != n:
219: (20)                               nbrs[u][n] += wt
220: (4)                else:
221: (8)                    degrees = dict(G.degree(weight="weight"))
222: (8)                    Stot = list(degrees.values())
223: (8)                    nbrs = {u: {v: data["weight"] for v, data in G[u].items() if v != u}
for u in G}
224: (4)                rand_nodes = list(G.nodes)
225: (4)                seed.shuffle(rand_nodes)
```

```
226: (4)                nb_moves = 1
227: (4)                improvement = False
228: (4)                while nb_moves > 0:
229: (8)                    nb_moves = 0
230: (8)                    for u in rand_nodes:
231: (12)                       best_mod = 0
232: (12)                       best_com = node2com[u]
233: (12)                       weights2com = _neighbor_weights(nbrs[u], node2com)
234: (12)                       if is_directed:
235: (16)                           in_degree = in_degrees[u]
236: (16)                           out_degree = out_degrees[u]
237: (16)                           Stot_in[best_com] -= in_degree
238: (16)                           Stot_out[best_com] -= out_degree
239: (16)                           remove_cost = (
240: (20)                               -weights2com[best_com] / m
241: (20)                               + resolution
242: (20)                               * (out_degree * Stot_in[best_com] + in_degree *
Stot_out[best_com])
243: (20)                               / m**2
244: (16)                           )
245: (12)                       else:
246: (16)                           degree = degrees[u]
247: (16)                           Stot[best_com] -= degree
248: (16)                           remove_cost = -weights2com[best_com] / m + resolution * (
249: (20)                               Stot[best_com] * degree
250: (16)                           ) / (2 * m**2)
251: (12)                       for nbr_com, wt in weights2com.items():
252: (16)                           if is_directed:
253: (20)                               gain = (
254: (24)                                   remove_cost
255: (24)                                   + wt / m
256: (24)                                   - resolution
257: (24)                                   * (
258: (28)                                       out_degree * Stot_in[nbr_com]
259: (28)                                       + in_degree * Stot_out[nbr_com]
260: (24)                                   )
261: (24)                                   / m**2
262: (20)                               )
263: (16)                           else:
264: (20)                               gain = (
265: (24)                                   remove_cost
266: (24)                                   + wt / m
267: (24)                                   - resolution * (Stot[nbr_com] * degree) / (2 * m**2)
268: (20)                               )
269: (16)                           if gain > best_mod:
270: (20)                               best_mod = gain
271: (20)                               best_com = nbr_com
272: (12)                       if is_directed:
273: (16)                           Stot_in[best_com] += in_degree
274: (16)                           Stot_out[best_com] += out_degree
275: (12)                       else:
276: (16)                           Stot[best_com] += degree
277: (12)                       if best_com != node2com[u]:
278: (16)                           com = G.nodes[u].get("nodes", {u})
279: (16)                           partition[node2com[u]].difference_update(com)
280: (16)                           inner_partition[node2com[u]].remove(u)
281: (16)                           partition[best_com].update(com)
282: (16)                           inner_partition[best_com].add(u)
283: (16)                           improvement = True
284: (16)                           nb_moves += 1
285: (16)                           node2com[u] = best_com
286: (4)                partition = list(filter(len, partition))
287: (4)                inner_partition = list(filter(len, inner_partition))
288: (4)                return partition, inner_partition, improvement
289: (0)            def _neighbor_weights(nbrs, node2com):
290: (4)                """Calculate weights between node and its neighbor communities.
291: (4)                Parameters
292: (4)                ----------
293: (4)                nbrs : dictionary
```

```
294: (11)                       Dictionary with nodes' neighbors as keys and their edge weight as
value.
295: (4)                node2com : dictionary
296: (11)                       Dictionary with all graph's nodes as keys and their community index
as value.
297: (4)                """
298: (4)                weights = defaultdict(float)
299: (4)                for nbr, wt in nbrs.items():
300: (8)                    weights[node2com[nbr]] += wt
301: (4)                return weights
302: (0)            def _gen_graph(G, partition):
303: (4)                """Generate a new graph based on the partitions of a given graph"""
304: (4)                H = G.__class__()
305: (4)                node2com = {}
306: (4)                for i, part in enumerate(partition):
307: (8)                    nodes = set()
308: (8)                    for node in part:
309: (12)                       node2com[node] = i
310: (12)                       nodes.update(G.nodes[node].get("nodes", {node}))
311: (8)                    H.add_node(i, nodes=nodes)
312: (4)                for node1, node2, wt in G.edges(data=True):
313: (8)                    wt = wt["weight"]
314: (8)                    com1 = node2com[node1]
315: (8)                    com2 = node2com[node2]
316: (8)                    temp = H.get_edge_data(com1, com2, {"weight": 0})["weight"]
317: (8)                    H.add_edge(com1, com2, weight=wt + temp)
318: (4)                return H
319: (0)            def _convert_multigraph(G, weight, is_directed):
320: (4)                """Convert a Multigraph to normal Graph"""
321: (4)                if is_directed:
322: (8)                    H = nx.DiGraph()
323: (4)                else:
324: (8)                    H = nx.Graph()
325: (4)                H.add_nodes_from(G)
326: (4)                for u, v, wt in G.edges(data=weight, default=1):
327: (8)                    if H.has_edge(u, v):
328: (12)                       H[u][v]["weight"] += wt
329: (8)                    else:
330: (12)                       H.add_edge(u, v, weight=wt)
331: (4)                return H

----------------------------------------

File 189 - quality.py:

1: (0)              """Functions for measuring the quality of a partition (into
2: (0)              communities).
3: (0)              """
4: (0)              from itertools import combinations
5: (0)              import networkx as nx
6: (0)              from networkx import NetworkXError
7: (0)              from networkx.algorithms.community.community_utils import is_partition
8: (0)              from networkx.utils.decorators import argmap
9: (0)              __all__ = ["modularity", "partition_quality"]
10: (0)             class NotAPartition(NetworkXError):
11: (4)                 """Raised if a given collection is not a partition."""
12: (4)                 def __init__(self, G, collection):
13: (8)                     msg = f"{collection} is not a valid partition of the graph {G}"
14: (8)                     super().__init__(msg)
15: (0)             def _require_partition(G, partition):
16: (4)                 """Decorator to check that a valid partition is input to a function
17: (4)                 Raises :exc:`networkx.NetworkXError` if the partition is not valid.
18: (4)                 This decorator should be used on functions whose first two arguments
19: (4)                 are a graph and a partition of the nodes of that graph (in that
20: (4)                 order)::
21: (8)                     >>> @require_partition
22: (8)                     ... def foo(G, partition):
23: (8)                     ...     print("partition is valid!")
24: (8)                     ...
```

```
25: (8)                     >>> G = nx.complete_graph(5)
26: (8)                     >>> partition = [{0, 1}, {2, 3}, {4}]
27: (8)                     >>> foo(G, partition)
28: (8)                     partition is valid!
29: (8)                     >>> partition = [{0}, {2, 3}, {4}]
30: (8)                     >>> foo(G, partition)
31: (8)                     Traceback (most recent call last):
32: (10)                      ...
33: (8)                     networkx.exception.NetworkXError: `partition` is not a valid partition
of the nodes of G
34: (8)                     >>> partition = [{0, 1}, {1, 2, 3}, {4}]
35: (8)                     >>> foo(G, partition)
36: (8)                     Traceback (most recent call last):
37: (10)                      ...
38: (8)                     networkx.exception.NetworkXError: `partition` is not a valid partition
of the nodes of G
39: (4)                 """
40: (4)                 if is_partition(G, partition):
41: (8)                     return G, partition
42: (4)                 raise nx.NetworkXError("`partition` is not a valid partition of the nodes
of G")
43: (0)             require_partition = argmap(_require_partition, (0, 1))
44: (0)             @nx._dispatchable
45: (0)             def intra_community_edges(G, partition):
46: (4)                 """Returns the number of intra-community edges for a partition of `G`.
47: (4)                 Parameters
48: (4)                 ----------
49: (4)                 G : NetworkX graph.
50: (4)                 partition : iterable of sets of nodes
51: (8)                     This must be a partition of the nodes of `G`.
52: (4)                 The "intra-community edges" are those edges joining a pair of nodes
53: (4)                 in the same block of the partition.
54: (4)                 """
55: (4)                 return sum(G.subgraph(block).size() for block in partition)
56: (0)             @nx._dispatchable
57: (0)             def inter_community_edges(G, partition):
58: (4)                 """Returns the number of inter-community edges for a partition of `G`.
59: (4)                 according to the given
60: (4)                 partition of the nodes of `G`.
61: (4)                 Parameters
62: (4)                 ----------
63: (4)                 G : NetworkX graph.
64: (4)                 partition : iterable of sets of nodes
65: (8)                     This must be a partition of the nodes of `G`.
66: (4)                 The *inter-community edges* are those edges joining a pair of nodes
67: (4)                 in different blocks of the partition.
68: (4)                 Implementation note: this function creates an intermediate graph
69: (4)                 that may require the same amount of memory as that of `G`.
70: (4)                 """
71: (4)                 MG = nx.MultiDiGraph if G.is_directed() else nx.MultiGraph
72: (4)                 return nx.quotient_graph(G, partition, create_using=MG).size()
73: (0)             @nx._dispatchable
74: (0)             def inter_community_non_edges(G, partition):
75: (4)                 """Returns the number of inter-community non-edges according to the
76: (4)                 given partition of the nodes of `G`.
77: (4)                 Parameters
78: (4)                 ----------
79: (4)                 G : NetworkX graph.
80: (4)                 partition : iterable of sets of nodes
81: (8)                     This must be a partition of the nodes of `G`.
82: (4)                 A *non-edge* is a pair of nodes (undirected if `G` is undirected)
83: (4)                 that are not adjacent in `G`. The *inter-community non-edges* are
84: (4)                 those non-edges on a pair of nodes in different blocks of the
85: (4)                 partition.
86: (4)                 Implementation note: this function creates two intermediate graphs,
87: (4)                 which may require up to twice the amount of memory as required to
88: (4)                 store `G`.
89: (4)                 """
90: (4)                 return inter_community_edges(nx.complement(G), partition)
```

```
@nx._dispatchable(edge_attrs="weight")
def modularity(G, communities, weight="weight", resolution=1):
    r"""Returns the modularity of the given partition of the graph.
    Modularity is defined in [1]_ as
    .. math::
        Q = \frac{1}{2m} \sum_{ij} \left( A_{ij} - \gamma\frac{k_ik_j}{2m}\right)
            \delta(c_i,c_j)
    where $m$ is the number of edges (or sum of all edge weights as in [5]_),
    $A$ is the adjacency matrix of `G`, $k_i$ is the (weighted) degree of $i$,
    $\gamma$ is the resolution parameter, and $\delta(c_i, c_j)$ is 1 if $i$ and
    $j$ are in the same community else 0.
    According to [2]_ (and verified by some algebra) this can be reduced to
    .. math::
       Q = \sum_{c=1}^{n}
       \left[ \frac{L_c}{m} - \gamma\left( \frac{k_c}{2m} \right) ^2 \right]
    where the sum iterates over all communities $c$, $m$ is the number of edges,
    $L_c$ is the number of intra-community links for community $c$,
    $k_c$ is the sum of degrees of the nodes in community $c$,
    and $\gamma$ is the resolution parameter.
    The resolution parameter sets an arbitrary tradeoff between intra-group
    edges and inter-group edges. More complex grouping patterns can be
    discovered by analyzing the same network with multiple values of gamma
    and then combining the results [3]_. That said, it is very common to
    simply use gamma=1. More on the choice of gamma is in [4]_.
    The second formula is the one actually used in calculation of the modularity.
    For directed graphs the second formula replaces $k_c$ with $k^{in}_c k^{out}_c$.
    Parameters
    ----------
    G : NetworkX Graph
    communities : list or iterable of set of nodes
        These node sets must represent a partition of G's nodes.
    weight : string or None, optional (default="weight")
        The edge attribute that holds the numerical value used
        as a weight. If None or an edge does not have that attribute,
        then that edge has weight 1.
    resolution : float (default=1)
        If resolution is less than 1, modularity favors larger communities.
        Greater than 1 favors smaller communities.
    Returns
    -------
    Q : float
        The modularity of the partition.
    Raises
    ------
    NotAPartition
        If `communities` is not a partition of the nodes of `G`.
    Examples
    --------
    >>> G = nx.barbell_graph(3, 0)
    >>> nx.community.modularity(G, [{0, 1, 2}, {3, 4, 5}])
    0.35714285714285715
    >>> nx.community.modularity(G, nx.community.label_propagation_communities(G))
    0.35714285714285715
    References
    ----------
    .. [1] M. E. J. Newman "Networks: An Introduction", page 224.
       Oxford University Press, 2011.
    .. [2] Clauset, Aaron, Mark EJ Newman, and Cristopher Moore.
       "Finding community structure in very large networks."
       Phys. Rev. E 70.6 (2004). <https://arxiv.org/abs/cond-mat/0408187>
    .. [3] Reichardt and Bornholdt "Statistical Mechanics of Community Detection"
       Phys. Rev. E 74, 016110, 2006.
```

```
https://doi.org/10.1103/PhysRevE.74.016110
153: (4)                .. [4] M. E. J. Newman, "Equivalence between modularity optimization and
154: (7)                   maximum likelihood methods for community detection"
155: (7)                   Phys. Rev. E 94, 052315, 2016.
https://doi.org/10.1103/PhysRevE.94.052315
156: (4)                .. [5] Blondel, V.D. et al. "Fast unfolding of communities in large
157: (7)                   networks" J. Stat. Mech 10008, 1-12 (2008).
158: (7)                   https://doi.org/10.1088/1742-5468/2008/10/P10008
159: (4)                """
160: (4)                if not isinstance(communities, list):
161: (8)                    communities = list(communities)
162: (4)                if not is_partition(G, communities):
163: (8)                    raise NotAPartition(G, communities)
164: (4)                directed = G.is_directed()
165: (4)                if directed:
166: (8)                    out_degree = dict(G.out_degree(weight=weight))
167: (8)                    in_degree = dict(G.in_degree(weight=weight))
168: (8)                    m = sum(out_degree.values())
169: (8)                    norm = 1 / m**2
170: (4)                else:
171: (8)                    out_degree = in_degree = dict(G.degree(weight=weight))
172: (8)                    deg_sum = sum(out_degree.values())
173: (8)                    m = deg_sum / 2
174: (8)                    norm = 1 / deg_sum**2
175: (4)                def community_contribution(community):
176: (8)                    comm = set(community)
177: (8)                    L_c = sum(wt for u, v, wt in G.edges(comm, data=weight, default=1) if
v in comm)
178: (8)                    out_degree_sum = sum(out_degree[u] for u in comm)
179: (8)                    in_degree_sum = sum(in_degree[u] for u in comm) if directed else
out_degree_sum
180: (8)                    return L_c / m - resolution * out_degree_sum * in_degree_sum * norm
181: (4)                return sum(map(community_contribution, communities))
182: (0)            @require_partition
183: (0)            @nx._dispatchable
184: (0)            def partition_quality(G, partition):
185: (4)                """Returns the coverage and performance of a partition of G.
186: (4)                The *coverage* of a partition is the ratio of the number of
187: (4)                intra-community edges to the total number of edges in the graph.
188: (4)                The *performance* of a partition is the number of
189: (4)                intra-community edges plus inter-community non-edges divided by the total
190: (4)                number of potential edges.
191: (4)                This algorithm has complexity $O(C^2 + L)$ where C is the number of
communities and L is the number of links.
192: (4)                Parameters
193: (4)                ----------
194: (4)                G : NetworkX graph
195: (4)                partition : sequence
196: (8)                    Partition of the nodes of `G`, represented as a sequence of
197: (8)                    sets of nodes (blocks). Each block of the partition represents a
198: (8)                    community.
199: (4)                Returns
200: (4)                -------
201: (4)                (float, float)
202: (8)                    The (coverage, performance) tuple of the partition, as defined above.
203: (4)                Raises
204: (4)                ------
205: (4)                NetworkXError
206: (8)                    If `partition` is not a valid partition of the nodes of `G`.
207: (4)                Notes
208: (4)                -----
209: (4)                If `G` is a multigraph;
210: (8)                    - for coverage, the multiplicity of edges is counted
211: (8)                    - for performance, the result is -1 (total number of possible edges is
not defined)
212: (4)                References
213: (4)                ----------
214: (4)                .. [1] Santo Fortunato.
215: (11)                      "Community Detection in Graphs".
```

```
216: (11)                       *Physical Reports*, Volume 486, Issue 3--5 pp. 75--174
217: (11)                       <https://arxiv.org/abs/0906.0612>
218: (4)                """
219: (4)                node_community = {}
220: (4)                for i, community in enumerate(partition):
221: (8)                    for node in community:
222: (12)                       node_community[node] = i
223: (4)                if not G.is_multigraph():
224: (8)                    possible_inter_community_edges = sum(
225: (12)                       len(p1) * len(p2) for p1, p2 in combinations(partition, 2)
226: (8)                    )
227: (8)                    if G.is_directed():
228: (12)                       possible_inter_community_edges *= 2
229: (4)                else:
230: (8)                    possible_inter_community_edges = 0
231: (4)                n = len(G)
232: (4)                total_pairs = n * (n - 1)
233: (4)                if not G.is_directed():
234: (8)                    total_pairs //= 2
235: (4)                intra_community_edges = 0
236: (4)                inter_community_non_edges = possible_inter_community_edges
237: (4)                for e in G.edges():
238: (8)                    if node_community[e[0]] == node_community[e[1]]:
239: (12)                       intra_community_edges += 1
240: (8)                    else:
241: (12)                       inter_community_non_edges -= 1
242: (4)                coverage = intra_community_edges / len(G.edges)
243: (4)                if G.is_multigraph():
244: (8)                    performance = -1.0
245: (4)                else:
246: (8)                    performance = (intra_community_edges + inter_community_non_edges) /
total_pairs
247: (4)                return coverage, performance

----------------------------------------

File 190 - modularity_max.py:

1: (0)              """Functions for detecting communities based on modularity."""
2: (0)              from collections import defaultdict
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.community.quality import modularity
5: (0)              from networkx.utils import not_implemented_for
6: (0)              from networkx.utils.mapped_queue import MappedQueue
7: (0)              __all__ = [
8: (4)                  "greedy_modularity_communities",
9: (4)                  "naive_greedy_modularity_communities",
10: (0)             ]
11: (0)             def _greedy_modularity_communities_generator(G, weight=None, resolution=1):
12: (4)                 r"""Yield community partitions of G and the modularity change at each
step.
13: (4)                 This function performs Clauset-Newman-Moore greedy modularity maximization
[2]_
14: (4)                 At each step of the process it yields the change in modularity that will
occur in
15: (4)                 the next step followed by yielding the new community partition after that
step.
16: (4)                 Greedy modularity maximization begins with each node in its own community
17: (4)                 and repeatedly joins the pair of communities that lead to the largest
18: (4)                 modularity until one community contains all nodes (the partition has one
set).
19: (4)                 This function maximizes the generalized modularity, where `resolution`
20: (4)                 is the resolution parameter, often expressed as $\gamma$.
21: (4)                 See :func:`~networkx.algorithms.community.quality.modularity`.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 G : NetworkX graph
25: (4)                 weight : string or None, optional (default=None)
26: (8)                     The name of an edge attribute that holds the numerical value used
```

```
(8)                     as a weight.  If None, then each edge has weight 1.
(8)                     The degree is the sum of the edge weights adjacent to the node.
(4)                 resolution : float (default=1)
(8)                     If resolution is less than 1, modularity favors larger communities.
(8)                     Greater than 1 favors smaller communities.
(4)                 Yields
(4)                 ------
(4)                 Alternating yield statements produce the following two objects:
(4)                 communities: dict_values
(8)                     A dict_values of frozensets of nodes, one for each community.
(8)                     This represents a partition of the nodes of the graph into communities.
(8)                     The first yield is the partition with each node in its own community.
(4)                 dq: float
(8)                     The change in modularity when merging the next two communities
(8)                     that leads to the largest modularity.
(4)                 See Also
(4)                 --------
(4)                 modularity
(4)                 References
(4)                 ----------
(4)                 .. [1] Newman, M. E. J. "Networks: An Introduction", page 224
(7)                    Oxford University Press 2011.
(4)                 .. [2] Clauset, A., Newman, M. E., & Moore, C.
(7)                    "Finding community structure in very large networks."
(7)                    Physical Review E 70(6), 2004.
(4)                 .. [3] Reichardt and Bornholdt "Statistical Mechanics of Community
(7)                    Detection" Phys. Rev. E74, 2006.
(4)                 .. [4] Newman, M. E. J."Analysis of weighted networks"
(7)                    Physical Review E 70(5 Pt 2):056131, 2004.
(4)                 """
(4)                 directed = G.is_directed()
(4)                 N = G.number_of_nodes()
(4)                 m = G.size(weight)
(4)                 q0 = 1 / m
(4)                 if directed:
(8)                     a = {node: deg_out * q0 for node, deg_out in G.out_degree(weight=weight)}
(8)                     b = {node: deg_in * q0 for node, deg_in in G.in_degree(weight=weight)}
(4)                 else:
(8)                     a = b = {node: deg * q0 * 0.5 for node, deg in G.degree(weight=weight)}
(4)                 dq_dict = defaultdict(lambda: defaultdict(float))
(4)                 for u, v, wt in G.edges(data=weight, default=1):
(8)                     if u == v:
(12)                        continue
(8)                     dq_dict[u][v] += wt
(8)                     dq_dict[v][u] += wt
(4)                 for u, nbrdict in dq_dict.items():
(8)                     for v, wt in nbrdict.items():
(12)                        dq_dict[u][v] = q0 * wt - resolution * (a[u] * b[v] + b[u] * a[v])
(4)                 dq_heap = {u: MappedQueue({(u, v): -dq for v, dq in dq_dict[u].items()}) for u in G}
(4)                 H = MappedQueue([dq_heap[n].heap[0] for n in G if len(dq_heap[n]) > 0])
(4)                 communities = {n: frozenset([n]) for n in G}
(4)                 yield communities.values()
(4)                 while len(H) > 1:
(8)                     try:
(12)                        negdq, u, v = H.pop()
(8)                     except IndexError:
(12)                        break
(8)                     dq = -negdq
(8)                     yield dq
(8)                     dq_heap[u].pop()
(8)                     if len(dq_heap[u]) > 0:
(12)                        H.push(dq_heap[u].heap[0])
(8)                     if dq_heap[v].heap[0] == (v, u):
(12)                        H.remove((v, u))
(12)                        dq_heap[v].remove((v, u))
```

```
92: (12)                        if len(dq_heap[v]) > 0:
93: (16)                            H.push(dq_heap[v].heap[0])
94: (8)                     else:
95: (12)                        dq_heap[v].remove((v, u))
96: (8)                     communities[v] = frozenset(communities[u] | communities[v])
97: (8)                     del communities[u]
98: (8)                     u_nbrs = set(dq_dict[u])
99: (8)                     v_nbrs = set(dq_dict[v])
100: (8)                    all_nbrs = (u_nbrs | v_nbrs) - {u, v}
101: (8)                    both_nbrs = u_nbrs & v_nbrs
102: (8)                    for w in all_nbrs:
103: (12)                       if w in both_nbrs:
104: (16)                           dq_vw = dq_dict[v][w] + dq_dict[u][w]
105: (12)                       elif w in v_nbrs:
106: (16)                           dq_vw = dq_dict[v][w] - resolution * (a[u] * b[w] + a[w] *
b[u])
107: (12)                       else:  # w in u_nbrs
108: (16)                           dq_vw = dq_dict[u][w] - resolution * (a[v] * b[w] + a[w] *
b[v])
109: (12)                       for row, col in [(v, w), (w, v)]:
110: (16)                           dq_heap_row = dq_heap[row]
111: (16)                           dq_dict[row][col] = dq_vw
112: (16)                           if len(dq_heap_row) > 0:
113: (20)                               d_oldmax = dq_heap_row.heap[0]
114: (16)                           else:
115: (20)                               d_oldmax = None
116: (16)                           d = (row, col)
117: (16)                           d_negdq = -dq_vw
118: (16)                           if w in v_nbrs:
119: (20)                               dq_heap_row.update(d, d, priority=d_negdq)
120: (16)                           else:
121: (20)                               dq_heap_row.push(d, priority=d_negdq)
122: (16)                           if d_oldmax is None:
123: (20)                               H.push(d, priority=d_negdq)
124: (16)                           else:
125: (20)                               row_max = dq_heap_row.heap[0]
126: (20)                               if d_oldmax != row_max or d_oldmax.priority !=
row_max.priority:
127: (24)                                   H.update(d_oldmax, row_max)
128: (8)                    for w in dq_dict[u]:
129: (12)                       dq_old = dq_dict[w][u]
130: (12)                       del dq_dict[w][u]
131: (12)                       if w != v:
132: (16)                           for row, col in [(w, u), (u, w)]:
133: (20)                               dq_heap_row = dq_heap[row]
134: (20)                               d_old = (row, col)
135: (20)                               if dq_heap_row.heap[0] == d_old:
136: (24)                                   dq_heap_row.remove(d_old)
137: (24)                                   H.remove(d_old)
138: (24)                                   if len(dq_heap_row) > 0:
139: (28)                                       H.push(dq_heap_row.heap[0])
140: (20)                               else:
141: (24)                                   dq_heap_row.remove(d_old)
142: (8)                    del dq_dict[u]
143: (8)                    dq_heap[u] = MappedQueue()
144: (8)                    a[v] += a[u]
145: (8)                    a[u] = 0
146: (8)                    if directed:
147: (12)                       b[v] += b[u]
148: (12)                       b[u] = 0
149: (8)                    yield communities.values()
150: (0)            @nx._dispatchable(edge_attrs="weight")
151: (0)            def greedy_modularity_communities(
152: (4)                G,
153: (4)                weight=None,
154: (4)                resolution=1,
155: (4)                cutoff=1,
156: (4)                best_n=None,
157: (0)            ):
```

```
(4)                 r"""Find communities in G using greedy modularity maximization.
(4)                 This function uses Clauset-Newman-Moore greedy modularity maximization [2]_
(4)                 to find the community partition with the largest modularity.
(4)                 Greedy modularity maximization begins with each node in its own community
(4)                 and repeatedly joins the pair of communities that lead to the largest
(4)                 modularity until no further increase in modularity is possible (a maximum).
(4)                 Two keyword arguments adjust the stopping condition. `cutoff` is a lower
(4)                 limit on the number of communities so you can stop the process before
(4)                 reaching a maximum (used to save computation time). `best_n` is an upper
(4)                 limit on the number of communities so you can make the process continue
(4)                 until at most n communities remain even if the maximum modularity occurs
(4)                 for more. To obtain exactly n communities, set both `cutoff` and `best_n` to n.
(4)                 This function maximizes the generalized modularity, where `resolution`
(4)                 is the resolution parameter, often expressed as $\gamma$.
(4)                 See :func:`~networkx.algorithms.community.quality.modularity`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 weight : string or None, optional (default=None)
(8)                     The name of an edge attribute that holds the numerical value used
(8)                     as a weight.  If None, then each edge has weight 1.
(8)                     The degree is the sum of the edge weights adjacent to the node.
(4)                 resolution : float, optional (default=1)
(8)                     If resolution is less than 1, modularity favors larger communities.
(8)                     Greater than 1 favors smaller communities.
(4)                 cutoff : int, optional (default=1)
(8)                     A minimum number of communities below which the merging process stops.
(8)                     The process stops at this number of communities even if modularity
(8)                     is not maximized. The goal is to let the user stop the process early.
(8)                     The process stops before the cutoff if it finds a maximum of modularity.
(4)                 best_n : int or None, optional (default=None)
(8)                     A maximum number of communities above which the merging process will
(8)                     not stop. This forces community merging to continue after modularity
(8)                     starts to decrease until `best_n` communities remain.
(8)                     If ``None``, don't force it to continue beyond a maximum.
(4)                 Raises
(4)                 ------
(4)                 ValueError : If the `cutoff` or `best_n`  value is not in the range
(8)                     ``[1, G.number_of_nodes()]``, or if `best_n` < `cutoff`.
(4)                 Returns
(4)                 -------
(4)                 communities: list
(8)                     A list of frozensets of nodes, one for each community.
(8)                     Sorted by length with largest communities first.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.karate_club_graph()
(4)                 >>> c = nx.community.greedy_modularity_communities(G)
(4)                 >>> sorted(c[0])
(4)                 [8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]
(4)                 See Also
(4)                 --------
(4)                 modularity
(4)                 References
(4)                 ----------
(4)                 .. [1] Newman, M. E. J. "Networks: An Introduction", page 224
(7)                    Oxford University Press 2011.
(4)                 .. [2] Clauset, A., Newman, M. E., & Moore, C.
(7)                    "Finding community structure in very large networks."
(7)                    Physical Review E 70(6), 2004.
(4)                 .. [3] Reichardt and Bornholdt "Statistical Mechanics of Community
(7)                    Detection" Phys. Rev. E74, 2006.
(4)                 .. [4] Newman, M. E. J."Analysis of weighted networks"
(7)                    Physical Review E 70(5 Pt 2):056131, 2004.
(4)                 """
```

```
(4)                 if not G.size():
(8)                     return [{n} for n in G]
(4)                 if (cutoff < 1) or (cutoff > G.number_of_nodes()):
(8)                     raise ValueError(f"cutoff must be between 1 and {len(G)}. Got
{cutoff}.")
(4)                 if best_n is not None:
(8)                     if (best_n < 1) or (best_n > G.number_of_nodes()):
(12)                        raise ValueError(f"best_n must be between 1 and {len(G)}. Got
{best_n}.")
(8)                     if best_n < cutoff:
(12)                        raise ValueError(f"Must have best_n >= cutoff. Got {best_n} <
{cutoff}")
(8)                     if best_n == 1:
(12)                        return [set(G)]
(4)                 else:
(8)                     best_n = G.number_of_nodes()
(4)                 community_gen = _greedy_modularity_communities_generator(
(8)                     G, weight=weight, resolution=resolution
(4)                 )
(4)                 communities = next(community_gen)
(4)                 while len(communities) > cutoff:
(8)                     try:
(12)                        dq = next(community_gen)
(8)                     except StopIteration:
(12)                        communities = sorted(communities, key=len, reverse=True)
(12)                        while len(communities) > best_n:
(16)                            comm1, comm2, *rest = communities
(16)                            communities = [comm1 ^ comm2]
(16)                            communities.extend(rest)
(12)                        return communities
(8)                     if dq < 0 and len(communities) <= best_n:
(12)                        break
(8)                     communities = next(community_gen)
(4)                 return sorted(communities, key=len, reverse=True)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def naive_greedy_modularity_communities(G, resolution=1, weight=None):
(4)                 r"""Find communities in G using greedy modularity maximization.
(4)                 This implementation is O(n^4), much slower than alternatives, but it is
(4)                 provided as an easy-to-understand reference implementation.
(4)                 Greedy modularity maximization begins with each node in its own community
(4)                 and joins the pair of communities that most increases modularity until no
(4)                 such pair exists.
(4)                 This function maximizes the generalized modularity, where `resolution`
(4)                 is the resolution parameter, often expressed as $\gamma$.
(4)                 See :func:`~networkx.algorithms.community.quality.modularity`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Graph must be simple and undirected.
(4)                 resolution : float (default=1)
(8)                     If resolution is less than 1, modularity favors larger communities.
(8)                     Greater than 1 favors smaller communities.
(4)                 weight : string or None, optional (default=None)
(8)                     The name of an edge attribute that holds the numerical value used
(8)                     as a weight.  If None, then each edge has weight 1.
(8)                     The degree is the sum of the edge weights adjacent to the node.
(4)                 Returns
(4)                 -------
(4)                 list
(8)                     A list of sets of nodes, one for each community.
(8)                     Sorted by length with largest communities first.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.karate_club_graph()
(4)                 >>> c = nx.community.naive_greedy_modularity_communities(G)
(4)                 >>> sorted(c[0])
(4)                 [8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]
```

```
289: (4)                See Also
290: (4)                --------
291: (4)                greedy_modularity_communities
292: (4)                modularity
293: (4)                """
294: (4)                communities = [frozenset([u]) for u in G.nodes()]
295: (4)                merges = []
296: (4)                old_modularity = None
297: (4)                new_modularity = modularity(G, communities, resolution=resolution,
weight=weight)
298: (4)                while old_modularity is None or new_modularity > old_modularity:
299: (8)                    old_modularity = new_modularity
300: (8)                    trial_communities = list(communities)
301: (8)                    to_merge = None
302: (8)                    for i, u in enumerate(communities):
303: (12)                       for j, v in enumerate(communities):
304: (16)                           if j <= i or len(u) == 0 or len(v) == 0:
305: (20)                               continue
306: (16)                           trial_communities[j] = u | v
307: (16)                           trial_communities[i] = frozenset([])
308: (16)                           trial_modularity = modularity(
309: (20)                               G, trial_communities, resolution=resolution, weight=weight
310: (16)                           )
311: (16)                           if trial_modularity >= new_modularity:
312: (20)                               if trial_modularity > new_modularity:
313: (24)                                   new_modularity = trial_modularity
314: (24)                                   to_merge = (i, j, new_modularity - old_modularity)
315: (20)                               elif to_merge and min(i, j) < min(to_merge[0],
to_merge[1]):
316: (24)                                   new_modularity = trial_modularity
317: (24)                                   to_merge = (i, j, new_modularity - old_modularity)
318: (16)                           trial_communities[i] = u
319: (16)                           trial_communities[j] = v
320: (8)                    if to_merge is not None:
321: (12)                       merges.append(to_merge)
322: (12)                       i, j, dq = to_merge
323: (12)                       u, v = communities[i], communities[j]
324: (12)                       communities[j] = u | v
325: (12)                       communities[i] = frozenset([])
326: (4)                return sorted((c for c in communities if len(c) > 0), key=len,
reverse=True)

----------------------------------------

File 191 - __init__.py:

1: (0)

----------------------------------------

File 192 - test_asyn_fluid.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import Graph, NetworkXError
4: (0)              from networkx.algorithms.community import asyn_fluidc
5: (0)              @pytest.mark.parametrize("graph_constructor", (nx.DiGraph, nx.MultiGraph))
6: (0)              def test_raises_on_directed_and_multigraphs(graph_constructor):
7: (4)                  G = graph_constructor([(0, 1), (1, 2)])
8: (4)                  with pytest.raises(nx.NetworkXNotImplemented):
9: (8)                      nx.community.asyn_fluidc(G, 1)
10: (0)             def test_exceptions():
11: (4)                 test = Graph()
12: (4)                 test.add_node("a")
13: (4)                 pytest.raises(NetworkXError, asyn_fluidc, test, "hi")
14: (4)                 pytest.raises(NetworkXError, asyn_fluidc, test, -1)
15: (4)                 pytest.raises(NetworkXError, asyn_fluidc, test, 3)
16: (4)                 test.add_node("b")
17: (4)                 pytest.raises(NetworkXError, asyn_fluidc, test, 1)
```

```
(0)             def test_single_node():
(4)                 test = Graph()
(4)                 test.add_node("a")
(4)                 ground_truth = {frozenset(["a"])}
(4)                 communities = asyn_fluidc(test, 1)
(4)                 result = {frozenset(c) for c in communities}
(4)                 assert result == ground_truth
(0)             def test_two_nodes():
(4)                 test = Graph()
(4)                 test.add_edge("a", "b")
(4)                 ground_truth = {frozenset(["a"]), frozenset(["b"])}
(4)                 communities = asyn_fluidc(test, 2)
(4)                 result = {frozenset(c) for c in communities}
(4)                 assert result == ground_truth
(0)             def test_two_clique_communities():
(4)                 test = Graph()
(4)                 test.add_edge("a", "b")
(4)                 test.add_edge("a", "c")
(4)                 test.add_edge("b", "c")
(4)                 test.add_edge("c", "d")
(4)                 test.add_edge("d", "e")
(4)                 test.add_edge("d", "f")
(4)                 test.add_edge("f", "e")
(4)                 ground_truth = {frozenset(["a", "c", "b"]), frozenset(["e", "d", "f"])}
(4)                 communities = asyn_fluidc(test, 2, seed=7)
(4)                 result = {frozenset(c) for c in communities}
(4)                 assert result == ground_truth
(0)             def test_five_clique_ring():
(4)                 test = Graph()
(4)                 test.add_edge("1a", "1b")
(4)                 test.add_edge("1a", "1c")
(4)                 test.add_edge("1a", "1d")
(4)                 test.add_edge("1b", "1c")
(4)                 test.add_edge("1b", "1d")
(4)                 test.add_edge("1c", "1d")
(4)                 test.add_edge("2a", "2b")
(4)                 test.add_edge("2a", "2c")
(4)                 test.add_edge("2a", "2d")
(4)                 test.add_edge("2b", "2c")
(4)                 test.add_edge("2b", "2d")
(4)                 test.add_edge("2c", "2d")
(4)                 test.add_edge("3a", "3b")
(4)                 test.add_edge("3a", "3c")
(4)                 test.add_edge("3a", "3d")
(4)                 test.add_edge("3b", "3c")
(4)                 test.add_edge("3b", "3d")
(4)                 test.add_edge("3c", "3d")
(4)                 test.add_edge("4a", "4b")
(4)                 test.add_edge("4a", "4c")
(4)                 test.add_edge("4a", "4d")
(4)                 test.add_edge("4b", "4c")
(4)                 test.add_edge("4b", "4d")
(4)                 test.add_edge("4c", "4d")
(4)                 test.add_edge("5a", "5b")
(4)                 test.add_edge("5a", "5c")
(4)                 test.add_edge("5a", "5d")
(4)                 test.add_edge("5b", "5c")
(4)                 test.add_edge("5b", "5d")
(4)                 test.add_edge("5c", "5d")
(4)                 test.add_edge("1a", "2c")
(4)                 test.add_edge("2a", "3c")
(4)                 test.add_edge("3a", "4c")
(4)                 test.add_edge("4a", "5c")
(4)                 test.add_edge("5a", "1c")
(4)                 ground_truth = {
(8)                     frozenset(["1a", "1b", "1c", "1d"]),
(8)                     frozenset(["2a", "2b", "2c", "2d"]),
(8)                     frozenset(["3a", "3b", "3c", "3d"]),
(8)                     frozenset(["4a", "4b", "4c", "4d"]),
```

```
87: (8)                     frozenset(["5a", "5b", "5c", "5d"]),
88: (4)                 }
89: (4)                 communities = asyn_fluidc(test, 5, seed=9)
90: (4)                 result = {frozenset(c) for c in communities}
91: (4)                 assert result == ground_truth
```

File 193 - test_centrality.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.community.centrality`
2: (0)              module.
3: (0)              """
4: (0)              from operator import itemgetter
5: (0)              import networkx as nx
6: (0)              def set_of_sets(iterable):
7: (4)                  return set(map(frozenset, iterable))
8: (0)              def validate_communities(result, expected):
9: (4)                  assert set_of_sets(result) == set_of_sets(expected)
10: (0)             def validate_possible_communities(result, *expected):
11: (4)                 assert any(set_of_sets(result) == set_of_sets(p) for p in expected)
12: (0)             class TestGirvanNewman:
13: (4)                 """Unit tests for the
14: (4)                 :func:`networkx.algorithms.community.centrality.girvan_newman`
15: (4)                 function.
16: (4)                 """
17: (4)                 def test_no_edges(self):
18: (8)                     G = nx.empty_graph(3)
19: (8)                     communities = list(nx.community.girvan_newman(G))
20: (8)                     assert len(communities) == 1
21: (8)                     validate_communities(communities[0], [{0}, {1}, {2}])
22: (4)                 def test_undirected(self):
23: (8)                     G = nx.path_graph(4)
24: (8)                     communities = list(nx.community.girvan_newman(G))
25: (8)                     assert len(communities) == 3
26: (8)                     validate_communities(communities[0], [{0, 1}, {2, 3}])
27: (8)                     validate_possible_communities(
28: (12)                        communities[1], [{0}, {1}, {2, 3}], [{0, 1}, {2}, {3}]
29: (8)                     )
30: (8)                     validate_communities(communities[2], [{0}, {1}, {2}, {3}])
31: (4)                 def test_directed(self):
32: (8)                     G = nx.DiGraph(nx.path_graph(4))
33: (8)                     communities = list(nx.community.girvan_newman(G))
34: (8)                     assert len(communities) == 3
35: (8)                     validate_communities(communities[0], [{0, 1}, {2, 3}])
36: (8)                     validate_possible_communities(
37: (12)                        communities[1], [{0}, {1}, {2, 3}], [{0, 1}, {2}, {3}]
38: (8)                     )
39: (8)                     validate_communities(communities[2], [{0}, {1}, {2}, {3}])
40: (4)                 def test_selfloops(self):
41: (8)                     G = nx.path_graph(4)
42: (8)                     G.add_edge(0, 0)
43: (8)                     G.add_edge(2, 2)
44: (8)                     communities = list(nx.community.girvan_newman(G))
45: (8)                     assert len(communities) == 3
46: (8)                     validate_communities(communities[0], [{0, 1}, {2, 3}])
47: (8)                     validate_possible_communities(
48: (12)                        communities[1], [{0}, {1}, {2, 3}], [{0, 1}, {2}, {3}]
49: (8)                     )
50: (8)                     validate_communities(communities[2], [{0}, {1}, {2}, {3}])
51: (4)                 def test_most_valuable_edge(self):
52: (8)                     G = nx.Graph()
53: (8)                     G.add_weighted_edges_from([(0, 1, 3), (1, 2, 2), (2, 3, 1)])
54: (8)                     def heaviest(G):
55: (12)                        return max(G.edges(data="weight"), key=itemgetter(2))[:2]
56: (8)                     communities = list(nx.community.girvan_newman(G, heaviest))
57: (8)                     assert len(communities) == 3
58: (8)                     validate_communities(communities[0], [{0}, {1, 2, 3}])
59: (8)                     validate_communities(communities[1], [{0}, {1}, {2, 3}])
```

```
60: (8)                     validate_communities(communities[2], [{0}, {1}, {2}, {3}])

----------------------------------------

File 194 - test_divisive.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_edge_betweenness_partition():
4: (4)                  G = nx.barbell_graph(3, 0)
5: (4)                  C = nx.community.edge_betweenness_partition(G, 2)
6: (4)                  answer = [{0, 1, 2}, {3, 4, 5}]
7: (4)                  assert len(C) == len(answer)
8: (4)                  for s in answer:
9: (8)                      assert s in C
10: (4)                 G = nx.barbell_graph(3, 1)
11: (4)                 C = nx.community.edge_betweenness_partition(G, 3)
12: (4)                 answer = [{0, 1, 2}, {4, 5, 6}, {3}]
13: (4)                 assert len(C) == len(answer)
14: (4)                 for s in answer:
15: (8)                     assert s in C
16: (4)                 C = nx.community.edge_betweenness_partition(G, 7)
17: (4)                 answer = [{n} for n in G]
18: (4)                 assert len(C) == len(answer)
19: (4)                 for s in answer:
20: (8)                     assert s in C
21: (4)                 C = nx.community.edge_betweenness_partition(G, 1)
22: (4)                 assert C == [set(G)]
23: (4)                 C = nx.community.edge_betweenness_partition(G, 1, weight="weight")
24: (4)                 assert C == [set(G)]
25: (4)                 with pytest.raises(nx.NetworkXError):
26: (8)                     nx.community.edge_betweenness_partition(G, 0)
27: (4)                 with pytest.raises(nx.NetworkXError):
28: (8)                     nx.community.edge_betweenness_partition(G, -1)
29: (4)                 with pytest.raises(nx.NetworkXError):
30: (8)                     nx.community.edge_betweenness_partition(G, 10)
31: (0)             def test_edge_current_flow_betweenness_partition():
32: (4)                 pytest.importorskip("scipy")
33: (4)                 G = nx.barbell_graph(3, 0)
34: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 2)
35: (4)                 answer = [{0, 1, 2}, {3, 4, 5}]
36: (4)                 assert len(C) == len(answer)
37: (4)                 for s in answer:
38: (8)                     assert s in C
39: (4)                 G = nx.barbell_graph(3, 1)
40: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 2)
41: (4)                 answers = [[{0, 1, 2, 3}, {4, 5, 6}], [{0, 1, 2}, {3, 4, 5, 6}]]
42: (4)                 assert len(C) == len(answers[0])
43: (4)                 assert any(all(s in answer for s in C) for answer in answers)
44: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 3)
45: (4)                 answer = [{0, 1, 2}, {4, 5, 6}, {3}]
46: (4)                 assert len(C) == len(answer)
47: (4)                 for s in answer:
48: (8)                     assert s in C
49: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 4)
50: (4)                 answers = [[{1, 2}, {4, 5, 6}, {3}, {0}], [{0, 1, 2}, {5, 6}, {3}, {4}]]
51: (4)                 assert len(C) == len(answers[0])
52: (4)                 assert any(all(s in answer for s in C) for answer in answers)
53: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 5)
54: (4)                 answer = [{1, 2}, {5, 6}, {3}, {0}, {4}]
55: (4)                 assert len(C) == len(answer)
56: (4)                 for s in answer:
57: (8)                     assert s in C
58: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 6)
59: (4)                 answers = [[{2}, {5, 6}, {3}, {0}, {4}, {1}], [{1, 2}, {6}, {3}, {0}, {4},
{5}]]
60: (4)                 assert len(C) == len(answers[0])
61: (4)                 assert any(all(s in answer for s in C) for answer in answers)
62: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 7)
```

```
63: (4)                 answer = [{n} for n in G]
64: (4)                 assert len(C) == len(answer)
65: (4)                 for s in answer:
66: (8)                     assert s in C
67: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 1)
68: (4)                 assert C == [set(G)]
69: (4)                 C = nx.community.edge_current_flow_betweenness_partition(G, 1,
weight="weight")
70: (4)                 assert C == [set(G)]
71: (4)                 with pytest.raises(nx.NetworkXError):
72: (8)                     nx.community.edge_current_flow_betweenness_partition(G, 0)
73: (4)                 with pytest.raises(nx.NetworkXError):
74: (8)                     nx.community.edge_current_flow_betweenness_partition(G, -1)
75: (4)                 with pytest.raises(nx.NetworkXError):
76: (8)                     nx.community.edge_current_flow_betweenness_partition(G, 10)
77: (4)                 N = 10
78: (4)                 G = nx.empty_graph(N)
79: (4)                 for i in range(2, N - 1):
80: (8)                     C = nx.community.edge_current_flow_betweenness_partition(G, i)
81: (8)                     assert C == [{n} for n in G]

----------------------------------------

File 195 - test_kclique.py:

1: (0)              from itertools import combinations
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              def test_overlapping_K5():
5: (4)                  G = nx.Graph()
6: (4)                  G.add_edges_from(combinations(range(5), 2))  # Add a five clique
7: (4)                  G.add_edges_from(combinations(range(2, 7), 2))  # Add another five clique
8: (4)                  c = list(nx.community.k_clique_communities(G, 4))
9: (4)                  assert c == [frozenset(range(7))]
10: (4)                 c = set(nx.community.k_clique_communities(G, 5))
11: (4)                 assert c == {frozenset(range(5)), frozenset(range(2, 7))}
12: (0)             def test_isolated_K5():
13: (4)                 G = nx.Graph()
14: (4)                 G.add_edges_from(combinations(range(5), 2))  # Add a five clique
15: (4)                 G.add_edges_from(combinations(range(5, 10), 2))  # Add another five clique
16: (4)                 c = set(nx.community.k_clique_communities(G, 5))
17: (4)                 assert c == {frozenset(range(5)), frozenset(range(5, 10))}
18: (0)             class TestZacharyKarateClub:
19: (4)                 def setup_method(self):
20: (8)                     self.G = nx.karate_club_graph()
21: (4)                 def _check_communities(self, k, expected):
22: (8)                     communities = set(nx.community.k_clique_communities(self.G, k))
23: (8)                     assert communities == expected
24: (4)                 def test_k2(self):
25: (8)                     expected = {frozenset(self.G)}
26: (8)                     self._check_communities(2, expected)
27: (4)                 def test_k3(self):
28: (8)                     comm1 = [
29: (12)                        0,
30: (12)                        1,
31: (12)                        2,
32: (12)                        3,
33: (12)                        7,
34: (12)                        8,
35: (12)                        12,
36: (12)                        13,
37: (12)                        14,
38: (12)                        15,
39: (12)                        17,
40: (12)                        18,
41: (12)                        19,
42: (12)                        20,
43: (12)                        21,
44: (12)                        22,
```

```
45: (12)                        23,
46: (12)                        26,
47: (12)                        27,
48: (12)                        28,
49: (12)                        29,
50: (12)                        30,
51: (12)                        31,
52: (12)                        32,
53: (12)                        33,
54: (8)                     ]
55: (8)                     comm2 = [0, 4, 5, 6, 10, 16]
56: (8)                     comm3 = [24, 25, 31]
57: (8)                     expected = {frozenset(comm1), frozenset(comm2), frozenset(comm3)}
58: (8)                     self._check_communities(3, expected)
59: (4)                 def test_k4(self):
60: (8)                     expected = {
61: (12)                        frozenset([0, 1, 2, 3, 7, 13]),
62: (12)                        frozenset([8, 32, 30, 33]),
63: (12)                        frozenset([32, 33, 29, 23]),
64: (8)                     }
65: (8)                     self._check_communities(4, expected)
66: (4)                 def test_k5(self):
67: (8)                     expected = {frozenset([0, 1, 2, 3, 7, 13])}
68: (8)                     self._check_communities(5, expected)
69: (4)                 def test_k6(self):
70: (8)                     expected = set()
71: (8)                     self._check_communities(6, expected)
72: (0)             def test_bad_k():
73: (4)                 with pytest.raises(nx.NetworkXError):
74: (8)                     list(nx.community.k_clique_communities(nx.Graph(), 1))
```

File 196 - test_kernighan_lin.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.community.kernighan_lin`
2: (0)              module.
3: (0)              """
4: (0)              from itertools import permutations
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              from networkx.algorithms.community import kernighan_lin_bisection
8: (0)              def assert_partition_equal(x, y):
9: (4)                  assert set(map(frozenset, x)) == set(map(frozenset, y))
10: (0)             def test_partition():
11: (4)                 G = nx.barbell_graph(3, 0)
12: (4)                 C = kernighan_lin_bisection(G)
13: (4)                 assert_partition_equal(C, [{0, 1, 2}, {3, 4, 5}])
14: (0)             def test_partition_argument():
15: (4)                 G = nx.barbell_graph(3, 0)
16: (4)                 partition = [{0, 1, 2}, {3, 4, 5}]
17: (4)                 C = kernighan_lin_bisection(G, partition)
18: (4)                 assert_partition_equal(C, partition)
19: (0)             def test_partition_argument_non_integer_nodes():
20: (4)                 G = nx.Graph([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
21: (4)                 partition = ({"A", "B"}, {"C", "D"})
22: (4)                 C = kernighan_lin_bisection(G, partition)
23: (4)                 assert_partition_equal(C, partition)
24: (0)             def test_seed_argument():
25: (4)                 G = nx.barbell_graph(3, 0)
26: (4)                 C = kernighan_lin_bisection(G, seed=1)
27: (4)                 assert_partition_equal(C, [{0, 1, 2}, {3, 4, 5}])
28: (0)             def test_non_disjoint_partition():
29: (4)                 with pytest.raises(nx.NetworkXError):
30: (8)                     G = nx.barbell_graph(3, 0)
31: (8)                     partition = ({0, 1, 2}, {2, 3, 4, 5})
32: (8)                     kernighan_lin_bisection(G, partition)
33: (0)             def test_too_many_blocks():
34: (4)                 with pytest.raises(nx.NetworkXError):
```

```
35: (8)                     G = nx.barbell_graph(3, 0)
36: (8)                     partition = ({0, 1}, {2}, {3, 4, 5})
37: (8)                     kernighan_lin_bisection(G, partition)
38: (0)             def test_multigraph():
39: (4)                 G = nx.cycle_graph(4)
40: (4)                 M = nx.MultiGraph(G.edges())
41: (4)                 M.add_edges_from(G.edges())
42: (4)                 M.remove_edge(1, 2)
43: (4)                 for labels in permutations(range(4)):
44: (8)                     mapping = dict(zip(M, labels))
45: (8)                     A, B = kernighan_lin_bisection(nx.relabel_nodes(M, mapping), seed=0)
46: (8)                     assert_partition_equal(
47: (12)                        [A, B], [{mapping[0], mapping[1]}, {mapping[2], mapping[3]}]
48: (8)                     )
49: (0)             def test_max_iter_argument():
50: (4)                 G = nx.Graph(
51: (8)                     [
52: (12)                        ("A", "B", {"weight": 1}),
53: (12)                        ("A", "C", {"weight": 2}),
54: (12)                        ("A", "D", {"weight": 3}),
55: (12)                        ("A", "E", {"weight": 2}),
56: (12)                        ("A", "F", {"weight": 4}),
57: (12)                        ("B", "C", {"weight": 1}),
58: (12)                        ("B", "D", {"weight": 4}),
59: (12)                        ("B", "E", {"weight": 2}),
60: (12)                        ("B", "F", {"weight": 1}),
61: (12)                        ("C", "D", {"weight": 3}),
62: (12)                        ("C", "E", {"weight": 2}),
63: (12)                        ("C", "F", {"weight": 1}),
64: (12)                        ("D", "E", {"weight": 4}),
65: (12)                        ("D", "F", {"weight": 3}),
66: (12)                        ("E", "F", {"weight": 2}),
67: (8)                     ]
68: (4)                 )
69: (4)                 partition = ({"A", "B", "C"}, {"D", "E", "F"})
70: (4)                 C = kernighan_lin_bisection(G, partition, max_iter=1)
71: (4)                 assert_partition_equal(C, ({"A", "F", "C"}, {"D", "E", "B"}))
```

File 197 - test_label_propagation.py:

```
1: (0)              from itertools import chain, combinations
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              def test_directed_not_supported():
5: (4)                  with pytest.raises(nx.NetworkXNotImplemented):
6: (8)                      test = nx.DiGraph()
7: (8)                      test.add_edge("a", "b")
8: (8)                      test.add_edge("a", "c")
9: (8)                      test.add_edge("b", "d")
10: (8)                     result = nx.community.label_propagation_communities(test)
11: (0)             def test_iterator_vs_iterable():
12: (4)                 G = nx.empty_graph("a")
13: (4)                 assert list(nx.community.label_propagation_communities(G)) == [{"a"}]
14: (4)                 for community in nx.community.label_propagation_communities(G):
15: (8)                     assert community == {"a"}
16: (4)                 pytest.raises(TypeError, next,
nx.community.label_propagation_communities(G))
17: (0)             def test_one_node():
18: (4)                 test = nx.Graph()
19: (4)                 test.add_node("a")
20: (4)                 ground_truth = {frozenset(["a"])}
21: (4)                 communities = nx.community.label_propagation_communities(test)
22: (4)                 result = {frozenset(c) for c in communities}
23: (4)                 assert result == ground_truth
24: (0)             def test_unconnected_communities():
25: (4)                 test = nx.Graph()
26: (4)                 test.add_edge("a", "c")
```

```
27: (4)                 test.add_edge("a", "d")
28: (4)                 test.add_edge("d", "c")
29: (4)                 test.add_edge("b", "e")
30: (4)                 test.add_edge("e", "f")
31: (4)                 test.add_edge("f", "b")
32: (4)                 ground_truth = {frozenset(["a", "c", "d"]), frozenset(["b", "e", "f"])}
33: (4)                 communities = nx.community.label_propagation_communities(test)
34: (4)                 result = {frozenset(c) for c in communities}
35: (4)                 assert result == ground_truth
36: (0)             def test_connected_communities():
37: (4)                 test = nx.Graph()
38: (4)                 test.add_edge("a", "b")
39: (4)                 test.add_edge("c", "a")
40: (4)                 test.add_edge("c", "b")
41: (4)                 test.add_edge("d", "a")
42: (4)                 test.add_edge("d", "b")
43: (4)                 test.add_edge("d", "c")
44: (4)                 test.add_edge("e", "a")
45: (4)                 test.add_edge("e", "b")
46: (4)                 test.add_edge("e", "c")
47: (4)                 test.add_edge("e", "d")
48: (4)                 test.add_edge("1", "2")
49: (4)                 test.add_edge("3", "1")
50: (4)                 test.add_edge("3", "2")
51: (4)                 test.add_edge("4", "1")
52: (4)                 test.add_edge("4", "2")
53: (4)                 test.add_edge("4", "3")
54: (4)                 test.add_edge("5", "1")
55: (4)                 test.add_edge("5", "2")
56: (4)                 test.add_edge("5", "3")
57: (4)                 test.add_edge("5", "4")
58: (4)                 test.add_edge("a", "1")
59: (4)                 test.add_edge("x", "y")
60: (4)                 test.add_node("z")
61: (4)                 ground_truth1 = {
62: (8)                     frozenset(["a", "b", "c", "d", "e"]),
63: (8)                     frozenset(["1", "2", "3", "4", "5"]),
64: (8)                     frozenset(["x", "y"]),
65: (8)                     frozenset(["z"]),
66: (4)                 }
67: (4)                 ground_truth2 = {
68: (8)                     frozenset(["a", "b", "c", "d", "e", "1", "2", "3", "4", "5"]),
69: (8)                     frozenset(["x", "y"]),
70: (8)                     frozenset(["z"]),
71: (4)                 }
72: (4)                 ground_truth = (ground_truth1, ground_truth2)
73: (4)                 communities = nx.community.label_propagation_communities(test)
74: (4)                 result = {frozenset(c) for c in communities}
75: (4)                 assert result in ground_truth
76: (0)             def test_termination():
77: (4)                 test1 = nx.karate_club_graph()
78: (4)                 test2 = nx.caveman_graph(2, 10)
79: (4)                 test2.add_edges_from([(0, 20), (20, 10)])
80: (4)                 nx.community.asyn_lpa_communities(test1)
81: (4)                 nx.community.asyn_lpa_communities(test2)
82: (0)             class TestAsynLpaCommunities:
83: (4)                 def _check_communities(self, G, expected):
84: (8)                     """Checks that the communities computed from the given graph ``G``
85: (8)                     using the :func:`~networkx.asyn_lpa_communities` function match
86: (8)                     the set of nodes given in ``expected``.
87: (8)                     ``expected`` must be a :class:`set` of :class:`frozenset`
88: (8)                     instances, each element of which is a node in the graph.
89: (8)                     """
90: (8)                     communities = nx.community.asyn_lpa_communities(G)
91: (8)                     result = {frozenset(c) for c in communities}
92: (8)                     assert result == expected
93: (4)                 def test_null_graph(self):
94: (8)                     G = nx.null_graph()
95: (8)                     ground_truth = set()
```

```
96: (8)                     self._check_communities(G, ground_truth)
97: (4)                 def test_single_node(self):
98: (8)                     G = nx.empty_graph(1)
99: (8)                     ground_truth = {frozenset([0])}
100: (8)                    self._check_communities(G, ground_truth)
101: (4)                def test_simple_communities(self):
102: (8)                    G = nx.Graph(["ab", "ac", "bc", "de", "df", "fe"])
103: (8)                    ground_truth = {frozenset("abc"), frozenset("def")}
104: (8)                    self._check_communities(G, ground_truth)
105: (4)                def test_seed_argument(self):
106: (8)                    G = nx.Graph(["ab", "ac", "bc", "de", "df", "fe"])
107: (8)                    ground_truth = {frozenset("abc"), frozenset("def")}
108: (8)                    communities = nx.community.asyn_lpa_communities(G, seed=1)
109: (8)                    result = {frozenset(c) for c in communities}
110: (8)                    assert result == ground_truth
111: (4)                def test_several_communities(self):
112: (8)                    ground_truth = {frozenset(range(3 * i, 3 * (i + 1))) for i in
range(5)}
113: (8)                    edges = chain.from_iterable(combinations(c, 2) for c in ground_truth)
114: (8)                    G = nx.Graph(edges)
115: (8)                    self._check_communities(G, ground_truth)
116: (0)            class TestFastLabelPropagationCommunities:
117: (4)                N = 100  # number of nodes
118: (4)                K = 15  # average node degree
119: (4)                def _check_communities(self, G, truth, weight=None, seed=42):
120: (8)                    C = nx.community.fast_label_propagation_communities(G, weight=weight,
seed=seed)
121: (8)                    assert {frozenset(c) for c in C} == truth
122: (4)                def test_null_graph(self):
123: (8)                    G = nx.null_graph()
124: (8)                    truth = set()
125: (8)                    self._check_communities(G, truth)
126: (4)                def test_empty_graph(self):
127: (8)                    G = nx.empty_graph(self.N)
128: (8)                    truth = {frozenset([i]) for i in G}
129: (8)                    self._check_communities(G, truth)
130: (4)                def test_star_graph(self):
131: (8)                    G = nx.star_graph(self.N)
132: (8)                    truth = {frozenset(G)}
133: (8)                    self._check_communities(G, truth)
134: (4)                def test_complete_graph(self):
135: (8)                    G = nx.complete_graph(self.N)
136: (8)                    truth = {frozenset(G)}
137: (8)                    self._check_communities(G, truth)
138: (4)                def test_bipartite_graph(self):
139: (8)                    G = nx.complete_bipartite_graph(self.N // 2, self.N // 2)
140: (8)                    truth = {frozenset(G)}
141: (8)                    self._check_communities(G, truth)
142: (4)                def test_random_graph(self):
143: (8)                    G = nx.gnm_random_graph(self.N, self.N * self.K // 2, seed=42)
144: (8)                    truth = {frozenset(G)}
145: (8)                    self._check_communities(G, truth)
146: (4)                def test_disjoin_cliques(self):
147: (8)                    G = nx.Graph(["ab", "AB", "AC", "BC", "12", "13", "14", "23", "24",
"34"])
148: (8)                    truth = {frozenset("ab"), frozenset("ABC"), frozenset("1234")}
149: (8)                    self._check_communities(G, truth)
150: (4)                def test_ring_of_cliques(self):
151: (8)                    N, K = self.N, self.K
152: (8)                    G = nx.ring_of_cliques(N, K)
153: (8)                    truth = {frozenset([K * i + k for k in range(K)]) for i in range(N)}
154: (8)                    self._check_communities(G, truth)
155: (4)                def test_larger_graph(self):
156: (8)                    G = nx.gnm_random_graph(100 * self.N, 50 * self.N * self.K, seed=42)
157: (8)                    nx.community.fast_label_propagation_communities(G)
158: (4)                def test_graph_type(self):
159: (8)                    G1 = nx.complete_graph(self.N, nx.MultiDiGraph())
160: (8)                    G2 = nx.MultiGraph(G1)
161: (8)                    G3 = nx.DiGraph(G1)
```

```
162: (8)                    G4 = nx.Graph(G1)
163: (8)                    truth = {frozenset(G1)}
164: (8)                    self._check_communities(G1, truth)
165: (8)                    self._check_communities(G2, truth)
166: (8)                    self._check_communities(G3, truth)
167: (8)                    self._check_communities(G4, truth)
168: (4)                def test_weight_argument(self):
169: (8)                    G = nx.MultiDiGraph()
170: (8)                    G.add_edge(1, 2, weight=1.41)
171: (8)                    G.add_edge(2, 1, weight=1.41)
172: (8)                    G.add_edge(2, 3)
173: (8)                    G.add_edge(3, 4, weight=3.14)
174: (8)                    truth = {frozenset({1, 2}), frozenset({3, 4})}
175: (8)                    self._check_communities(G, truth, weight="weight")
176: (4)                def test_seed_argument(self):
177: (8)                    G = nx.karate_club_graph()
178: (8)                    C = nx.community.fast_label_propagation_communities(G, seed=2023)
179: (8)                    truth = {frozenset(c) for c in C}
180: (8)                    self._check_communities(G, truth, seed=2023)
181: (8)                    C = nx.community.fast_label_propagation_communities(G, seed=None)

----------------------------------------

File 198 - test_lukes.py:

1: (0)              from itertools import product
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              EWL = "e_weight"
5: (0)              NWL = "n_weight"
6: (0)              def paper_1_case(float_edge_wt=False, explicit_node_wt=True, directed=False):
7: (4)                  limit = 3
8: (4)                  if float_edge_wt:
9: (8)                      shift = 0.001
10: (4)                 else:
11: (8)                     shift = 0
12: (4)                 if directed:
13: (8)                     example_1 = nx.DiGraph()
14: (4)                 else:
15: (8)                     example_1 = nx.Graph()
16: (4)                 example_1.add_edge(1, 2, **{EWL: 3 + shift})
17: (4)                 example_1.add_edge(1, 4, **{EWL: 2 + shift})
18: (4)                 example_1.add_edge(2, 3, **{EWL: 4 + shift})
19: (4)                 example_1.add_edge(2, 5, **{EWL: 6 + shift})
20: (4)                 if explicit_node_wt:
21: (8)                     nx.set_node_attributes(example_1, 1, NWL)
22: (8)                     wtu = NWL
23: (4)                 else:
24: (8)                     wtu = None
25: (4)                 clusters_1 = {
26: (8)                     frozenset(x)
27: (8)                     for x in nx.community.lukes_partitioning(
28: (12)                        example_1, limit, node_weight=wtu, edge_weight=EWL
29: (8)                     )
30: (4)                 }
31: (4)                 return clusters_1
32: (0)             def paper_2_case(explicit_edge_wt=True, directed=False):
33: (4)                 byte_block_size = 32
34: (4)                 if directed:
35: (8)                     example_2 = nx.DiGraph()
36: (4)                 else:
37: (8)                     example_2 = nx.Graph()
38: (4)                 if explicit_edge_wt:
39: (8)                     edic = {EWL: 1}
40: (8)                     wtu = EWL
41: (4)                 else:
42: (8)                     edic = {}
43: (8)                     wtu = None
44: (4)                 example_2.add_edge("name", "home_address", **edic)
```

```
45: (4)                 example_2.add_edge("name", "education", **edic)
46: (4)                 example_2.add_edge("education", "bs", **edic)
47: (4)                 example_2.add_edge("education", "ms", **edic)
48: (4)                 example_2.add_edge("education", "phd", **edic)
49: (4)                 example_2.add_edge("name", "telephone", **edic)
50: (4)                 example_2.add_edge("telephone", "home", **edic)
51: (4)                 example_2.add_edge("telephone", "office", **edic)
52: (4)                 example_2.add_edge("office", "no1", **edic)
53: (4)                 example_2.add_edge("office", "no2", **edic)
54: (4)                 example_2.nodes["name"][NWL] = 20
55: (4)                 example_2.nodes["education"][NWL] = 10
56: (4)                 example_2.nodes["bs"][NWL] = 1
57: (4)                 example_2.nodes["ms"][NWL] = 1
58: (4)                 example_2.nodes["phd"][NWL] = 1
59: (4)                 example_2.nodes["home_address"][NWL] = 8
60: (4)                 example_2.nodes["telephone"][NWL] = 8
61: (4)                 example_2.nodes["home"][NWL] = 8
62: (4)                 example_2.nodes["office"][NWL] = 4
63: (4)                 example_2.nodes["no1"][NWL] = 1
64: (4)                 example_2.nodes["no2"][NWL] = 1
65: (4)                 clusters_2 = {
66: (8)                     frozenset(x)
67: (8)                     for x in nx.community.lukes_partitioning(
68: (12)                        example_2, byte_block_size, node_weight=NWL, edge_weight=wtu
69: (8)                     )
70: (4)                 }
71: (4)                 return clusters_2
72: (0)             def test_paper_1_case():
73: (4)                 ground_truth = {frozenset([1, 4]), frozenset([2, 3, 5])}
74: (4)                 tf = (True, False)
75: (4)                 for flt, nwt, drc in product(tf, tf, tf):
76: (8)                     part = paper_1_case(flt, nwt, drc)
77: (8)                     assert part == ground_truth
78: (0)             def test_paper_2_case():
79: (4)                 ground_truth = {
80: (8)                     frozenset(["education", "bs", "ms", "phd"]),
81: (8)                     frozenset(["name", "home_address"]),
82: (8)                     frozenset(["telephone", "home", "office", "no1", "no2"]),
83: (4)                 }
84: (4)                 tf = (True, False)
85: (4)                 for ewt, drc in product(tf, tf):
86: (8)                     part = paper_2_case(ewt, drc)
87: (8)                     assert part == ground_truth
88: (0)             def test_mandatory_tree():
89: (4)                 not_a_tree = nx.complete_graph(4)
90: (4)                 with pytest.raises(nx.NotATree):
91: (8)                     nx.community.lukes_partitioning(not_a_tree, 5)
92: (0)             def test_mandatory_integrality():
93: (4)                 byte_block_size = 32
94: (4)                 ex_1_broken = nx.DiGraph()
95: (4)                 ex_1_broken.add_edge(1, 2, **{EWL: 3.2})
96: (4)                 ex_1_broken.add_edge(1, 4, **{EWL: 2.4})
97: (4)                 ex_1_broken.add_edge(2, 3, **{EWL: 4.0})
98: (4)                 ex_1_broken.add_edge(2, 5, **{EWL: 6.3})
99: (4)                 ex_1_broken.nodes[1][NWL] = 1.2  # !
100: (4)                ex_1_broken.nodes[2][NWL] = 1
101: (4)                ex_1_broken.nodes[3][NWL] = 1
102: (4)                ex_1_broken.nodes[4][NWL] = 1
103: (4)                ex_1_broken.nodes[5][NWL] = 2
104: (4)                with pytest.raises(TypeError):
105: (8)                    nx.community.lukes_partitioning(
106: (12)                       ex_1_broken, byte_block_size, node_weight=NWL, edge_weight=EWL
107: (8)                    )

----------------------------------------

File 199 - test_louvain.py:

1: (0)              import pytest
```

```
2: (0)              import networkx as nx
3: (0)              def test_modularity_increase():
4: (4)                  G = nx.LFR_benchmark_graph(
5: (8)                      250, 3, 1.5, 0.009, average_degree=5, min_community=20, seed=10
6: (4)                  )
7: (4)                  partition = [{u} for u in G.nodes()]
8: (4)                  mod = nx.community.modularity(G, partition)
9: (4)                  partition = nx.community.louvain_communities(G)
10: (4)                 assert nx.community.modularity(G, partition) > mod
11: (0)             def test_valid_partition():
12: (4)                 G = nx.LFR_benchmark_graph(
13: (8)                     250, 3, 1.5, 0.009, average_degree=5, min_community=20, seed=10
14: (4)                 )
15: (4)                 H = G.to_directed()
16: (4)                 partition = nx.community.louvain_communities(G)
17: (4)                 partition2 = nx.community.louvain_communities(H)
18: (4)                 assert nx.community.is_partition(G, partition)
19: (4)                 assert nx.community.is_partition(H, partition2)
20: (0)             def test_karate_club_partition():
21: (4)                 G = nx.karate_club_graph()
22: (4)                 part = [
23: (8)                     {0, 1, 2, 3, 7, 9, 11, 12, 13, 17, 19, 21},
24: (8)                     {16, 4, 5, 6, 10},
25: (8)                     {23, 25, 27, 28, 24, 31},
26: (8)                     {32, 33, 8, 14, 15, 18, 20, 22, 26, 29, 30},
27: (4)                 ]
28: (4)                 partition = nx.community.louvain_communities(G, seed=2, weight=None)
29: (4)                 assert part == partition
30: (0)             def test_partition_iterator():
31: (4)                 G = nx.path_graph(15)
32: (4)                 parts_iter = nx.community.louvain_partitions(G, seed=42)
33: (4)                 first_part = next(parts_iter)
34: (4)                 first_copy = [s.copy() for s in first_part]
35: (4)                 assert first_copy[0] == first_part[0]
36: (4)                 second_part = next(parts_iter)
37: (4)                 assert first_copy[0] == first_part[0]
38: (0)             def test_undirected_selfloops():
39: (4)                 G = nx.karate_club_graph()
40: (4)                 expected_partition = nx.community.louvain_communities(G, seed=2,
weight=None)
41: (4)                 part = [
42: (8)                     {0, 1, 2, 3, 7, 9, 11, 12, 13, 17, 19, 21},
43: (8)                     {16, 4, 5, 6, 10},
44: (8)                     {23, 25, 27, 28, 24, 31},
45: (8)                     {32, 33, 8, 14, 15, 18, 20, 22, 26, 29, 30},
46: (4)                 ]
47: (4)                 assert expected_partition == part
48: (4)                 G.add_weighted_edges_from([(i, i, i * 1000) for i in range(9)])
49: (4)                 partition = nx.community.louvain_communities(G, seed=2, weight="weight")
50: (4)                 assert part != partition
51: (4)                 partition = nx.community.louvain_communities(G, seed=2, weight=None)
52: (4)                 assert part == partition
53: (0)             def test_directed_selfloops():
54: (4)                 G = nx.DiGraph()
55: (4)                 G.add_nodes_from(range(11))
56: (4)                 G_edges = [
57: (8)                     (0, 2),
58: (8)                     (0, 1),
59: (8)                     (1, 0),
60: (8)                     (2, 1),
61: (8)                     (2, 0),
62: (8)                     (3, 4),
63: (8)                     (4, 3),
64: (8)                     (7, 8),
65: (8)                     (8, 7),
66: (8)                     (9, 10),
67: (8)                     (10, 9),
68: (4)                 ]
69: (4)                 G.add_edges_from(G_edges)
```

```
70: (4)                 G_expected_partition = nx.community.louvain_communities(G, seed=123,
weight=None)
71: (4)                 G.add_weighted_edges_from([(i, i, i * 1000) for i in range(3)])
72: (4)                 G_partition = nx.community.louvain_communities(G, seed=123,
weight="weight")
73: (4)                 assert G_partition != G_expected_partition
74: (4)                 G_partition = nx.community.louvain_communities(G, seed=123, weight=None)
75: (4)                 assert G_partition == G_expected_partition
76: (0)             def test_directed_partition():
77: (4)                 """
78: (4)                 Test 2 cases that were looping infinitely
79: (4)                 from issues #5175 and #5704
80: (4)                 """
81: (4)                 G = nx.DiGraph()
82: (4)                 H = nx.DiGraph()
83: (4)                 G.add_nodes_from(range(10))
84: (4)                 H.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
85: (4)                 G_edges = [
86: (8)                     (0, 2),
87: (8)                     (0, 1),
88: (8)                     (1, 0),
89: (8)                     (2, 1),
90: (8)                     (2, 0),
91: (8)                     (3, 4),
92: (8)                     (4, 3),
93: (8)                     (7, 8),
94: (8)                     (8, 7),
95: (8)                     (9, 10),
96: (8)                     (10, 9),
97: (4)                 ]
98: (4)                 H_edges = [
99: (8)                     (1, 2),
100: (8)                    (1, 6),
101: (8)                    (1, 9),
102: (8)                    (2, 3),
103: (8)                    (2, 4),
104: (8)                    (2, 5),
105: (8)                    (3, 4),
106: (8)                    (4, 3),
107: (8)                    (4, 5),
108: (8)                    (5, 4),
109: (8)                    (6, 7),
110: (8)                    (6, 8),
111: (8)                    (9, 10),
112: (8)                    (9, 11),
113: (8)                    (10, 11),
114: (8)                    (11, 10),
115: (4)                ]
116: (4)                G.add_edges_from(G_edges)
117: (4)                H.add_edges_from(H_edges)
118: (4)                G_expected_partition = [{0, 1, 2}, {3, 4}, {5}, {6}, {8, 7}, {9, 10}]
119: (4)                G_partition = nx.community.louvain_communities(G, seed=123, weight=None)
120: (4)                H_expected_partition = [{2, 3, 4, 5}, {8, 1, 6, 7}, {9, 10, 11}]
121: (4)                H_partition = nx.community.louvain_communities(H, seed=123, weight=None)
122: (4)                assert G_partition == G_expected_partition
123: (4)                assert H_partition == H_expected_partition
124: (0)            def test_none_weight_param():
125: (4)                G = nx.karate_club_graph()
126: (4)                nx.set_edge_attributes(
127: (8)                    G, {edge: i * i for i, edge in enumerate(G.edges)}, name="foo"
128: (4)                )
129: (4)                part = [
130: (8)                    {0, 1, 2, 3, 7, 9, 11, 12, 13, 17, 19, 21},
131: (8)                    {16, 4, 5, 6, 10},
132: (8)                    {23, 25, 27, 28, 24, 31},
133: (8)                    {32, 33, 8, 14, 15, 18, 20, 22, 26, 29, 30},
134: (4)                ]
135: (4)                partition1 = nx.community.louvain_communities(G, weight=None, seed=2)
136: (4)                partition2 = nx.community.louvain_communities(G, weight="foo", seed=2)
```

```
137: (4)                partition3 = nx.community.louvain_communities(G, weight="weight", seed=2)
138: (4)                assert part == partition1
139: (4)                assert part != partition2
140: (4)                assert part != partition3
141: (4)                assert partition2 != partition3
142: (0)            def test_quality():
143: (4)                G = nx.LFR_benchmark_graph(
144: (8)                    250, 3, 1.5, 0.009, average_degree=5, min_community=20, seed=10
145: (4)                )
146: (4)                H = nx.gn_graph(200, seed=1234)
147: (4)                I = nx.MultiGraph(G)
148: (4)                J = nx.MultiDiGraph(H)
149: (4)                partition = nx.community.louvain_communities(G)
150: (4)                partition2 = nx.community.louvain_communities(H)
151: (4)                partition3 = nx.community.louvain_communities(I)
152: (4)                partition4 = nx.community.louvain_communities(J)
153: (4)                quality = nx.community.partition_quality(G, partition)[0]
154: (4)                quality2 = nx.community.partition_quality(H, partition2)[0]
155: (4)                quality3 = nx.community.partition_quality(I, partition3)[0]
156: (4)                quality4 = nx.community.partition_quality(J, partition4)[0]
157: (4)                assert quality >= 0.65
158: (4)                assert quality2 >= 0.65
159: (4)                assert quality3 >= 0.65
160: (4)                assert quality4 >= 0.65
161: (0)            def test_multigraph():
162: (4)                G = nx.karate_club_graph()
163: (4)                H = nx.MultiGraph(G)
164: (4)                G.add_edge(0, 1, weight=10)
165: (4)                H.add_edge(0, 1, weight=9)
166: (4)                G.add_edge(0, 9, foo=20)
167: (4)                H.add_edge(0, 9, foo=20)
168: (4)                partition1 = nx.community.louvain_communities(G, seed=1234)
169: (4)                partition2 = nx.community.louvain_communities(H, seed=1234)
170: (4)                partition3 = nx.community.louvain_communities(H, weight="foo", seed=1234)
171: (4)                assert partition1 == partition2 != partition3
172: (0)            def test_resolution():
173: (4)                G = nx.LFR_benchmark_graph(
174: (8)                    250, 3, 1.5, 0.009, average_degree=5, min_community=20, seed=10
175: (4)                )
176: (4)                partition1 = nx.community.louvain_communities(G, resolution=0.5, seed=12)
177: (4)                partition2 = nx.community.louvain_communities(G, seed=12)
178: (4)                partition3 = nx.community.louvain_communities(G, resolution=2, seed=12)
179: (4)                assert len(partition1) <= len(partition2) <= len(partition3)
180: (0)            def test_threshold():
181: (4)                G = nx.LFR_benchmark_graph(
182: (8)                    250, 3, 1.5, 0.009, average_degree=5, min_community=20, seed=10
183: (4)                )
184: (4)                partition1 = nx.community.louvain_communities(G, threshold=0.3, seed=2)
185: (4)                partition2 = nx.community.louvain_communities(G, seed=2)
186: (4)                mod1 = nx.community.modularity(G, partition1)
187: (4)                mod2 = nx.community.modularity(G, partition2)
188: (4)                assert mod1 <= mod2
189: (0)            def test_empty_graph():
190: (4)                G = nx.Graph()
191: (4)                G.add_nodes_from(range(5))
192: (4)                expected = [{0}, {1}, {2}, {3}, {4}]
193: (4)                assert nx.community.louvain_communities(G) == expected
194: (0)            def test_max_level():
195: (4)                G = nx.LFR_benchmark_graph(
196: (8)                    250, 3, 1.5, 0.009, average_degree=5, min_community=20, seed=10
197: (4)                )
198: (4)                parts_iter = nx.community.louvain_partitions(G, seed=42)
199: (4)                for max_level, expected in enumerate(parts_iter, 1):
200: (8)                    partition = nx.community.louvain_communities(G, max_level=max_level,
seed=42)
201: (8)                    assert partition == expected
202: (4)                assert max_level > 1  # Ensure we are actually testing max_level
203: (4)                partition = nx.community.louvain_communities(G, max_level=max_level + 1,
seed=42)
```

```
204: (4)                assert partition == expected
205: (4)                with pytest.raises(
206: (8)                    ValueError, match="max_level argument must be a positive integer"
207: (4)                ):
208: (8)                    nx.community.louvain_communities(G, max_level=0)
```

File 200 - test_quality.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.community.quality`
2: (0)              module.
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx import barbell_graph
7: (0)              from networkx.algorithms.community import modularity, partition_quality
8: (0)              from networkx.algorithms.community.quality import inter_community_edges
9: (0)              class TestPerformance:
10: (4)                 """Unit tests for the :func:`performance` function."""
11: (4)                 def test_bad_partition(self):
12: (8)                     """Tests that a poor partition has a low performance measure."""
13: (8)                     G = barbell_graph(3, 0)
14: (8)                     partition = [{0, 1, 4}, {2, 3, 5}]
15: (8)                     assert 8 / 15 == pytest.approx(partition_quality(G, partition)[1],
abs=1e-7)
16: (4)                 def test_good_partition(self):
17: (8)                     """Tests that a good partition has a high performance measure."""
18: (8)                     G = barbell_graph(3, 0)
19: (8)                     partition = [{0, 1, 2}, {3, 4, 5}]
20: (8)                     assert 14 / 15 == pytest.approx(partition_quality(G, partition)[1],
abs=1e-7)
21: (0)             class TestCoverage:
22: (4)                 """Unit tests for the :func:`coverage` function."""
23: (4)                 def test_bad_partition(self):
24: (8)                     """Tests that a poor partition has a low coverage measure."""
25: (8)                     G = barbell_graph(3, 0)
26: (8)                     partition = [{0, 1, 4}, {2, 3, 5}]
27: (8)                     assert 3 / 7 == pytest.approx(partition_quality(G, partition)[0],
abs=1e-7)
28: (4)                 def test_good_partition(self):
29: (8)                     """Tests that a good partition has a high coverage measure."""
30: (8)                     G = barbell_graph(3, 0)
31: (8)                     partition = [{0, 1, 2}, {3, 4, 5}]
32: (8)                     assert 6 / 7 == pytest.approx(partition_quality(G, partition)[0],
abs=1e-7)
33: (0)             def test_modularity():
34: (4)                 G = nx.barbell_graph(3, 0)
35: (4)                 C = [{0, 1, 4}, {2, 3, 5}]
36: (4)                 assert (-16 / (14**2)) == pytest.approx(modularity(G, C), abs=1e-7)
37: (4)                 C = [{0, 1, 2}, {3, 4, 5}]
38: (4)                 assert (35 * 2) / (14**2) == pytest.approx(modularity(G, C), abs=1e-7)
39: (4)                 n = 1000
40: (4)                 G = nx.erdos_renyi_graph(n, 0.09, seed=42, directed=True)
41: (4)                 C = [set(range(n // 2)), set(range(n // 2, n))]
42: (4)                 assert 0.00017154251389292754 == pytest.approx(modularity(G, C), abs=1e-7)
43: (4)                 G = nx.margulis_gabber_galil_graph(10)
44: (4)                 mid_value = G.number_of_nodes() // 2
45: (4)                 nodes = list(G.nodes)
46: (4)                 C = [set(nodes[:mid_value]), set(nodes[mid_value:])]
47: (4)                 assert 0.13 == pytest.approx(modularity(G, C), abs=1e-7)
48: (4)                 G = nx.DiGraph()
49: (4)                 G.add_edges_from([(2, 1), (2, 3), (3, 4)])
50: (4)                 C = [{1, 2}, {3, 4}]
51: (4)                 assert 2 / 9 == pytest.approx(modularity(G, C), abs=1e-7)
52: (0)             def test_modularity_resolution():
53: (4)                 G = nx.barbell_graph(3, 0)
54: (4)                 C = [{0, 1, 4}, {2, 3, 5}]
55: (4)                 assert modularity(G, C) == pytest.approx(3 / 7 - 100 / 14**2)
```

```
56: (4)                 gamma = 2
57: (4)                 result = modularity(G, C, resolution=gamma)
58: (4)                 assert result == pytest.approx(3 / 7 - gamma * 100 / 14**2)
59: (4)                 gamma = 0.2
60: (4)                 result = modularity(G, C, resolution=gamma)
61: (4)                 assert result == pytest.approx(3 / 7 - gamma * 100 / 14**2)
62: (4)                 C = [{0, 1, 2}, {3, 4, 5}]
63: (4)                 assert modularity(G, C) == pytest.approx(6 / 7 - 98 / 14**2)
64: (4)                 gamma = 2
65: (4)                 result = modularity(G, C, resolution=gamma)
66: (4)                 assert result == pytest.approx(6 / 7 - gamma * 98 / 14**2)
67: (4)                 gamma = 0.2
68: (4)                 result = modularity(G, C, resolution=gamma)
69: (4)                 assert result == pytest.approx(6 / 7 - gamma * 98 / 14**2)
70: (4)                 G = nx.barbell_graph(5, 3)
71: (4)                 C = [frozenset(range(5)), frozenset(range(8, 13)), frozenset(range(5, 8))]
72: (4)                 gamma = 1
73: (4)                 result = modularity(G, C, resolution=gamma)
74: (4)                 assert result == pytest.approx((22 / 24) - gamma * (918 / (48**2)))
75: (4)                 gamma = 2
76: (4)                 result = modularity(G, C, resolution=gamma)
77: (4)                 assert result == pytest.approx((22 / 24) - gamma * (918 / (48**2)))
78: (4)                 gamma = 0.2
79: (4)                 result = modularity(G, C, resolution=gamma)
80: (4)                 assert result == pytest.approx((22 / 24) - gamma * (918 / (48**2)))
81: (4)                 C = [{0, 1, 2, 3}, {9, 10, 11, 12}, {5, 6, 7}, {4}, {8}]
82: (4)                 gamma = 1
83: (4)                 result = modularity(G, C, resolution=gamma)
84: (4)                 assert result == pytest.approx((14 / 24) - gamma * (598 / (48**2)))
85: (4)                 gamma = 2.5
86: (4)                 result = modularity(G, C, resolution=gamma)
87: (4)                 assert result == pytest.approx((14 / 24) - gamma * (598 / (48**2)))
88: (4)                 gamma = 0.2
89: (4)                 result = modularity(G, C, resolution=gamma)
90: (4)                 assert result == pytest.approx((14 / 24) - gamma * (598 / (48**2)))
91: (4)                 C = [frozenset(range(8)), frozenset(range(8, 13))]
92: (4)                 gamma = 1
93: (4)                 result = modularity(G, C, resolution=gamma)
94: (4)                 assert result == pytest.approx((23 / 24) - gamma * (1170 / (48**2)))
95: (4)                 gamma = 2
96: (4)                 result = modularity(G, C, resolution=gamma)
97: (4)                 assert result == pytest.approx((23 / 24) - gamma * (1170 / (48**2)))
98: (4)                 gamma = 0.3
99: (4)                 result = modularity(G, C, resolution=gamma)
100: (4)                assert result == pytest.approx((23 / 24) - gamma * (1170 / (48**2)))
101: (0)            def test_inter_community_edges_with_digraphs():
102: (4)                G = nx.complete_graph(2, create_using=nx.DiGraph())
103: (4)                partition = [{0}, {1}]
104: (4)                assert inter_community_edges(G, partition) == 2
105: (4)                G = nx.complete_graph(10, create_using=nx.DiGraph())
106: (4)                partition = [{0}, {1, 2}, {3, 4, 5}, {6, 7, 8, 9}]
107: (4)                assert inter_community_edges(G, partition) == 70
108: (4)                G = nx.cycle_graph(4, create_using=nx.DiGraph())
109: (4)                partition = [{0, 1}, {2, 3}]
110: (4)                assert inter_community_edges(G, partition) == 2

----------------------------------------

File 201 - test_modularity_max.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.community import (
4: (4)                  greedy_modularity_communities,
5: (4)                  naive_greedy_modularity_communities,
6: (0)              )
7: (0)              @pytest.mark.parametrize(
8: (4)                  "func", (greedy_modularity_communities,
naive_greedy_modularity_communities)
```

```
9: (0)              )
10: (0)             def test_modularity_communities(func):
11: (4)                 G = nx.karate_club_graph()
12: (4)                 john_a = frozenset(
13: (8)                     [8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]
14: (4)                 )
15: (4)                 mr_hi = frozenset([0, 4, 5, 6, 10, 11, 16, 19])
16: (4)                 overlap = frozenset([1, 2, 3, 7, 9, 12, 13, 17, 21])
17: (4)                 expected = {john_a, overlap, mr_hi}
18: (4)                 assert set(func(G, weight=None)) == expected
19: (0)             @pytest.mark.parametrize(
20: (4)                 "func", (greedy_modularity_communities,
naive_greedy_modularity_communities)
21: (0)             )
22: (0)             def test_modularity_communities_categorical_labels(func):
23: (4)                 G = nx.Graph(
24: (8)                     [
25: (12)                        ("a", "b"),
26: (12)                        ("a", "c"),
27: (12)                        ("b", "c"),
28: (12)                        ("b", "d"),  # inter-community edge
29: (12)                        ("d", "e"),
30: (12)                        ("d", "f"),
31: (12)                        ("d", "g"),
32: (12)                        ("f", "g"),
33: (12)                        ("d", "e"),
34: (12)                        ("f", "e"),
35: (8)                     ]
36: (4)                 )
37: (4)                 expected = {frozenset({"f", "g", "e", "d"}), frozenset({"a", "b", "c"})}
38: (4)                 assert set(func(G)) == expected
39: (0)             def test_greedy_modularity_communities_components():
40: (4)                 G = nx.Graph([(0, 1), (2, 3), (4, 5), (5, 6)])
41: (4)                 assert greedy_modularity_communities(G) == [{4, 5, 6}, {0, 1}, {2, 3}]
42: (4)                 assert greedy_modularity_communities(G, best_n=3) == [{4, 5, 6}, {0, 1},
{2, 3}]
43: (4)                 assert greedy_modularity_communities(G, best_n=2) == [{0, 1, 4, 5, 6}, {2,
3}]
44: (4)                 assert greedy_modularity_communities(G, best_n=1) == [{0, 1, 2, 3, 4, 5,
6}]
45: (0)             def test_greedy_modularity_communities_relabeled():
46: (4)                 G = nx.balanced_tree(2, 2)
47: (4)                 mapping = {0: "a", 1: "b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g", 7: "h"}
48: (4)                 G = nx.relabel_nodes(G, mapping)
49: (4)                 expected = [frozenset({"e", "d", "a", "b"}), frozenset({"c", "f", "g"})]
50: (4)                 assert greedy_modularity_communities(G) == expected
51: (0)             def test_greedy_modularity_communities_directed():
52: (4)                 G = nx.DiGraph(
53: (8)                     [
54: (12)                        ("a", "b"),
55: (12)                        ("a", "c"),
56: (12)                        ("b", "c"),
57: (12)                        ("b", "d"),  # inter-community edge
58: (12)                        ("d", "e"),
59: (12)                        ("d", "f"),
60: (12)                        ("d", "g"),
61: (12)                        ("f", "g"),
62: (12)                        ("d", "e"),
63: (12)                        ("f", "e"),
64: (8)                     ]
65: (4)                 )
66: (4)                 expected = [frozenset({"f", "g", "e", "d"}), frozenset({"a", "b", "c"})]
67: (4)                 assert greedy_modularity_communities(G) == expected
68: (4)                 G = nx.DiGraph()
69: (4)                 G.add_edges_from(
70: (8)                     [(1, 1), (1, 2), (1, 3), (2, 3), (1, 4), (4, 4), (5, 5), (4, 5), (4,
6), (5, 6)]
71: (4)                 )
72: (4)                 expected = [frozenset({1, 2, 3}), frozenset({4, 5, 6})]
```

```
(4)                 assert greedy_modularity_communities(G) == expected
(0)             @pytest.mark.parametrize(
(4)                 "func", (greedy_modularity_communities,
naive_greedy_modularity_communities)
(0)             )
(0)             def test_modularity_communities_weighted(func):
(4)                 G = nx.balanced_tree(2, 3)
(4)                 for a, b in G.edges:
(8)                     if ((a == 1) or (a == 2)) and (b != 0):
(12)                        G[a][b]["weight"] = 10.0
(8)                     else:
(12)                        G[a][b]["weight"] = 1.0
(4)                 expected = [{0, 1, 3, 4, 7, 8, 9, 10}, {2, 5, 6, 11, 12, 13, 14}]
(4)                 assert func(G, weight="weight") == expected
(4)                 assert func(G, weight="weight", resolution=0.9) == expected
(4)                 assert func(G, weight="weight", resolution=0.3) == expected
(4)                 assert func(G, weight="weight", resolution=1.1) != expected
(0)             def test_modularity_communities_floating_point():
(4)                 G = nx.Graph()
(4)                 G.add_weighted_edges_from(
(8)                     [(0, 1, 12), (1, 4, 71), (2, 3, 15), (2, 4, 10), (3, 6, 13)]
(4)                 )
(4)                 expected = [{0, 1, 4}, {2, 3, 6}]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(4)                 assert (
(8)                     greedy_modularity_communities(G, weight="weight", resolution=0.99) ==
expected
(4)                 )
(0)             def test_modularity_communities_directed_weighted():
(4)                 G = nx.DiGraph()
(4)                 G.add_weighted_edges_from(
(8)                     [
(12)                        (1, 2, 5),
(12)                        (1, 3, 3),
(12)                        (2, 3, 6),
(12)                        (2, 6, 1),
(12)                        (1, 4, 1),
(12)                        (4, 5, 3),
(12)                        (4, 6, 7),
(12)                        (5, 6, 2),
(12)                        (5, 7, 5),
(12)                        (5, 8, 4),
(12)                        (6, 8, 3),
(8)                     ]
(4)                 )
(4)                 expected = [frozenset({4, 5, 6, 7, 8}), frozenset({1, 2, 3})]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(4)                 G[2][6]["weight"] = 20
(4)                 expected = [frozenset({1, 2, 3, 6}), frozenset({4, 5, 7, 8})]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(0)             def test_greedy_modularity_communities_multigraph():
(4)                 G = nx.MultiGraph()
(4)                 G.add_edges_from(
(8)                     [
(12)                        (1, 2),
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (2, 3),
(12)                        (1, 4),
(12)                        (2, 4),
(12)                        (4, 5),
(12)                        (5, 6),
(12)                        (5, 7),
(12)                        (5, 7),
(12)                        (6, 7),
(12)                        (7, 8),
(12)                        (5, 8),
(8)                     ]
(4)                 )
```

```
(4)                 expected = [frozenset({1, 2, 3, 4}), frozenset({5, 6, 7, 8})]
(4)                 assert greedy_modularity_communities(G) == expected
(4)                 G.add_edge(4, 5)
(4)                 expected = [frozenset({4, 5, 6, 7, 8}), frozenset({1, 2, 3})]
(4)                 assert greedy_modularity_communities(G) == expected
(0)             def test_greedy_modularity_communities_multigraph_weighted():
(4)                 G = nx.MultiGraph()
(4)                 G.add_weighted_edges_from(
(8)                     [
(12)                        (1, 2, 5),
(12)                        (1, 2, 3),
(12)                        (1, 3, 6),
(12)                        (1, 3, 6),
(12)                        (2, 3, 4),
(12)                        (1, 4, 1),
(12)                        (1, 4, 1),
(12)                        (2, 4, 3),
(12)                        (2, 4, 3),
(12)                        (4, 5, 1),
(12)                        (5, 6, 3),
(12)                        (5, 6, 7),
(12)                        (5, 6, 4),
(12)                        (5, 7, 9),
(12)                        (5, 7, 9),
(12)                        (6, 7, 8),
(12)                        (7, 8, 2),
(12)                        (7, 8, 2),
(12)                        (5, 8, 6),
(12)                        (5, 8, 6),
(8)                     ]
(4)                 )
(4)                 expected = [frozenset({1, 2, 3, 4}), frozenset({5, 6, 7, 8})]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(4)                 G.add_edge(4, 5, weight=16)
(4)                 expected = [frozenset({4, 5, 6, 7, 8}), frozenset({1, 2, 3})]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(4)                 G[1][4][1]["weight"] = 3
(4)                 expected = [frozenset({1, 2, 3, 4}), frozenset({5, 6, 7, 8})]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(0)             def test_greed_modularity_communities_multidigraph():
(4)                 G = nx.MultiDiGraph()
(4)                 G.add_edges_from(
(8)                     [
(12)                        (1, 2),
(12)                        (1, 2),
(12)                        (3, 1),
(12)                        (2, 3),
(12)                        (2, 3),
(12)                        (3, 2),
(12)                        (1, 4),
(12)                        (2, 4),
(12)                        (4, 2),
(12)                        (4, 5),
(12)                        (5, 6),
(12)                        (5, 6),
(12)                        (6, 5),
(12)                        (5, 7),
(12)                        (6, 7),
(12)                        (7, 8),
(12)                        (5, 8),
(12)                        (8, 4),
(8)                     ]
(4)                 )
(4)                 expected = [frozenset({1, 2, 3, 4}), frozenset({5, 6, 7, 8})]
(4)                 assert greedy_modularity_communities(G, weight="weight") == expected
(0)             def test_greed_modularity_communities_multidigraph_weighted():
(4)                 G = nx.MultiDiGraph()
(4)                 G.add_weighted_edges_from(
(8)                     [
```

```
209: (12)                        (1, 2, 5),
210: (12)                        (1, 2, 3),
211: (12)                        (3, 1, 6),
212: (12)                        (1, 3, 6),
213: (12)                        (3, 2, 4),
214: (12)                        (1, 4, 2),
215: (12)                        (1, 4, 5),
216: (12)                        (2, 4, 3),
217: (12)                        (3, 2, 8),
218: (12)                        (4, 2, 3),
219: (12)                        (4, 3, 5),
220: (12)                        (4, 5, 2),
221: (12)                        (5, 6, 3),
222: (12)                        (5, 6, 7),
223: (12)                        (6, 5, 4),
224: (12)                        (5, 7, 9),
225: (12)                        (5, 7, 9),
226: (12)                        (7, 6, 8),
227: (12)                        (7, 8, 2),
228: (12)                        (8, 7, 2),
229: (12)                        (5, 8, 6),
230: (12)                        (5, 8, 6),
231: (8)                     ]
232: (4)                 )
233: (4)                 expected = [frozenset({1, 2, 3, 4}), frozenset({5, 6, 7, 8})]
234: (4)                 assert greedy_modularity_communities(G, weight="weight") == expected
235: (0)             def test_resolution_parameter_impact():
236: (4)                 G = nx.barbell_graph(5, 3)
237: (4)                 gamma = 1
238: (4)                 expected = [frozenset(range(5)), frozenset(range(8, 13)),
frozenset(range(5, 8))]
239: (4)                 assert greedy_modularity_communities(G, resolution=gamma) == expected
240: (4)                 assert naive_greedy_modularity_communities(G, resolution=gamma) ==
expected
241: (4)                 gamma = 2.5
242: (4)                 expected = [{0, 1, 2, 3}, {9, 10, 11, 12}, {5, 6, 7}, {4}, {8}]
243: (4)                 assert greedy_modularity_communities(G, resolution=gamma) == expected
244: (4)                 assert naive_greedy_modularity_communities(G, resolution=gamma) ==
expected
245: (4)                 gamma = 0.3
246: (4)                 expected = [frozenset(range(8)), frozenset(range(8, 13))]
247: (4)                 assert greedy_modularity_communities(G, resolution=gamma) == expected
248: (4)                 assert naive_greedy_modularity_communities(G, resolution=gamma) ==
expected
249: (0)             def test_cutoff_parameter():
250: (4)                 G = nx.circular_ladder_graph(4)
251: (4)                 expected = [{k} for k in range(8)]
252: (4)                 assert greedy_modularity_communities(G, cutoff=8) == expected
253: (4)                 expected = [{k, k + 1} for k in range(0, 8, 2)]
254: (4)                 assert greedy_modularity_communities(G, cutoff=4) == expected
255: (4)                 expected = [frozenset(range(4)), frozenset(range(4, 8))]
256: (4)                 assert greedy_modularity_communities(G, cutoff=1) == expected
257: (0)             def test_best_n():
258: (4)                 G = nx.barbell_graph(5, 3)
259: (4)                 best_n = 3
260: (4)                 expected = [frozenset(range(5)), frozenset(range(8, 13)),
frozenset(range(5, 8))]
261: (4)                 assert greedy_modularity_communities(G, best_n=best_n) == expected
262: (4)                 best_n = 2
263: (4)                 expected = [frozenset(range(8)), frozenset(range(8, 13))]
264: (4)                 assert greedy_modularity_communities(G, best_n=best_n) == expected
265: (4)                 best_n = 1
266: (4)                 expected = [frozenset(range(13))]
267: (4)                 assert greedy_modularity_communities(G, best_n=best_n) == expected
268: (0)             def test_greedy_modularity_communities_corner_cases():
269: (4)                 G = nx.empty_graph()
270: (4)                 assert nx.community.greedy_modularity_communities(G) == []
271: (4)                 G.add_nodes_from(range(3))
272: (4)                 assert nx.community.greedy_modularity_communities(G) == [{0}, {1}, {2}]
```

```
----------------------------------------

File 202 - __init__.py:

1: (0)              from .connected import *
2: (0)              from .strongly_connected import *
3: (0)              from .weakly_connected import *
4: (0)              from .attracting import *
5: (0)              from .biconnected import *
6: (0)              from .semiconnected import *

----------------------------------------

File 203 - test_utils.py:

1: (0)              """Unit tests for the :mod:`networkx.algorithms.community.utils` module.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              def test_is_partition():
5: (4)                  G = nx.empty_graph(3)
6: (4)                  assert nx.community.is_partition(G, [{0, 1}, {2}])
7: (4)                  assert nx.community.is_partition(G, ({0, 1}, {2}))
8: (4)                  assert nx.community.is_partition(G, ([0, 1], [2]))
9: (4)                  assert nx.community.is_partition(G, [[0, 1], [2]])
10: (0)             def test_not_covering():
11: (4)                 G = nx.empty_graph(3)
12: (4)                 assert not nx.community.is_partition(G, [{0}, {1}])
13: (0)             def test_not_disjoint():
14: (4)                 G = nx.empty_graph(3)
15: (4)                 assert not nx.community.is_partition(G, [{0, 1}, {1, 2}])
16: (0)             def test_not_node():
17: (4)                 G = nx.empty_graph(3)
18: (4)                 assert not nx.community.is_partition(G, [{0, 1}, {3}])

----------------------------------------

File 204 - attracting.py:

1: (0)              """Attracting components."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils.decorators import not_implemented_for
4: (0)              __all__ = [
5: (4)                  "number_attracting_components",
6: (4)                  "attracting_components",
7: (4)                  "is_attracting_component",
8: (0)              ]
9: (0)              @not_implemented_for("undirected")
10: (0)             @nx._dispatchable
11: (0)             def attracting_components(G):
12: (4)                 """Generates the attracting components in `G`.
13: (4)                 An attracting component in a directed graph `G` is a strongly connected
14: (4)                 component with the property that a random walker on the graph will never
15: (4)                 leave the component, once it enters the component.
16: (4)                 The nodes in attracting components can also be thought of as recurrent
17: (4)                 nodes.  If a random walker enters the attractor containing the node, then
18: (4)                 the node will be visited infinitely often.
19: (4)                 To obtain induced subgraphs on each component use:
20: (4)                 ``(G.subgraph(c).copy() for c in attracting_components(G))``
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : DiGraph, MultiDiGraph
24: (8)                     The graph to be analyzed.
25: (4)                 Returns
26: (4)                 -------
27: (4)                 attractors : generator of sets
28: (8)                     A generator of sets of nodes, one for each attracting component of G.
29: (4)                 Raises
30: (4)                 ------
```

```
31: (4)                 NetworkXNotImplemented
32: (8)                     If the input graph is undirected.
33: (4)                 See Also
34: (4)                 --------
35: (4)                 number_attracting_components
36: (4)                 is_attracting_component
37: (4)                 """
38: (4)                 scc = list(nx.strongly_connected_components(G))
39: (4)                 cG = nx.condensation(G, scc)
40: (4)                 for n in cG:
41: (8)                     if cG.out_degree(n) == 0:
42: (12)                        yield scc[n]
43: (0)             @not_implemented_for("undirected")
44: (0)             @nx._dispatchable
45: (0)             def number_attracting_components(G):
46: (4)                 """Returns the number of attracting components in `G`.
47: (4)                 Parameters
48: (4)                 ----------
49: (4)                 G : DiGraph, MultiDiGraph
50: (8)                     The graph to be analyzed.
51: (4)                 Returns
52: (4)                 -------
53: (4)                 n : int
54: (8)                     The number of attracting components in G.
55: (4)                 Raises
56: (4)                 ------
57: (4)                 NetworkXNotImplemented
58: (8)                     If the input graph is undirected.
59: (4)                 See Also
60: (4)                 --------
61: (4)                 attracting_components
62: (4)                 is_attracting_component
63: (4)                 """
64: (4)                 return sum(1 for ac in attracting_components(G))
65: (0)             @not_implemented_for("undirected")
66: (0)             @nx._dispatchable
67: (0)             def is_attracting_component(G):
68: (4)                 """Returns True if `G` consists of a single attracting component.
69: (4)                 Parameters
70: (4)                 ----------
71: (4)                 G : DiGraph, MultiDiGraph
72: (8)                     The graph to be analyzed.
73: (4)                 Returns
74: (4)                 -------
75: (4)                 attracting : bool
76: (8)                     True if `G` has a single attracting component. Otherwise, False.
77: (4)                 Raises
78: (4)                 ------
79: (4)                 NetworkXNotImplemented
80: (8)                     If the input graph is undirected.
81: (4)                 See Also
82: (4)                 --------
83: (4)                 attracting_components
84: (4)                 number_attracting_components
85: (4)                 """
86: (4)                 ac = list(attracting_components(G))
87: (4)                 if len(ac) == 1:
88: (8)                     return len(ac[0]) == len(G)
89: (4)                 return False
```

File 205 - biconnected.py:

```
1: (0)              """Biconnected components and articulation points."""
2: (0)              from itertools import chain
3: (0)              import networkx as nx
4: (0)              from networkx.utils.decorators import not_implemented_for
5: (0)              __all__ = [
```

```
 6: (4)                 "biconnected_components",
 7: (4)                 "biconnected_component_edges",
 8: (4)                 "is_biconnected",
 9: (4)                 "articulation_points",
10: (0)             ]
11: (0)             @not_implemented_for("directed")
12: (0)             @nx._dispatchable
13: (0)             def is_biconnected(G):
14: (4)                 """Returns True if the graph is biconnected, False otherwise.
15: (4)                 A graph is biconnected if, and only if, it cannot be disconnected by
16: (4)                 removing only one node (and all edges incident on that node).  If
17: (4)                 removing a node increases the number of disconnected components
18: (4)                 in the graph, that node is called an articulation point, or cut
19: (4)                 vertex.  A biconnected graph has no articulation points.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : NetworkX Graph
23: (8)                     An undirected graph.
24: (4)                 Returns
25: (4)                 -------
26: (4)                 biconnected : bool
27: (8)                     True if the graph is biconnected, False otherwise.
28: (4)                 Raises
29: (4)                 ------
30: (4)                 NetworkXNotImplemented
31: (8)                     If the input graph is not undirected.
32: (4)                 Examples
33: (4)                 --------
34: (4)                 >>> G = nx.path_graph(4)
35: (4)                 >>> print(nx.is_biconnected(G))
36: (4)                 False
37: (4)                 >>> G.add_edge(0, 3)
38: (4)                 >>> print(nx.is_biconnected(G))
39: (4)                 True
40: (4)                 See Also
41: (4)                 --------
42: (4)                 biconnected_components
43: (4)                 articulation_points
44: (4)                 biconnected_component_edges
45: (4)                 is_strongly_connected
46: (4)                 is_weakly_connected
47: (4)                 is_connected
48: (4)                 is_semiconnected
49: (4)                 Notes
50: (4)                 -----
51: (4)                 The algorithm to find articulation points and biconnected
52: (4)                 components is implemented using a non-recursive depth-first-search
53: (4)                 (DFS) that keeps track of the highest level that back edges reach
54: (4)                 in the DFS tree.  A node `n` is an articulation point if, and only
55: (4)                 if, there exists a subtree rooted at `n` such that there is no
56: (4)                 back edge from any successor of `n` that links to a predecessor of
57: (4)                 `n` in the DFS tree.  By keeping track of all the edges traversed
58: (4)                 by the DFS we can obtain the biconnected components because all
59: (4)                 edges of a bicomponent will be traversed consecutively between
60: (4)                 articulation points.
61: (4)                 References
62: (4)                 ----------
63: (4)                 .. [1] Hopcroft, J.; Tarjan, R. (1973).
64: (7)                    "Efficient algorithms for graph manipulation".
65: (7)                    Communications of the ACM 16: 372–378. doi:10.1145/362248.362272
66: (4)                 """
67: (4)                 bccs = biconnected_components(G)
68: (4)                 try:
69: (8)                     bcc = next(bccs)
70: (4)                 except StopIteration:
71: (8)                     return False
72: (4)                 try:
73: (8)                     next(bccs)
74: (4)                 except StopIteration:
```

```
 75: (8)                    return len(bcc) == len(G)
 76: (4)                else:
 77: (8)                    return False
 78: (0)            @not_implemented_for("directed")
 79: (0)            @nx._dispatchable
 80: (0)            def biconnected_component_edges(G):
 81: (4)                """Returns a generator of lists of edges, one list for each biconnected
 82: (4)                component of the input graph.
 83: (4)                Biconnected components are maximal subgraphs such that the removal of a
 84: (4)                node (and all edges incident on that node) will not disconnect the
 85: (4)                subgraph.  Note that nodes may be part of more than one biconnected
 86: (4)                component.  Those nodes are articulation points, or cut vertices.
 87: (4)                However, each edge belongs to one, and only one, biconnected component.
 88: (4)                Notice that by convention a dyad is considered a biconnected component.
 89: (4)                Parameters
 90: (4)                ----------
 91: (4)                G : NetworkX Graph
 92: (8)                    An undirected graph.
 93: (4)                Returns
 94: (4)                -------
 95: (4)                edges : generator of lists
 96: (8)                    Generator of lists of edges, one list for each bicomponent.
 97: (4)                Raises
 98: (4)                ------
 99: (4)                NetworkXNotImplemented
100: (8)                    If the input graph is not undirected.
101: (4)                Examples
102: (4)                --------
103: (4)                >>> G = nx.barbell_graph(4, 2)
104: (4)                >>> print(nx.is_biconnected(G))
105: (4)                False
106: (4)                >>> bicomponents_edges = list(nx.biconnected_component_edges(G))
107: (4)                >>> len(bicomponents_edges)
108: (4)                5
109: (4)                >>> G.add_edge(2, 8)
110: (4)                >>> print(nx.is_biconnected(G))
111: (4)                True
112: (4)                >>> bicomponents_edges = list(nx.biconnected_component_edges(G))
113: (4)                >>> len(bicomponents_edges)
114: (4)                1
115: (4)                See Also
116: (4)                --------
117: (4)                is_biconnected,
118: (4)                biconnected_components,
119: (4)                articulation_points,
120: (4)                Notes
121: (4)                -----
122: (4)                The algorithm to find articulation points and biconnected
123: (4)                components is implemented using a non-recursive depth-first-search
124: (4)                (DFS) that keeps track of the highest level that back edges reach
125: (4)                in the DFS tree.  A node `n` is an articulation point if, and only
126: (4)                if, there exists a subtree rooted at `n` such that there is no
127: (4)                back edge from any successor of `n` that links to a predecessor of
128: (4)                `n` in the DFS tree.  By keeping track of all the edges traversed
129: (4)                by the DFS we can obtain the biconnected components because all
130: (4)                edges of a bicomponent will be traversed consecutively between
131: (4)                articulation points.
132: (4)                References
133: (4)                ----------
134: (4)                .. [1] Hopcroft, J.; Tarjan, R. (1973).
135: (11)                      "Efficient algorithms for graph manipulation".
136: (11)                      Communications of the ACM 16: 372–378. doi:10.1145/362248.362272
137: (4)                """
138: (4)                yield from _biconnected_dfs(G, components=True)
139: (0)            @not_implemented_for("directed")
140: (0)            @nx._dispatchable
141: (0)            def biconnected_components(G):
142: (4)                """Returns a generator of sets of nodes, one set for each biconnected
143: (4)                component of the graph
```

```
    Biconnected components are maximal subgraphs such that the removal of a
    node (and all edges incident on that node) will not disconnect the
    subgraph. Note that nodes may be part of more than one biconnected
    component.  Those nodes are articulation points, or cut vertices.  The
    removal of articulation points will increase the number of connected
    components of the graph.
    Notice that by convention a dyad is considered a biconnected component.
    Parameters
    ----------
    G : NetworkX Graph
        An undirected graph.
    Returns
    -------
    nodes : generator
        Generator of sets of nodes, one set for each biconnected component.
    Raises
    ------
    NetworkXNotImplemented
        If the input graph is not undirected.
    Examples
    --------
    >>> G = nx.lollipop_graph(5, 1)
    >>> print(nx.is_biconnected(G))
    False
    >>> bicomponents = list(nx.biconnected_components(G))
    >>> len(bicomponents)
    2
    >>> G.add_edge(0, 5)
    >>> print(nx.is_biconnected(G))
    True
    >>> bicomponents = list(nx.biconnected_components(G))
    >>> len(bicomponents)
    1
    You can generate a sorted list of biconnected components, largest
    first, using sort.
    >>> G.remove_edge(0, 5)
    >>> [len(c) for c in sorted(nx.biconnected_components(G), key=len, reverse=True)]
    [5, 2]
    If you only want the largest connected component, it's more
    efficient to use max instead of sort.
    >>> Gc = max(nx.biconnected_components(G), key=len)
    To create the components as subgraphs use:
    ``(G.subgraph(c).copy() for c in biconnected_components(G))``
    See Also
    --------
    is_biconnected
    articulation_points
    biconnected_component_edges
    k_components : this function is a special case where k=2
    bridge_components : similar to this function, but is defined using
        2-edge-connectivity instead of 2-node-connectivity.
    Notes
    -----
    The algorithm to find articulation points and biconnected
    components is implemented using a non-recursive depth-first-search
    (DFS) that keeps track of the highest level that back edges reach
    in the DFS tree.  A node `n` is an articulation point if, and only
    if, there exists a subtree rooted at `n` such that there is no
    back edge from any successor of `n` that links to a predecessor of
    `n` in the DFS tree.  By keeping track of all the edges traversed
    by the DFS we can obtain the biconnected components because all
    edges of a bicomponent will be traversed consecutively between
    articulation points.
    References
    ----------
    .. [1] Hopcroft, J.; Tarjan, R. (1973).
           "Efficient algorithms for graph manipulation".
           Communications of the ACM 16: 372–378. doi:10.1145/362248.362272
```

```
    """
    for comp in _biconnected_dfs(G, components=True):
        yield set(chain.from_iterable(comp))
@not_implemented_for("directed")
@nx._dispatchable
def articulation_points(G):
    """Yield the articulation points, or cut vertices, of a graph.
    An articulation point or cut vertex is any node whose removal (along with
    all its incident edges) increases the number of connected components of
    a graph.  An undirected connected graph without articulation points is
    biconnected. Articulation points belong to more than one biconnected
    component of a graph.
    Notice that by convention a dyad is considered a biconnected component.
    Parameters
    ----------
    G : NetworkX Graph
        An undirected graph.
    Yields
    ------
    node
        An articulation point in the graph.
    Raises
    ------
    NetworkXNotImplemented
        If the input graph is not undirected.
    Examples
    --------
    >>> G = nx.barbell_graph(4, 2)
    >>> print(nx.is_biconnected(G))
    False
    >>> len(list(nx.articulation_points(G)))
    4
    >>> G.add_edge(2, 8)
    >>> print(nx.is_biconnected(G))
    True
    >>> len(list(nx.articulation_points(G)))
    0
    See Also
    --------
    is_biconnected
    biconnected_components
    biconnected_component_edges
    Notes
    -----
    The algorithm to find articulation points and biconnected
    components is implemented using a non-recursive depth-first-search
    (DFS) that keeps track of the highest level that back edges reach
    in the DFS tree.  A node `n` is an articulation point if, and only
    if, there exists a subtree rooted at `n` such that there is no
    back edge from any successor of `n` that links to a predecessor of
    `n` in the DFS tree.  By keeping track of all the edges traversed
    by the DFS we can obtain the biconnected components because all
    edges of a bicomponent will be traversed consecutively between
    articulation points.
    References
    ----------
    .. [1] Hopcroft, J.; Tarjan, R. (1973).
           "Efficient algorithms for graph manipulation".
           Communications of the ACM 16: 372–378. doi:10.1145/362248.362272
    """
    seen = set()
    for articulation in _biconnected_dfs(G, components=False):
        if articulation not in seen:
            seen.add(articulation)
            yield articulation
@not_implemented_for("directed")
def _biconnected_dfs(G, components=True):
    visited = set()
    for start in G:
```

```
281: (8)                    if start in visited:
282: (12)                       continue
283: (8)                    discovery = {start: 0}  # time of first discovery of node during
search
284: (8)                    low = {start: 0}
285: (8)                    root_children = 0
286: (8)                    visited.add(start)
287: (8)                    edge_stack = []
288: (8)                    stack = [(start, start, iter(G[start]))]
289: (8)                    edge_index = {}
290: (8)                    while stack:
291: (12)                       grandparent, parent, children = stack[-1]
292: (12)                       try:
293: (16)                           child = next(children)
294: (16)                           if grandparent == child:
295: (20)                               continue
296: (16)                           if child in visited:
297: (20)                               if discovery[child] <= discovery[parent]:  # back edge
298: (24)                                   low[parent] = min(low[parent], discovery[child])
299: (24)                                   if components:
300: (28)                                       edge_index[parent, child] = len(edge_stack)
301: (28)                                       edge_stack.append((parent, child))
302: (16)                           else:
303: (20)                               low[child] = discovery[child] = len(discovery)
304: (20)                               visited.add(child)
305: (20)                               stack.append((parent, child, iter(G[child])))
306: (20)                               if components:
307: (24)                                   edge_index[parent, child] = len(edge_stack)
308: (24)                                   edge_stack.append((parent, child))
309: (12)                       except StopIteration:
310: (16)                           stack.pop()
311: (16)                           if len(stack) > 1:
312: (20)                               if low[parent] >= discovery[grandparent]:
313: (24)                                   if components:
314: (28)                                       ind = edge_index[grandparent, parent]
315: (28)                                       yield edge_stack[ind:]
316: (28)                                       del edge_stack[ind:]
317: (24)                                   else:
318: (28)                                       yield grandparent
319: (20)                               low[grandparent] = min(low[parent], low[grandparent])
320: (16)                           elif stack:  # length 1 so grandparent is root
321: (20)                               root_children += 1
322: (20)                               if components:
323: (24)                                   ind = edge_index[grandparent, parent]
324: (24)                                   yield edge_stack[ind:]
325: (24)                                   del edge_stack[ind:]
326: (8)                    if not components:
327: (12)                       if root_children > 1:
328: (16)                           yield start
```

File 206 - connected.py:

```
1: (0)              """Connected components."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils.decorators import not_implemented_for
4: (0)              from ...utils import arbitrary_element
5: (0)              __all__ = [
6: (4)                  "number_connected_components",
7: (4)                  "connected_components",
8: (4)                  "is_connected",
9: (4)                  "node_connected_component",
10: (0)             ]
11: (0)             @not_implemented_for("directed")
12: (0)             @nx._dispatchable
13: (0)             def connected_components(G):
14: (4)                 """Generate connected components.
15: (4)                 Parameters
```

```
    ----------
    G : NetworkX graph
       An undirected graph
    Returns
    -------
    comp : generator of sets
       A generator of sets of nodes, one for each component of G.
    Raises
    ------
    NetworkXNotImplemented
        If G is directed.
    Examples
    --------
    Generate a sorted list of connected components, largest first.
    >>> G = nx.path_graph(4)
    >>> nx.add_path(G, [10, 11, 12])
    >>> [len(c) for c in sorted(nx.connected_components(G), key=len, reverse=True)]
    [4, 3]
    If you only want the largest connected component, it's more
    efficient to use max instead of sort.
    >>> largest_cc = max(nx.connected_components(G), key=len)
    To create the induced subgraph of each component use:
    >>> S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
    See Also
    --------
    strongly_connected_components
    weakly_connected_components
    Notes
    -----
    For undirected graphs only.
    """
    seen = set()
    for v in G:
        if v not in seen:
            c = _plain_bfs(G, v)
            seen.update(c)
            yield c
@not_implemented_for("directed")
@nx._dispatchable
def number_connected_components(G):
    """Returns the number of connected components.
    Parameters
    ----------
    G : NetworkX graph
       An undirected graph.
    Returns
    -------
    n : integer
       Number of connected components
    Raises
    ------
    NetworkXNotImplemented
        If G is directed.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (1, 2), (5, 6), (3, 4)])
    >>> nx.number_connected_components(G)
    3
    See Also
    --------
    connected_components
    number_weakly_connected_components
    number_strongly_connected_components
    Notes
    -----
    For undirected graphs only.
    """
    return sum(1 for cc in connected_components(G))
```

```
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable
(0)             def is_connected(G):
(4)                 """Returns True if the graph is connected, False otherwise.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(7)                    An undirected graph.
(4)                 Returns
(4)                 -------
(4)                 connected : bool
(6)                   True if the graph is connected, false otherwise.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If G is directed.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> print(nx.is_connected(G))
(4)                 True
(4)                 See Also
(4)                 --------
(4)                 is_strongly_connected
(4)                 is_weakly_connected
(4)                 is_semiconnected
(4)                 is_biconnected
(4)                 connected_components
(4)                 Notes
(4)                 -----
(4)                 For undirected graphs only.
(4)                 """
(4)                 if len(G) == 0:
(8)                     raise nx.NetworkXPointlessConcept(
(12)                        "Connectivity is undefined for the null graph."
(8)                     )
(4)                 return sum(1 for node in _plain_bfs(G, arbitrary_element(G))) == len(G)
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable
(0)             def node_connected_component(G, n):
(4)                 """Returns the set of nodes in the component of graph containing node n.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(7)                    An undirected graph.
(4)                 n : node label
(7)                    A node in G
(4)                 Returns
(4)                 -------
(4)                 comp : set
(7)                    A set of nodes in the component of G containing node n.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If G is directed.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(0, 1), (1, 2), (5, 6), (3, 4)])
(4)                 >>> nx.node_connected_component(G, 0)  # nodes of component that contains
node 0
(4)                 {0, 1, 2}
(4)                 See Also
(4)                 --------
(4)                 connected_components
(4)                 Notes
(4)                 -----
(4)                 For undirected graphs only.
(4)                 """
(4)                 return _plain_bfs(G, n)
```

```
152: (0)            def _plain_bfs(G, source):
153: (4)                """A fast BFS node generator"""
154: (4)                adj = G._adj
155: (4)                n = len(adj)
156: (4)                seen = {source}
157: (4)                nextlevel = [source]
158: (4)                while nextlevel:
159: (8)                    thislevel = nextlevel
160: (8)                    nextlevel = []
161: (8)                    for v in thislevel:
162: (12)                       for w in adj[v]:
163: (16)                           if w not in seen:
164: (20)                               seen.add(w)
165: (20)                               nextlevel.append(w)
166: (12)                       if len(seen) == n:
167: (16)                           return seen
168: (4)                return seen
```

File 207 - semiconnected.py:

```
1: (0)              """Semiconnectedness."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for, pairwise
4: (0)              __all__ = ["is_semiconnected"]
5: (0)              @not_implemented_for("undirected")
6: (0)              @nx._dispatchable
7: (0)              def is_semiconnected(G):
8: (4)                  r"""Returns True if the graph is semiconnected, False otherwise.
9: (4)                  A graph is semiconnected if and only if for any pair of nodes, either one
10: (4)                 is reachable from the other, or they are mutually reachable.
11: (4)                 This function uses a theorem that states that a DAG is semiconnected
12: (4)                 if for any topological sort, for node $v_n$ in that sort, there is an
13: (4)                 edge $(v_i, v_{i+1})$. That allows us to check if a non-DAG `G` is
14: (4)                 semiconnected by condensing the graph: i.e. constructing a new graph `H`
15: (4)                 with nodes being the strongly connected components of `G`, and edges
16: (4)                 (scc_1, scc_2) if there is a edge $(v_1, v_2)$ in `G` for some
17: (4)                 $v_1 \in scc_1$ and $v_2 \in scc_2$. That results in a DAG, so we compute
18: (4)                 the topological sort of `H` and check if for every $n$ there is an edge
19: (4)                 $(scc_n, scc_{n+1})$.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : NetworkX graph
23: (8)                     A directed graph.
24: (4)                 Returns
25: (4)                 -------
26: (4)                 semiconnected : bool
27: (8)                     True if the graph is semiconnected, False otherwise.
28: (4)                 Raises
29: (4)                 ------
30: (4)                 NetworkXNotImplemented
31: (8)                     If the input graph is undirected.
32: (4)                 NetworkXPointlessConcept
33: (8)                     If the graph is empty.
34: (4)                 Examples
35: (4)                 --------
36: (4)                 >>> G = nx.path_graph(4, create_using=nx.DiGraph())
37: (4)                 >>> print(nx.is_semiconnected(G))
38: (4)                 True
39: (4)                 >>> G = nx.DiGraph([(1, 2), (3, 2)])
40: (4)                 >>> print(nx.is_semiconnected(G))
41: (4)                 False
42: (4)                 See Also
43: (4)                 --------
44: (4)                 is_strongly_connected
45: (4)                 is_weakly_connected
46: (4)                 is_connected
47: (4)                 is_biconnected
```

```
48: (4)                 """
49: (4)                 if len(G) == 0:
50: (8)                     raise nx.NetworkXPointlessConcept(
51: (12)                        "Connectivity is undefined for the null graph."
52: (8)                     )
53: (4)                 if not nx.is_weakly_connected(G):
54: (8)                     return False
55: (4)                 H = nx.condensation(G)
56: (4)                 return all(H.has_edge(u, v) for u, v in pairwise(nx.topological_sort(H)))
```

File 208 - strongly_connected.py:

```
1: (0)              """Strongly connected components."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils.decorators import not_implemented_for
4: (0)              __all__ = [
5: (4)                  "number_strongly_connected_components",
6: (4)                  "strongly_connected_components",
7: (4)                  "is_strongly_connected",
8: (4)                  "strongly_connected_components_recursive",
9: (4)                  "kosaraju_strongly_connected_components",
10: (4)                 "condensation",
11: (0)             ]
12: (0)             @not_implemented_for("undirected")
13: (0)             @nx._dispatchable
14: (0)             def strongly_connected_components(G):
15: (4)                 """Generate nodes in strongly connected components of graph.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G : NetworkX Graph
19: (8)                     A directed graph.
20: (4)                 Returns
21: (4)                 -------
22: (4)                 comp : generator of sets
23: (8)                     A generator of sets of nodes, one for each strongly connected
24: (8)                     component of G.
25: (4)                 Raises
26: (4)                 ------
27: (4)                 NetworkXNotImplemented
28: (8)                     If G is undirected.
29: (4)                 Examples
30: (4)                 --------
31: (4)                 Generate a sorted list of strongly connected components, largest first.
32: (4)                 >>> G = nx.cycle_graph(4, create_using=nx.DiGraph())
33: (4)                 >>> nx.add_cycle(G, [10, 11, 12])
34: (4)                 >>> [len(c) for c in sorted(nx.strongly_connected_components(G), key=len,
reverse=True)]
35: (4)                 [4, 3]
36: (4)                 If you only want the largest component, it's more efficient to
37: (4)                 use max instead of sort.
38: (4)                 >>> largest = max(nx.strongly_connected_components(G), key=len)
39: (4)                 See Also
40: (4)                 --------
41: (4)                 connected_components
42: (4)                 weakly_connected_components
43: (4)                 kosaraju_strongly_connected_components
44: (4)                 Notes
45: (4)                 -----
46: (4)                 Uses Tarjan's algorithm[1]_ with Nuutila's modifications[2]_.
47: (4)                 Nonrecursive version of algorithm.
48: (4)                 References
49: (4)                 ----------
50: (4)                 .. [1] Depth-first search and linear graph algorithms, R. Tarjan
51: (7)                    SIAM Journal of Computing 1(2):146-160, (1972).
52: (4)                 .. [2] On finding the strongly connected components in a directed graph.
53: (7)                    E. Nuutila and E. Soisalon-Soinen
54: (7)                    Information Processing Letters 49(1): 9-14, (1994)..
```

```
55: (4)                 """
56: (4)                 preorder = {}
57: (4)                 lowlink = {}
58: (4)                 scc_found = set()
59: (4)                 scc_queue = []
60: (4)                 i = 0  # Preorder counter
61: (4)                 neighbors = {v: iter(G[v]) for v in G}
62: (4)                 for source in G:
63: (8)                     if source not in scc_found:
64: (12)                        queue = [source]
65: (12)                        while queue:
66: (16)                            v = queue[-1]
67: (16)                            if v not in preorder:
68: (20)                                i = i + 1
69: (20)                                preorder[v] = i
70: (16)                            done = True
71: (16)                            for w in neighbors[v]:
72: (20)                                if w not in preorder:
73: (24)                                    queue.append(w)
74: (24)                                    done = False
75: (24)                                    break
76: (16)                            if done:
77: (20)                                lowlink[v] = preorder[v]
78: (20)                                for w in G[v]:
79: (24)                                    if w not in scc_found:
80: (28)                                        if preorder[w] > preorder[v]:
81: (32)                                            lowlink[v] = min([lowlink[v], lowlink[w]])
82: (28)                                        else:
83: (32)                                            lowlink[v] = min([lowlink[v], preorder[w]])
84: (20)                                queue.pop()
85: (20)                                if lowlink[v] == preorder[v]:
86: (24)                                    scc = {v}
87: (24)                                    while scc_queue and preorder[scc_queue[-1]] >
preorder[v]:
88: (28)                                        k = scc_queue.pop()
89: (28)                                        scc.add(k)
90: (24)                                    scc_found.update(scc)
91: (24)                                    yield scc
92: (20)                                else:
93: (24)                                    scc_queue.append(v)
94: (0)             @not_implemented_for("undirected")
95: (0)             @nx._dispatchable
96: (0)             def kosaraju_strongly_connected_components(G, source=None):
97: (4)                 """Generate nodes in strongly connected components of graph.
98: (4)                 Parameters
99: (4)                 ----------
100: (4)                G : NetworkX Graph
101: (8)                    A directed graph.
102: (4)                Returns
103: (4)                -------
104: (4)                comp : generator of sets
105: (8)                    A generator of sets of nodes, one for each strongly connected
106: (8)                    component of G.
107: (4)                Raises
108: (4)                ------
109: (4)                NetworkXNotImplemented
110: (8)                    If G is undirected.
111: (4)                Examples
112: (4)                --------
113: (4)                Generate a sorted list of strongly connected components, largest first.
114: (4)                >>> G = nx.cycle_graph(4, create_using=nx.DiGraph())
115: (4)                >>> nx.add_cycle(G, [10, 11, 12])
116: (4)                >>> [
117: (4)                ...     len(c)
118: (4)                ...     for c in sorted(
119: (4)                ...         nx.kosaraju_strongly_connected_components(G), key=len,
reverse=True
120: (4)                ...     )
121: (4)                ... ]
```

```
122: (4)                [4, 3]
123: (4)                If you only want the largest component, it's more efficient to
124: (4)                use max instead of sort.
125: (4)                >>> largest = max(nx.kosaraju_strongly_connected_components(G), key=len)
126: (4)                See Also
127: (4)                --------
128: (4)                strongly_connected_components
129: (4)                Notes
130: (4)                -----
131: (4)                Uses Kosaraju's algorithm.
132: (4)                """
133: (4)                post = list(nx.dfs_postorder_nodes(G.reverse(copy=False), source=source))
134: (4)                seen = set()
135: (4)                while post:
136: (8)                    r = post.pop()
137: (8)                    if r in seen:
138: (12)                       continue
139: (8)                    c = nx.dfs_preorder_nodes(G, r)
140: (8)                    new = {v for v in c if v not in seen}
141: (8)                    seen.update(new)
142: (8)                    yield new
143: (0)            @not_implemented_for("undirected")
144: (0)            @nx._dispatchable
145: (0)            def strongly_connected_components_recursive(G):
146: (4)                """Generate nodes in strongly connected components of graph.
147: (4)                .. deprecated:: 3.2
148: (7)                   This function is deprecated and will be removed in a future version of
149: (7)                   NetworkX. Use `strongly_connected_components` instead.
150: (4)                Recursive version of algorithm.
151: (4)                Parameters
152: (4)                ----------
153: (4)                G : NetworkX Graph
154: (8)                    A directed graph.
155: (4)                Returns
156: (4)                -------
157: (4)                comp : generator of sets
158: (8)                    A generator of sets of nodes, one for each strongly connected
159: (8)                    component of G.
160: (4)                Raises
161: (4)                ------
162: (4)                NetworkXNotImplemented
163: (8)                    If G is undirected.
164: (4)                Examples
165: (4)                --------
166: (4)                Generate a sorted list of strongly connected components, largest first.
167: (4)                >>> G = nx.cycle_graph(4, create_using=nx.DiGraph())
168: (4)                >>> nx.add_cycle(G, [10, 11, 12])
169: (4)                >>> [
170: (4)                ...     len(c)
171: (4)                ...     for c in sorted(
172: (4)                ...         nx.strongly_connected_components_recursive(G), key=len,
reverse=True
173: (4)                ...     )
174: (4)                ... ]
175: (4)                [4, 3]
176: (4)                If you only want the largest component, it's more efficient to
177: (4)                use max instead of sort.
178: (4)                >>> largest = max(nx.strongly_connected_components_recursive(G), key=len)
179: (4)                To create the induced subgraph of the components use:
180: (4)                >>> S = [G.subgraph(c).copy() for c in nx.weakly_connected_components(G)]
181: (4)                See Also
182: (4)                --------
183: (4)                connected_components
184: (4)                Notes
185: (4)                -----
186: (4)                Uses Tarjan's algorithm[1]_ with Nuutila's modifications[2]_.
187: (4)                References
188: (4)                ----------
189: (4)                .. [1] Depth-first search and linear graph algorithms, R. Tarjan
```

```
190: (7)                   SIAM Journal of Computing 1(2):146-160, (1972).
191: (4)                .. [2] On finding the strongly connected components in a directed graph.
192: (7)                   E. Nuutila and E. Soisalon-Soinen
193: (7)                   Information Processing Letters 49(1): 9-14, (1994)..
194: (4)                """
195: (4)                import warnings
196: (4)                warnings.warn(
197: (8)                    (
198: (12)                       "\n\nstrongly_connected_components_recursive is deprecated and
will be\n"
199: (12)                       "removed in the future. Use strongly_connected_components
instead."
200: (8)                    ),
201: (8)                    category=DeprecationWarning,
202: (8)                    stacklevel=2,
203: (4)                )
204: (4)                yield from strongly_connected_components(G)
205: (0)            @not_implemented_for("undirected")
206: (0)            @nx._dispatchable
207: (0)            def number_strongly_connected_components(G):
208: (4)                """Returns number of strongly connected components in graph.
209: (4)                Parameters
210: (4)                ----------
211: (4)                G : NetworkX graph
212: (7)                   A directed graph.
213: (4)                Returns
214: (4)                -------
215: (4)                n : integer
216: (7)                   Number of strongly connected components
217: (4)                Raises
218: (4)                ------
219: (4)                NetworkXNotImplemented
220: (8)                    If G is undirected.
221: (4)                Examples
222: (4)                --------
223: (4)                >>> G = nx.DiGraph(
224: (4)                ...     [(0, 1), (1, 2), (2, 0), (2, 3), (4, 5), (3, 4), (5, 6), (6, 3),
(6, 7)]
225: (4)                ... )
226: (4)                >>> nx.number_strongly_connected_components(G)
227: (4)                3
228: (4)                See Also
229: (4)                --------
230: (4)                strongly_connected_components
231: (4)                number_connected_components
232: (4)                number_weakly_connected_components
233: (4)                Notes
234: (4)                -----
235: (4)                For directed graphs only.
236: (4)                """
237: (4)                return sum(1 for scc in strongly_connected_components(G))
238: (0)            @not_implemented_for("undirected")
239: (0)            @nx._dispatchable
240: (0)            def is_strongly_connected(G):
241: (4)                """Test directed graph for strong connectivity.
242: (4)                A directed graph is strongly connected if and only if every vertex in
243: (4)                the graph is reachable from every other vertex.
244: (4)                Parameters
245: (4)                ----------
246: (4)                G : NetworkX Graph
247: (7)                   A directed graph.
248: (4)                Returns
249: (4)                -------
250: (4)                connected : bool
251: (6)                  True if the graph is strongly connected, False otherwise.
252: (4)                Examples
253: (4)                --------
254: (4)                >>> G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 0), (2, 4), (4, 2)])
255: (4)                >>> nx.is_strongly_connected(G)
```

```
(4)                 True
(4)                 >>> G.remove_edge(2, 3)
(4)                 >>> nx.is_strongly_connected(G)
(4)                 False
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If G is undirected.
(4)                 See Also
(4)                 --------
(4)                 is_weakly_connected
(4)                 is_semiconnected
(4)                 is_connected
(4)                 is_biconnected
(4)                 strongly_connected_components
(4)                 Notes
(4)                 -----
(4)                 For directed graphs only.
(4)                 """
(4)                 if len(G) == 0:
(8)                     raise nx.NetworkXPointlessConcept(
(12)                        """Connectivity is undefined for the null graph."""
(8)                     )
(4)                 return len(next(strongly_connected_components(G))) == len(G)
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable(returns_graph=True)
(0)             def condensation(G, scc=None):
(4)                 """Returns the condensation of G.
(4)                 The condensation of G is the graph with each of the strongly connected
(4)                 components contracted into a single node.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX DiGraph
(7)                    A directed graph.
(4)                 scc:  list or generator (optional, default=None)
(7)                    Strongly connected components. If provided, the elements in
(7)                    `scc` must partition the nodes in `G`. If not provided, it will be
(7)                    calculated as scc=nx.strongly_connected_components(G).
(4)                 Returns
(4)                 -------
(4)                 C : NetworkX DiGraph
(7)                    The condensation graph C of G.  The node labels are integers
(7)                    corresponding to the index of the component in the list of
(7)                    strongly connected components of G.  C has a graph attribute named
(7)                    'mapping' with a dictionary mapping the original nodes to the
(7)                    nodes in C to which they belong.  Each node in C also has a node
(7)                    attribute 'members' with the set of original nodes in G that
(7)                    form the SCC that the node in C represents.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If G is undirected.
(4)                 Examples
(4)                 --------
(4)                 Contracting two sets of strongly connected nodes into two distinct SCC
(4)                 using the barbell graph.
(4)                 >>> G = nx.barbell_graph(4, 0)
(4)                 >>> G.remove_edge(3, 4)
(4)                 >>> G = nx.DiGraph(G)
(4)                 >>> H = nx.condensation(G)
(4)                 >>> H.nodes.data()
(4)                 NodeDataView({0: {'members': {0, 1, 2, 3}}, 1: {'members': {4, 5, 6, 7}}})
(4)                 >>> H.graph["mapping"]
(4)                 {0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 1}
(4)                 Contracting a complete graph into one single SCC.
(4)                 >>> G = nx.complete_graph(7, create_using=nx.DiGraph)
(4)                 >>> H = nx.condensation(G)
(4)                 >>> H.nodes
(4)                 NodeView((0,))
```

```
325: (4)                >>> H.nodes.data()
326: (4)                NodeDataView({0: {'members': {0, 1, 2, 3, 4, 5, 6}}})
327: (4)                Notes
328: (4)                -----
329: (4)                After contracting all strongly connected components to a single node,
330: (4)                the resulting graph is a directed acyclic graph.
331: (4)                """
332: (4)                if scc is None:
333: (8)                    scc = nx.strongly_connected_components(G)
334: (4)                mapping = {}
335: (4)                members = {}
336: (4)                C = nx.DiGraph()
337: (4)                C.graph["mapping"] = mapping
338: (4)                if len(G) == 0:
339: (8)                    return C
340: (4)                for i, component in enumerate(scc):
341: (8)                    members[i] = component
342: (8)                    mapping.update((n, i) for n in component)
343: (4)                number_of_components = i + 1
344: (4)                C.add_nodes_from(range(number_of_components))
345: (4)                C.add_edges_from(
346: (8)                    (mapping[u], mapping[v]) for u, v in G.edges() if mapping[u] !=
mapping[v]
347: (4)                )
348: (4)                nx.set_node_attributes(C, members, "members")
349: (4)                return C
```

File 209 - __init__.py:

```
1: (0)
```

File 210 - weakly_connected.py:

```
1: (0)              """Weakly connected components."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils.decorators import not_implemented_for
4: (0)              __all__ = [
5: (4)                  "number_weakly_connected_components",
6: (4)                  "weakly_connected_components",
7: (4)                  "is_weakly_connected",
8: (0)              ]
9: (0)              @not_implemented_for("undirected")
10: (0)             @nx._dispatchable
11: (0)             def weakly_connected_components(G):
12: (4)                 """Generate weakly connected components of G.
13: (4)                 Parameters
14: (4)                 ----------
15: (4)                 G : NetworkX graph
16: (8)                     A directed graph
17: (4)                 Returns
18: (4)                 -------
19: (4)                 comp : generator of sets
20: (8)                     A generator of sets of nodes, one for each weakly connected
21: (8)                     component of G.
22: (4)                 Raises
23: (4)                 ------
24: (4)                 NetworkXNotImplemented
25: (8)                     If G is undirected.
26: (4)                 Examples
27: (4)                 --------
28: (4)                 Generate a sorted list of weakly connected components, largest first.
29: (4)                 >>> G = nx.path_graph(4, create_using=nx.DiGraph())
30: (4)                 >>> nx.add_path(G, [10, 11, 12])
31: (4)                 >>> [len(c) for c in sorted(nx.weakly_connected_components(G), key=len,
reverse=True)]
```

```
(4)                 [4, 3]
(4)                 If you only want the largest component, it's more efficient to
(4)                 use max instead of sort:
(4)                 >>> largest_cc = max(nx.weakly_connected_components(G), key=len)
(4)                 See Also
(4)                 --------
(4)                 connected_components
(4)                 strongly_connected_components
(4)                 Notes
(4)                 -----
(4)                 For directed graphs only.
(4)                 """
(4)                 seen = set()
(4)                 for v in G:
(8)                     if v not in seen:
(12)                        c = set(_plain_bfs(G, v))
(12)                        seen.update(c)
(12)                        yield c
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable
(0)             def number_weakly_connected_components(G):
(4)                 """Returns the number of weakly connected components in G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     A directed graph.
(4)                 Returns
(4)                 -------
(4)                 n : integer
(8)                     Number of weakly connected components
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If G is undirected.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph([(0, 1), (2, 1), (3, 4)])
(4)                 >>> nx.number_weakly_connected_components(G)
(4)                 2
(4)                 See Also
(4)                 --------
(4)                 weakly_connected_components
(4)                 number_connected_components
(4)                 number_strongly_connected_components
(4)                 Notes
(4)                 -----
(4)                 For directed graphs only.
(4)                 """
(4)                 return sum(1 for wcc in weakly_connected_components(G))
(0)             @not_implemented_for("undirected")
(0)             @nx._dispatchable
(0)             def is_weakly_connected(G):
(4)                 """Test directed graph for weak connectivity.
(4)                 A directed graph is weakly connected if and only if the graph
(4)                 is connected when the direction of the edge between nodes is ignored.
(4)                 Note that if a graph is strongly connected (i.e. the graph is connected
(4)                 even when we account for directionality), it is by definition weakly
(4)                 connected as well.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(8)                     A directed graph.
(4)                 Returns
(4)                 -------
(4)                 connected : bool
(8)                     True if the graph is weakly connected, False otherwise.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
```

```
101: (8)                    If G is undirected.
102: (4)                Examples
103: (4)                --------
104: (4)                >>> G = nx.DiGraph([(0, 1), (2, 1)])
105: (4)                >>> G.add_node(3)
106: (4)                >>> nx.is_weakly_connected(G)  # node 3 is not connected to the graph
107: (4)                False
108: (4)                >>> G.add_edge(2, 3)
109: (4)                >>> nx.is_weakly_connected(G)
110: (4)                True
111: (4)                See Also
112: (4)                --------
113: (4)                is_strongly_connected
114: (4)                is_semiconnected
115: (4)                is_connected
116: (4)                is_biconnected
117: (4)                weakly_connected_components
118: (4)                Notes
119: (4)                -----
120: (4)                For directed graphs only.
121: (4)                """
122: (4)                if len(G) == 0:
123: (8)                    raise nx.NetworkXPointlessConcept(
124: (12)                       """Connectivity is undefined for the null graph."""
125: (8)                    )
126: (4)                return len(next(weakly_connected_components(G))) == len(G)
127: (0)            def _plain_bfs(G, source):
128: (4)                """A fast BFS node generator
129: (4)                The direction of the edge between nodes is ignored.
130: (4)                For directed graphs only.
131: (4)                """
132: (4)                n = len(G)
133: (4)                Gsucc = G._succ
134: (4)                Gpred = G._pred
135: (4)                seen = {source}
136: (4)                nextlevel = [source]
137: (4)                yield source
138: (4)                while nextlevel:
139: (8)                    thislevel = nextlevel
140: (8)                    nextlevel = []
141: (8)                    for v in thislevel:
142: (12)                       for w in Gsucc[v]:
143: (16)                           if w not in seen:
144: (20)                               seen.add(w)
145: (20)                               nextlevel.append(w)
146: (20)                               yield w
147: (12)                       for w in Gpred[v]:
148: (16)                           if w not in seen:
149: (20)                               seen.add(w)
150: (20)                               nextlevel.append(w)
151: (20)                               yield w
152: (12)                       if len(seen) == n:
153: (16)                           return

----------------------------------------

File 211 - test_attracting.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import NetworkXNotImplemented
4: (0)              class TestAttractingComponents:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      cls.G1 = nx.DiGraph()
8: (8)                      cls.G1.add_edges_from(
9: (12)                         [
10: (16)                            (5, 11),
11: (16)                            (11, 2),
```

```
12: (16)                            (11, 9),
13: (16)                            (11, 10),
14: (16)                            (7, 11),
15: (16)                            (7, 8),
16: (16)                            (8, 9),
17: (16)                            (3, 8),
18: (16)                            (3, 10),
19: (12)                        ]
20: (8)                     )
21: (8)                     cls.G2 = nx.DiGraph()
22: (8)                     cls.G2.add_edges_from([(0, 1), (0, 2), (1, 1), (1, 2), (2, 1)])
23: (8)                     cls.G3 = nx.DiGraph()
24: (8)                     cls.G3.add_edges_from([(0, 1), (1, 2), (2, 1), (0, 3), (3, 4), (4,
3)])
25: (8)                     cls.G4 = nx.DiGraph()
26: (4)                 def test_attracting_components(self):
27: (8)                     ac = list(nx.attracting_components(self.G1))
28: (8)                     assert {2} in ac
29: (8)                     assert {9} in ac
30: (8)                     assert {10} in ac
31: (8)                     ac = list(nx.attracting_components(self.G2))
32: (8)                     ac = [tuple(sorted(x)) for x in ac]
33: (8)                     assert ac == [(1, 2)]
34: (8)                     ac = list(nx.attracting_components(self.G3))
35: (8)                     ac = [tuple(sorted(x)) for x in ac]
36: (8)                     assert (1, 2) in ac
37: (8)                     assert (3, 4) in ac
38: (8)                     assert len(ac) == 2
39: (8)                     ac = list(nx.attracting_components(self.G4))
40: (8)                     assert ac == []
41: (4)                 def test_number_attacting_components(self):
42: (8)                     assert nx.number_attracting_components(self.G1) == 3
43: (8)                     assert nx.number_attracting_components(self.G2) == 1
44: (8)                     assert nx.number_attracting_components(self.G3) == 2
45: (8)                     assert nx.number_attracting_components(self.G4) == 0
46: (4)                 def test_is_attracting_component(self):
47: (8)                     assert not nx.is_attracting_component(self.G1)
48: (8)                     assert not nx.is_attracting_component(self.G2)
49: (8)                     assert not nx.is_attracting_component(self.G3)
50: (8)                     g2 = self.G3.subgraph([1, 2])
51: (8)                     assert nx.is_attracting_component(g2)
52: (8)                     assert not nx.is_attracting_component(self.G4)
53: (4)                 def test_connected_raise(self):
54: (8)                     G = nx.Graph()
55: (8)                     with pytest.raises(NetworkXNotImplemented):
56: (12)                        next(nx.attracting_components(G))
57: (8)                     pytest.raises(NetworkXNotImplemented, nx.number_attracting_components,
G)
58: (8)                     pytest.raises(NetworkXNotImplemented, nx.is_attracting_component, G)

----------------------------------------

File 212 - test_biconnected.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import NetworkXNotImplemented
4: (0)              def assert_components_edges_equal(x, y):
5: (4)                  sx = {frozenset(frozenset(e) for e in c) for c in x}
6: (4)                  sy = {frozenset(frozenset(e) for e in c) for c in y}
7: (4)                  assert sx == sy
8: (0)              def assert_components_equal(x, y):
9: (4)                  sx = {frozenset(c) for c in x}
10: (4)                 sy = {frozenset(c) for c in y}
11: (4)                 assert sx == sy
12: (0)             def test_barbell():
13: (4)                 G = nx.barbell_graph(8, 4)
14: (4)                 nx.add_path(G, [7, 20, 21, 22])
15: (4)                 nx.add_cycle(G, [22, 23, 24, 25])
```

```
(4)                 pts = set(nx.articulation_points(G))
(4)                 assert pts == {7, 8, 9, 10, 11, 12, 20, 21, 22}
(4)                 answer = [
(8)                     {12, 13, 14, 15, 16, 17, 18, 19},
(8)                     {0, 1, 2, 3, 4, 5, 6, 7},
(8)                     {22, 23, 24, 25},
(8)                     {11, 12},
(8)                     {10, 11},
(8)                     {9, 10},
(8)                     {8, 9},
(8)                     {7, 8},
(8)                     {21, 22},
(8)                     {20, 21},
(8)                     {7, 20},
(4)                 ]
(4)                 assert_components_equal(list(nx.biconnected_components(G)), answer)
(4)                 G.add_edge(2, 17)
(4)                 pts = set(nx.articulation_points(G))
(4)                 assert pts == {7, 20, 21, 22}
(0)             def test_articulation_points_repetitions():
(4)                 G = nx.Graph()
(4)                 G.add_edges_from([(0, 1), (1, 2), (1, 3)])
(4)                 assert list(nx.articulation_points(G)) == [1]
(0)             def test_articulation_points_cycle():
(4)                 G = nx.cycle_graph(3)
(4)                 nx.add_cycle(G, [1, 3, 4])
(4)                 pts = set(nx.articulation_points(G))
(4)                 assert pts == {1}
(0)             def test_is_biconnected():
(4)                 G = nx.cycle_graph(3)
(4)                 assert nx.is_biconnected(G)
(4)                 nx.add_cycle(G, [1, 3, 4])
(4)                 assert not nx.is_biconnected(G)
(0)             def test_empty_is_biconnected():
(4)                 G = nx.empty_graph(5)
(4)                 assert not nx.is_biconnected(G)
(4)                 G.add_edge(0, 1)
(4)                 assert not nx.is_biconnected(G)
(0)             def test_biconnected_components_cycle():
(4)                 G = nx.cycle_graph(3)
(4)                 nx.add_cycle(G, [1, 3, 4])
(4)                 answer = [{0, 1, 2}, {1, 3, 4}]
(4)                 assert_components_equal(list(nx.biconnected_components(G)), answer)
(0)             def test_biconnected_components1():
(4)                 edges = [
(8)                     (0, 1),
(8)                     (0, 5),
(8)                     (0, 6),
(8)                     (0, 14),
(8)                     (1, 5),
(8)                     (1, 6),
(8)                     (1, 14),
(8)                     (2, 4),
(8)                     (2, 10),
(8)                     (3, 4),
(8)                     (3, 15),
(8)                     (4, 6),
(8)                     (4, 7),
(8)                     (4, 10),
(8)                     (5, 14),
(8)                     (6, 14),
(8)                     (7, 9),
(8)                     (8, 9),
(8)                     (8, 12),
(8)                     (8, 13),
(8)                     (10, 15),
(8)                     (11, 12),
(8)                     (11, 13),
(8)                     (12, 13),
```

```
85: (4)                 ]
86: (4)                 G = nx.Graph(edges)
87: (4)                 pts = set(nx.articulation_points(G))
88: (4)                 assert pts == {4, 6, 7, 8, 9}
89: (4)                 comps = list(nx.biconnected_component_edges(G))
90: (4)                 answer = [
91: (8)                     [(3, 4), (15, 3), (10, 15), (10, 4), (2, 10), (4, 2)],
92: (8)                     [(13, 12), (13, 8), (11, 13), (12, 11), (8, 12)],
93: (8)                     [(9, 8)],
94: (8)                     [(7, 9)],
95: (8)                     [(4, 7)],
96: (8)                     [(6, 4)],
97: (8)                     [(14, 0), (5, 1), (5, 0), (14, 5), (14, 1), (6, 14), (6, 0), (1, 6),
(0, 1)],
98: (4)                 ]
99: (4)                 assert_components_edges_equal(comps, answer)
100: (0)            def test_biconnected_components2():
101: (4)                G = nx.Graph()
102: (4)                nx.add_cycle(G, "ABC")
103: (4)                nx.add_cycle(G, "CDE")
104: (4)                nx.add_cycle(G, "FIJHG")
105: (4)                nx.add_cycle(G, "GIJ")
106: (4)                G.add_edge("E", "G")
107: (4)                comps = list(nx.biconnected_component_edges(G))
108: (4)                answer = [
109: (8)                    [
110: (12)                       tuple("GF"),
111: (12)                       tuple("FI"),
112: (12)                       tuple("IG"),
113: (12)                       tuple("IJ"),
114: (12)                       tuple("JG"),
115: (12)                       tuple("JH"),
116: (12)                       tuple("HG"),
117: (8)                    ],
118: (8)                    [tuple("EG")],
119: (8)                    [tuple("CD"), tuple("DE"), tuple("CE")],
120: (8)                    [tuple("AB"), tuple("BC"), tuple("AC")],
121: (4)                ]
122: (4)                assert_components_edges_equal(comps, answer)
123: (0)            def test_biconnected_davis():
124: (4)                D = nx.davis_southern_women_graph()
125: (4)                bcc = list(nx.biconnected_components(D))[0]
126: (4)                assert set(D) == bcc  # All nodes in a giant bicomponent
127: (4)                assert len(list(nx.articulation_points(D))) == 0
128: (0)            def test_biconnected_karate():
129: (4)                K = nx.karate_club_graph()
130: (4)                answer = [
131: (8)                    {
132: (12)                       0,
133: (12)                       1,
134: (12)                       2,
135: (12)                       3,
136: (12)                       7,
137: (12)                       8,
138: (12)                       9,
139: (12)                       12,
140: (12)                       13,
141: (12)                       14,
142: (12)                       15,
143: (12)                       17,
144: (12)                       18,
145: (12)                       19,
146: (12)                       20,
147: (12)                       21,
148: (12)                       22,
149: (12)                       23,
150: (12)                       24,
151: (12)                       25,
152: (12)                       26,
```

```
153: (12)                        27,
154: (12)                        28,
155: (12)                        29,
156: (12)                        30,
157: (12)                        31,
158: (12)                        32,
159: (12)                        33,
160: (8)                    },
161: (8)                    {0, 4, 5, 6, 10, 16},
162: (8)                    {0, 11},
163: (4)                ]
164: (4)                bcc = list(nx.biconnected_components(K))
165: (4)                assert_components_equal(bcc, answer)
166: (4)                assert set(nx.articulation_points(K)) == {0}
167: (0)            def test_biconnected_eppstein():
168: (4)                G1 = nx.Graph(
169: (8)                    {
170: (12)                       0: [1, 2, 5],
171: (12)                       1: [0, 5],
172: (12)                       2: [0, 3, 4],
173: (12)                       3: [2, 4, 5, 6],
174: (12)                       4: [2, 3, 5, 6],
175: (12)                       5: [0, 1, 3, 4],
176: (12)                       6: [3, 4],
177: (8)                    }
178: (4)                )
179: (4)                G2 = nx.Graph(
180: (8)                    {
181: (12)                       0: [2, 5],
182: (12)                       1: [3, 8],
183: (12)                       2: [0, 3, 5],
184: (12)                       3: [1, 2, 6, 8],
185: (12)                       4: [7],
186: (12)                       5: [0, 2],
187: (12)                       6: [3, 8],
188: (12)                       7: [4],
189: (12)                       8: [1, 3, 6],
190: (8)                    }
191: (4)                )
192: (4)                assert nx.is_biconnected(G1)
193: (4)                assert not nx.is_biconnected(G2)
194: (4)                answer_G2 = [{1, 3, 6, 8}, {0, 2, 5}, {2, 3}, {4, 7}]
195: (4)                bcc = list(nx.biconnected_components(G2))
196: (4)                assert_components_equal(bcc, answer_G2)
197: (0)            def test_null_graph():
198: (4)                G = nx.Graph()
199: (4)                assert not nx.is_biconnected(G)
200: (4)                assert list(nx.biconnected_components(G)) == []
201: (4)                assert list(nx.biconnected_component_edges(G)) == []
202: (4)                assert list(nx.articulation_points(G)) == []
203: (0)            def test_connected_raise():
204: (4)                DG = nx.DiGraph()
205: (4)                with pytest.raises(NetworkXNotImplemented):
206: (8)                    next(nx.biconnected_components(DG))
207: (4)                with pytest.raises(NetworkXNotImplemented):
208: (8)                    next(nx.biconnected_component_edges(DG))
209: (4)                with pytest.raises(NetworkXNotImplemented):
210: (8)                    next(nx.articulation_points(DG))
211: (4)                pytest.raises(NetworkXNotImplemented, nx.is_biconnected, DG)
```

File 213 - test_connected.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import NetworkXNotImplemented
4: (0)              from networkx import convert_node_labels_to_integers as cnlti
5: (0)              from networkx.classes.tests import dispatch_interface
```

```
6: (0)              class TestConnected:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      G1 = cnlti(nx.grid_2d_graph(2, 2), first_label=0, ordering="sorted")
10: (8)                     G2 = cnlti(nx.lollipop_graph(3, 3), first_label=4, ordering="sorted")
11: (8)                     G3 = cnlti(nx.house_graph(), first_label=10, ordering="sorted")
12: (8)                     cls.G = nx.union(G1, G2)
13: (8)                     cls.G = nx.union(cls.G, G3)
14: (8)                     cls.DG = nx.DiGraph([(1, 2), (1, 3), (2, 3)])
15: (8)                     cls.grid = cnlti(nx.grid_2d_graph(4, 4), first_label=1)
16: (8)                     cls.gc = []
17: (8)                     G = nx.DiGraph()
18: (8)                     G.add_edges_from(
19: (12)                        [
20: (16)                            (1, 2),
21: (16)                            (2, 3),
22: (16)                            (2, 8),
23: (16)                            (3, 4),
24: (16)                            (3, 7),
25: (16)                            (4, 5),
26: (16)                            (5, 3),
27: (16)                            (5, 6),
28: (16)                            (7, 4),
29: (16)                            (7, 6),
30: (16)                            (8, 1),
31: (16)                            (8, 7),
32: (12)                        ]
33: (8)                     )
34: (8)                     C = [[3, 4, 5, 7], [1, 2, 8], [6]]
35: (8)                     cls.gc.append((G, C))
36: (8)                     G = nx.DiGraph()
37: (8)                     G.add_edges_from([(1, 2), (1, 3), (1, 4), (4, 2), (3, 4), (2, 3)])
38: (8)                     C = [[2, 3, 4], [1]]
39: (8)                     cls.gc.append((G, C))
40: (8)                     G = nx.DiGraph()
41: (8)                     G.add_edges_from([(1, 2), (2, 3), (3, 2), (2, 1)])
42: (8)                     C = [[1, 2, 3]]
43: (8)                     cls.gc.append((G, C))
44: (8)                     G = nx.DiGraph({0: [1], 1: [2, 3], 2: [4, 5], 3: [4, 5], 4: [6], 5:
[], 6: []})
45: (8)                     C = [[0], [1], [2], [3], [4], [5], [6]]
46: (8)                     cls.gc.append((G, C))
47: (8)                     G = nx.DiGraph({0: [1], 1: [2, 3, 4], 2: [0, 3], 3: [4], 4: [3]})
48: (8)                     C = [[0, 1, 2], [3, 4]]
49: (8)                     cls.gc.append((G, C))
50: (8)                     G = nx.DiGraph()
51: (8)                     C = []
52: (8)                     cls.gc.append((G, C))
53: (4)                 @pytest.mark.parametrize("wrapper", [lambda x: x,
dispatch_interface.convert])
54: (4)                 def test_connected_components(self, wrapper):
55: (8)                     cc = nx.connected_components
56: (8)                     G = wrapper(self.G)
57: (8)                     C = {
58: (12)                        frozenset([0, 1, 2, 3]),
59: (12)                        frozenset([4, 5, 6, 7, 8, 9]),
60: (12)                        frozenset([10, 11, 12, 13, 14]),
61: (8)                     }
62: (8)                     assert {frozenset(g) for g in cc(G)} == C
63: (4)                 def test_number_connected_components(self):
64: (8)                     ncc = nx.number_connected_components
65: (8)                     assert ncc(self.G) == 3
66: (4)                 def test_number_connected_components2(self):
67: (8)                     ncc = nx.number_connected_components
68: (8)                     assert ncc(self.grid) == 1
69: (4)                 def test_connected_components2(self):
70: (8)                     cc = nx.connected_components
71: (8)                     G = self.grid
72: (8)                     C = {frozenset([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
```

```
16])}
73: (8)                     assert {frozenset(g) for g in cc(G)} == C
74: (4)                 def test_node_connected_components(self):
75: (8)                     ncc = nx.node_connected_component
76: (8)                     G = self.grid
77: (8)                     C = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
78: (8)                     assert ncc(G, 1) == C
79: (4)                 def test_is_connected(self):
80: (8)                     assert nx.is_connected(self.grid)
81: (8)                     G = nx.Graph()
82: (8)                     G.add_nodes_from([1, 2])
83: (8)                     assert not nx.is_connected(G)
84: (4)                 def test_connected_raise(self):
85: (8)                     with pytest.raises(NetworkXNotImplemented):
86: (12)                        next(nx.connected_components(self.DG))
87: (8)                     pytest.raises(NetworkXNotImplemented, nx.number_connected_components,
self.DG)
88: (8)                     pytest.raises(NetworkXNotImplemented, nx.node_connected_component,
self.DG, 1)
89: (8)                     pytest.raises(NetworkXNotImplemented, nx.is_connected, self.DG)
90: (8)                     pytest.raises(nx.NetworkXPointlessConcept, nx.is_connected,
nx.Graph())
91: (4)                 def test_connected_mutability(self):
92: (8)                     G = self.grid
93: (8)                     seen = set()
94: (8)                     for component in nx.connected_components(G):
95: (12)                        assert len(seen & component) == 0
96: (12)                        seen.update(component)
97: (12)                        component.clear()

----------------------------------------

File 214 - test_semiconnected.py:

1: (0)              from itertools import chain
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestIsSemiconnected:
5: (4)                  def test_undirected(self):
6: (8)                      pytest.raises(nx.NetworkXNotImplemented, nx.is_semiconnected,
nx.Graph())
7: (8)                      pytest.raises(nx.NetworkXNotImplemented, nx.is_semiconnected,
nx.MultiGraph())
8: (4)                  def test_empty(self):
9: (8)                      pytest.raises(nx.NetworkXPointlessConcept, nx.is_semiconnected,
nx.DiGraph())
10: (8)                     pytest.raises(
11: (12)                        nx.NetworkXPointlessConcept, nx.is_semiconnected,
nx.MultiDiGraph()
12: (8)                     )
13: (4)                 def test_single_node_graph(self):
14: (8)                     G = nx.DiGraph()
15: (8)                     G.add_node(0)
16: (8)                     assert nx.is_semiconnected(G)
17: (4)                 def test_path(self):
18: (8)                     G = nx.path_graph(100, create_using=nx.DiGraph())
19: (8)                     assert nx.is_semiconnected(G)
20: (8)                     G.add_edge(100, 99)
21: (8)                     assert not nx.is_semiconnected(G)
22: (4)                 def test_cycle(self):
23: (8)                     G = nx.cycle_graph(100, create_using=nx.DiGraph())
24: (8)                     assert nx.is_semiconnected(G)
25: (8)                     G = nx.path_graph(100, create_using=nx.DiGraph())
26: (8)                     G.add_edge(0, 99)
27: (8)                     assert nx.is_semiconnected(G)
28: (4)                 def test_tree(self):
29: (8)                     G = nx.DiGraph()
30: (8)                     G.add_edges_from(
31: (12)                        chain.from_iterable([(i, 2 * i + 1), (i, 2 * i + 2)] for i in
```

```
range(100))
32: (8)                     )
33: (8)                     assert not nx.is_semiconnected(G)
34: (4)                 def test_dumbbell(self):
35: (8)                     G = nx.cycle_graph(100, create_using=nx.DiGraph())
36: (8)                     G.add_edges_from((i + 100, (i + 1) % 100 + 100) for i in range(100))
37: (8)                     assert not nx.is_semiconnected(G)  # G is disconnected.
38: (8)                     G.add_edge(100, 99)
39: (8)                     assert nx.is_semiconnected(G)
40: (4)                 def test_alternating_path(self):
41: (8)                     G = nx.DiGraph(
42: (12)                        chain.from_iterable([(i, i - 1), (i, i + 1)] for i in range(0,
100, 2))
43: (8)                     )
44: (8)                     assert not nx.is_semiconnected(G)

----------------------------------------

File 215 - test_strongly_connected.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import NetworkXNotImplemented
4: (0)              class TestStronglyConnected:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      cls.gc = []
8: (8)                      G = nx.DiGraph()
9: (8)                      G.add_edges_from(
10: (12)                        [
11: (16)                            (1, 2),
12: (16)                            (2, 3),
13: (16)                            (2, 8),
14: (16)                            (3, 4),
15: (16)                            (3, 7),
16: (16)                            (4, 5),
17: (16)                            (5, 3),
18: (16)                            (5, 6),
19: (16)                            (7, 4),
20: (16)                            (7, 6),
21: (16)                            (8, 1),
22: (16)                            (8, 7),
23: (12)                        ]
24: (8)                     )
25: (8)                     C = {frozenset([3, 4, 5, 7]), frozenset([1, 2, 8]), frozenset([6])}
26: (8)                     cls.gc.append((G, C))
27: (8)                     G = nx.DiGraph()
28: (8)                     G.add_edges_from([(1, 2), (1, 3), (1, 4), (4, 2), (3, 4), (2, 3)])
29: (8)                     C = {frozenset([2, 3, 4]), frozenset([1])}
30: (8)                     cls.gc.append((G, C))
31: (8)                     G = nx.DiGraph()
32: (8)                     G.add_edges_from([(1, 2), (2, 3), (3, 2), (2, 1)])
33: (8)                     C = {frozenset([1, 2, 3])}
34: (8)                     cls.gc.append((G, C))
35: (8)                     G = nx.DiGraph({0: [1], 1: [2, 3], 2: [4, 5], 3: [4, 5], 4: [6], 5:
[], 6: []})
36: (8)                     C = {
37: (12)                        frozenset([0]),
38: (12)                        frozenset([1]),
39: (12)                        frozenset([2]),
40: (12)                        frozenset([3]),
41: (12)                        frozenset([4]),
42: (12)                        frozenset([5]),
43: (12)                        frozenset([6]),
44: (8)                     }
45: (8)                     cls.gc.append((G, C))
46: (8)                     G = nx.DiGraph({0: [1], 1: [2, 3, 4], 2: [0, 3], 3: [4], 4: [3]})
47: (8)                     C = {frozenset([0, 1, 2]), frozenset([3, 4])}
48: (8)                     cls.gc.append((G, C))
```

```
(4)     def test_tarjan(self):
(8)         scc = nx.strongly_connected_components
(8)         for G, C in self.gc:
(12)            assert {frozenset(g) for g in scc(G)} == C
(4)     def test_tarjan_recursive(self):
(8)         scc = nx.strongly_connected_components_recursive
(8)         for G, C in self.gc:
(12)            with pytest.deprecated_call():
(16)                assert {frozenset(g) for g in scc(G)} == C
(4)     def test_kosaraju(self):
(8)         scc = nx.kosaraju_strongly_connected_components
(8)         for G, C in self.gc:
(12)            assert {frozenset(g) for g in scc(G)} == C
(4)     def test_number_strongly_connected_components(self):
(8)         ncc = nx.number_strongly_connected_components
(8)         for G, C in self.gc:
(12)            assert ncc(G) == len(C)
(4)     def test_is_strongly_connected(self):
(8)         for G, C in self.gc:
(12)            if len(C) == 1:
(16)                assert nx.is_strongly_connected(G)
(12)            else:
(16)                assert not nx.is_strongly_connected(G)
(4)     def test_contract_scc1(self):
(8)         G = nx.DiGraph()
(8)         G.add_edges_from(
(12)            [
(16)                (1, 2),
(16)                (2, 3),
(16)                (2, 11),
(16)                (2, 12),
(16)                (3, 4),
(16)                (4, 3),
(16)                (4, 5),
(16)                (5, 6),
(16)                (6, 5),
(16)                (6, 7),
(16)                (7, 8),
(16)                (7, 9),
(16)                (7, 10),
(16)                (8, 9),
(16)                (9, 7),
(16)                (10, 6),
(16)                (11, 2),
(16)                (11, 4),
(16)                (11, 6),
(16)                (12, 6),
(16)                (12, 11),
(12)            ]
(8)         )
(8)         scc = list(nx.strongly_connected_components(G))
(8)         cG = nx.condensation(G, scc)
(8)         assert nx.is_directed_acyclic_graph(cG)
(8)         assert sorted(cG.nodes()) == [0, 1, 2, 3]
(8)         mapping = {}
(8)         for i, component in enumerate(scc):
(12)            for n in component:
(16)                mapping[n] = i
(8)         edge = (mapping[2], mapping[3])
(8)         assert cG.has_edge(*edge)
(8)         edge = (mapping[2], mapping[5])
(8)         assert cG.has_edge(*edge)
(8)         edge = (mapping[3], mapping[5])
(8)         assert cG.has_edge(*edge)
(4)     def test_contract_scc_isolate(self):
(8)         G = nx.DiGraph()
(8)         G.add_edge(1, 2)
(8)         G.add_edge(2, 1)
(8)         scc = list(nx.strongly_connected_components(G))
```

```
118: (8)                    cG = nx.condensation(G, scc)
119: (8)                    assert list(cG.nodes()) == [0]
120: (8)                    assert list(cG.edges()) == []
121: (4)                def test_contract_scc_edge(self):
122: (8)                    G = nx.DiGraph()
123: (8)                    G.add_edge(1, 2)
124: (8)                    G.add_edge(2, 1)
125: (8)                    G.add_edge(2, 3)
126: (8)                    G.add_edge(3, 4)
127: (8)                    G.add_edge(4, 3)
128: (8)                    scc = list(nx.strongly_connected_components(G))
129: (8)                    cG = nx.condensation(G, scc)
130: (8)                    assert sorted(cG.nodes()) == [0, 1]
131: (8)                    if 1 in scc[0]:
132: (12)                       edge = (0, 1)
133: (8)                    else:
134: (12)                       edge = (1, 0)
135: (8)                    assert list(cG.edges()) == [edge]
136: (4)                def test_condensation_mapping_and_members(self):
137: (8)                    G, C = self.gc[1]
138: (8)                    C = sorted(C, key=len, reverse=True)
139: (8)                    cG = nx.condensation(G)
140: (8)                    mapping = cG.graph["mapping"]
141: (8)                    assert all(n in G for n in mapping)
142: (8)                    assert all(0 == cN for n, cN in mapping.items() if n in C[0])
143: (8)                    assert all(1 == cN for n, cN in mapping.items() if n in C[1])
144: (8)                    for n, d in cG.nodes(data=True):
145: (12)                       assert set(C[n]) == cG.nodes[n]["members"]
146: (4)                def test_null_graph(self):
147: (8)                    G = nx.DiGraph()
148: (8)                    assert list(nx.strongly_connected_components(G)) == []
149: (8)                    assert list(nx.kosaraju_strongly_connected_components(G)) == []
150: (8)                    with pytest.deprecated_call():
151: (12)                       assert list(nx.strongly_connected_components_recursive(G)) == []
152: (8)                    assert len(nx.condensation(G)) == 0
153: (8)                    pytest.raises(
154: (12)                       nx.NetworkXPointlessConcept, nx.is_strongly_connected,
nx.DiGraph()
155: (8)                    )
156: (4)                def test_connected_raise(self):
157: (8)                    G = nx.Graph()
158: (8)                    with pytest.raises(NetworkXNotImplemented):
159: (12)                       next(nx.strongly_connected_components(G))
160: (8)                    with pytest.raises(NetworkXNotImplemented):
161: (12)                       next(nx.kosaraju_strongly_connected_components(G))
162: (8)                    with pytest.raises(NetworkXNotImplemented):
163: (12)                       next(nx.strongly_connected_components_recursive(G))
164: (8)                    pytest.raises(NetworkXNotImplemented, nx.is_strongly_connected, G)
165: (8)                    pytest.raises(NetworkXNotImplemented, nx.condensation, G)
166: (4)                strong_cc_methods = (
167: (8)                    nx.strongly_connected_components,
168: (8)                    nx.kosaraju_strongly_connected_components,
169: (4)                )
170: (4)                @pytest.mark.parametrize("get_components", strong_cc_methods)
171: (4)                def test_connected_mutability(self, get_components):
172: (8)                    DG = nx.path_graph(5, create_using=nx.DiGraph)
173: (8)                    G = nx.disjoint_union(DG, DG)
174: (8)                    seen = set()
175: (8)                    for component in get_components(G):
176: (12)                       assert len(seen & component) == 0
177: (12)                       seen.update(component)
178: (12)                       component.clear()
```

File 216 - test_weakly_connected.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
```

```
3: (0)              from networkx import NetworkXNotImplemented
4: (0)              class TestWeaklyConnected:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      cls.gc = []
8: (8)                      G = nx.DiGraph()
9: (8)                      G.add_edges_from(
10: (12)                        [
11: (16)                            (1, 2),
12: (16)                            (2, 3),
13: (16)                            (2, 8),
14: (16)                            (3, 4),
15: (16)                            (3, 7),
16: (16)                            (4, 5),
17: (16)                            (5, 3),
18: (16)                            (5, 6),
19: (16)                            (7, 4),
20: (16)                            (7, 6),
21: (16)                            (8, 1),
22: (16)                            (8, 7),
23: (12)                        ]
24: (8)                     )
25: (8)                     C = [[3, 4, 5, 7], [1, 2, 8], [6]]
26: (8)                     cls.gc.append((G, C))
27: (8)                     G = nx.DiGraph()
28: (8)                     G.add_edges_from([(1, 2), (1, 3), (1, 4), (4, 2), (3, 4), (2, 3)])
29: (8)                     C = [[2, 3, 4], [1]]
30: (8)                     cls.gc.append((G, C))
31: (8)                     G = nx.DiGraph()
32: (8)                     G.add_edges_from([(1, 2), (2, 3), (3, 2), (2, 1)])
33: (8)                     C = [[1, 2, 3]]
34: (8)                     cls.gc.append((G, C))
35: (8)                     G = nx.DiGraph({0: [1], 1: [2, 3], 2: [4, 5], 3: [4, 5], 4: [6], 5:
[], 6: []})
36: (8)                     C = [[0], [1], [2], [3], [4], [5], [6]]
37: (8)                     cls.gc.append((G, C))
38: (8)                     G = nx.DiGraph({0: [1], 1: [2, 3, 4], 2: [0, 3], 3: [4], 4: [3]})
39: (8)                     C = [[0, 1, 2], [3, 4]]
40: (8)                     cls.gc.append((G, C))
41: (4)                 def test_weakly_connected_components(self):
42: (8)                     for G, C in self.gc:
43: (12)                        U = G.to_undirected()
44: (12)                        w = {frozenset(g) for g in nx.weakly_connected_components(G)}
45: (12)                        c = {frozenset(g) for g in nx.connected_components(U)}
46: (12)                        assert w == c
47: (4)                 def test_number_weakly_connected_components(self):
48: (8)                     for G, C in self.gc:
49: (12)                        U = G.to_undirected()
50: (12)                        w = nx.number_weakly_connected_components(G)
51: (12)                        c = nx.number_connected_components(U)
52: (12)                        assert w == c
53: (4)                 def test_is_weakly_connected(self):
54: (8)                     for G, C in self.gc:
55: (12)                        U = G.to_undirected()
56: (12)                        assert nx.is_weakly_connected(G) == nx.is_connected(U)
57: (4)                 def test_null_graph(self):
58: (8)                     G = nx.DiGraph()
59: (8)                     assert list(nx.weakly_connected_components(G)) == []
60: (8)                     assert nx.number_weakly_connected_components(G) == 0
61: (8)                     with pytest.raises(nx.NetworkXPointlessConcept):
62: (12)                        next(nx.is_weakly_connected(G))
63: (4)                 def test_connected_raise(self):
64: (8)                     G = nx.Graph()
65: (8)                     with pytest.raises(NetworkXNotImplemented):
66: (12)                        next(nx.weakly_connected_components(G))
67: (8)                     pytest.raises(NetworkXNotImplemented,
nx.number_weakly_connected_components, G)
68: (8)                     pytest.raises(NetworkXNotImplemented, nx.is_weakly_connected, G)
69: (4)                 def test_connected_mutability(self):
```

```
70: (8)                     DG = nx.path_graph(5, create_using=nx.DiGraph)
71: (8)                     G = nx.disjoint_union(DG, DG)
72: (8)                     seen = set()
73: (8)                     for component in nx.weakly_connected_components(G):
74: (12)                        assert len(seen & component) == 0
75: (12)                        seen.update(component)
76: (12)                        component.clear()
77: (0)             def test_is_weakly_connected_empty_graph_raises():
78: (4)                 G = nx.DiGraph()
79: (4)                 with pytest.raises(nx.NetworkXPointlessConcept, match="Connectivity is
undefined"):
80: (8)                     nx.is_weakly_connected(G)
```

File 217 - __init__.py:

```
1: (0)              """Connectivity and cut algorithms
2: (0)              """
3: (0)              from .connectivity import *
4: (0)              from .cuts import *
5: (0)              from .edge_augmentation import *
6: (0)              from .edge_kcomponents import *
7: (0)              from .disjoint_paths import *
8: (0)              from .kcomponents import *
9: (0)              from .kcutsets import *
10: (0)             from .stoerwagner import *
11: (0)             from .utils import *
```

File 218 - connectivity.py:

```
1: (0)              """
2: (0)              Flow based connectivity algorithms
3: (0)              """
4: (0)              import itertools
5: (0)              from operator import itemgetter
6: (0)              import networkx as nx
7: (0)              from networkx.algorithms.flow import (
8: (4)                  boykov_kolmogorov,
9: (4)                  build_residual_network,
10: (4)                 dinitz,
11: (4)                 edmonds_karp,
12: (4)                 shortest_augmenting_path,
13: (0)             )
14: (0)             default_flow_func = edmonds_karp
15: (0)             from .utils import build_auxiliary_edge_connectivity,
build_auxiliary_node_connectivity
16: (0)             __all__ = [
17: (4)                 "average_node_connectivity",
18: (4)                 "local_node_connectivity",
19: (4)                 "node_connectivity",
20: (4)                 "local_edge_connectivity",
21: (4)                 "edge_connectivity",
22: (4)                 "all_pairs_node_connectivity",
23: (0)             ]
24: (0)             @nx._dispatchable(graphs={"G": 0, "auxiliary?": 4}, preserve_graph_attrs=
{"auxiliary"})
25: (0)             def local_node_connectivity(
26: (4)                 G, s, t, flow_func=None, auxiliary=None, residual=None, cutoff=None
27: (0)             ):
28: (4)                 r"""Computes local node connectivity for nodes s and t.
29: (4)                 Local node connectivity for two non adjacent nodes s and t is the
30: (4)                 minimum number of nodes that must be removed (along with their incident
31: (4)                 edges) to disconnect them.
32: (4)                 This is a flow based implementation of node connectivity. We compute the
33: (4)                 maximum flow on an auxiliary digraph build from the original input
34: (4)                 graph (see below for details).
```

```
 35: (4)                 Parameters
 36: (4)                 ----------
 37: (4)                 G : NetworkX graph
 38: (8)                     Undirected graph
 39: (4)                 s : node
 40: (8)                     Source node
 41: (4)                 t : node
 42: (8)                     Target node
 43: (4)                 flow_func : function
 44: (8)                     A function for computing the maximum flow among a pair of nodes.
 45: (8)                     The function has to accept at least three parameters: a Digraph,
 46: (8)                     a source node, and a target node. And return a residual network
 47: (8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
 48: (8)                     details). If flow_func is None, the default maximum flow function
 49: (8)                     (:meth:`edmonds_karp`) is used. See below for details. The choice
 50: (8)                     of the default function may change from version to version and
 51: (8)                     should not be relied on. Default value: None.
 52: (4)                 auxiliary : NetworkX DiGraph
 53: (8)                     Auxiliary digraph to compute flow based node connectivity. It has
 54: (8)                     to have a graph attribute called mapping with a dictionary mapping
 55: (8)                     node names in G and in the auxiliary digraph. If provided
 56: (8)                     it will be reused instead of recreated. Default value: None.
 57: (4)                 residual : NetworkX DiGraph
 58: (8)                     Residual network to compute maximum flow. If provided it will be
 59: (8)                     reused instead of recreated. Default value: None.
 60: (4)                 cutoff : integer, float, or None (default: None)
 61: (8)                     If specified, the maximum flow algorithm will terminate when the
 62: (8)                     flow value reaches or exceeds the cutoff. This only works for flows
 63: (8)                     that support the cutoff parameter (most do) and is ignored otherwise.
 64: (4)                 Returns
 65: (4)                 -------
 66: (4)                 K : integer
 67: (8)                     local node connectivity for nodes s and t
 68: (4)                 Examples
 69: (4)                 --------
 70: (4)                 This function is not imported in the base NetworkX namespace, so you
 71: (4)                 have to explicitly import it from the connectivity package:
 72: (4)                 >>> from networkx.algorithms.connectivity import local_node_connectivity
 73: (4)                 We use in this example the platonic icosahedral graph, which has node
 74: (4)                 connectivity 5.
 75: (4)                 >>> G = nx.icosahedral_graph()
 76: (4)                 >>> local_node_connectivity(G, 0, 6)
 77: (4)                 5
 78: (4)                 If you need to compute local connectivity on several pairs of
 79: (4)                 nodes in the same graph, it is recommended that you reuse the
 80: (4)                 data structures that NetworkX uses in the computation: the
 81: (4)                 auxiliary digraph for node connectivity, and the residual
 82: (4)                 network for the underlying maximum flow computation.
 83: (4)                 Example of how to compute local node connectivity among
 84: (4)                 all pairs of nodes of the platonic icosahedral graph reusing
 85: (4)                 the data structures.
 86: (4)                 >>> import itertools
 87: (4)                 >>> # You also have to explicitly import the function for
 88: (4)                 >>> # building the auxiliary digraph from the connectivity package
 89: (4)                 >>> from networkx.algorithms.connectivity import
build_auxiliary_node_connectivity
 90: (4)                 >>> H = build_auxiliary_node_connectivity(G)
 91: (4)                 >>> # And the function for building the residual network from the
 92: (4)                 >>> # flow package
 93: (4)                 >>> from networkx.algorithms.flow import build_residual_network
 94: (4)                 >>> # Note that the auxiliary digraph has an edge attribute named capacity
 95: (4)                 >>> R = build_residual_network(H, "capacity")
 96: (4)                 >>> result = dict.fromkeys(G, dict())
 97: (4)                 >>> # Reuse the auxiliary digraph and the residual network by passing them
 98: (4)                 >>> # as parameters
 99: (4)                 >>> for u, v in itertools.combinations(G, 2):
100: (4)                 ...     k = local_node_connectivity(G, u, v, auxiliary=H, residual=R)
101: (4)                 ...     result[u][v] = k
102: (4)                 >>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
```

```
(4)                 True
(4)                 You can also use alternative flow algorithms for computing node
(4)                 connectivity. For instance, in dense networks the algorithm
(4)                 :meth:`shortest_augmenting_path` will usually perform better than
(4)                 the default :meth:`edmonds_karp` which is faster for sparse
(4)                 networks with highly skewed degree distributions. Alternative flow
(4)                 functions have to be explicitly imported from the flow package.
(4)                 >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)                 >>> local_node_connectivity(G, 0, 6, flow_func=shortest_augmenting_path)
(4)                 5
(4)                 Notes
(4)                 -----
(4)                 This is a flow based implementation of node connectivity. We compute the
(4)                 maximum flow using, by default, the :meth:`edmonds_karp` algorithm (see:
(4)                 :meth:`maximum_flow`) on an auxiliary digraph build from the original
(4)                 input graph:
(4)                 For an undirected graph G having `n` nodes and `m` edges we derive a
(4)                 directed graph H with `2n` nodes and `2m+n` arcs by replacing each
(4)                 original node `v` with two nodes `v_A`, `v_B` linked by an (internal)
(4)                 arc in H. Then for each edge (`u`, `v`) in G we add two arcs
(4)                 (`u_B`, `v_A`) and (`v_B`, `u_A`) in H. Finally we set the attribute
(4)                 capacity = 1 for each arc in H [1]_ .
(4)                 For a directed graph G having `n` nodes and `m` arcs we derive a
(4)                 directed graph H with `2n` nodes and `m+n` arcs by replacing each
(4)                 original node `v` with two nodes `v_A`, `v_B` linked by an (internal)
(4)                 arc (`v_A`, `v_B`) in H. Then for each arc (`u`, `v`) in G we add one arc
(4)                 (`u_B`, `v_A`) in H. Finally we set the attribute capacity = 1 for
(4)                 each arc in H.
(4)                 This is equal to the local node connectivity because the value of
(4)                 a maximum s-t-flow is equal to the capacity of a minimum s-t-cut.
(4)                 See also
(4)                 --------
(4)                 :meth:`local_edge_connectivity`
(4)                 :meth:`node_connectivity`
(4)                 :meth:`minimum_node_cut`
(4)                 :meth:`maximum_flow`
(4)                 :meth:`edmonds_karp`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 References
(4)                 ----------
(4)                 .. [1] Kammer, Frank and Hanjo Taubig. Graph Connectivity. in Brandes and
(8)                     Erlebach, 'Network Analysis: Methodological Foundations', Lecture
(8)                     Notes in Computer Science, Volume 3418, Springer-Verlag, 2005.
(8)                     http://www.informatik.uni-
augsburg.de/thi/personen/kammer/Graph_Connectivity.pdf
(4)                 """
(4)                 if flow_func is None:
(8)                     flow_func = default_flow_func
(4)                 if auxiliary is None:
(8)                     H = build_auxiliary_node_connectivity(G)
(4)                 else:
(8)                     H = auxiliary
(4)                 mapping = H.graph.get("mapping", None)
(4)                 if mapping is None:
(8)                     raise nx.NetworkXError("Invalid auxiliary digraph.")
(4)                 kwargs = {"flow_func": flow_func, "residual": residual}
(4)                 if flow_func is shortest_augmenting_path:
(8)                     kwargs["cutoff"] = cutoff
(8)                     kwargs["two_phase"] = True
(4)                 elif flow_func is edmonds_karp:
(8)                     kwargs["cutoff"] = cutoff
(4)                 elif flow_func is dinitz:
(8)                     kwargs["cutoff"] = cutoff
(4)                 elif flow_func is boykov_kolmogorov:
(8)                     kwargs["cutoff"] = cutoff
(4)                 return nx.maximum_flow_value(H, f"{mapping[s]}B", f"{mapping[t]}A",
**kwargs)
(0)             @nx._dispatchable
```

```
(0)            def node_connectivity(G, s=None, t=None, flow_func=None):
(4)                r"""Returns node connectivity for a graph or digraph G.
(4)                Node connectivity is equal to the minimum number of nodes that
(4)                must be removed to disconnect G or render it trivial. If source
(4)                and target nodes are provided, this function returns the local node
(4)                connectivity: the minimum number of nodes that must be removed to break
(4)                all paths from source to target in G.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX graph
(8)                    Undirected graph
(4)                s : node
(8)                    Source node. Optional. Default value: None.
(4)                t : node
(8)                    Target node. Optional. Default value: None.
(4)                flow_func : function
(8)                    A function for computing the maximum flow among a pair of nodes.
(8)                    The function has to accept at least three parameters: a Digraph,
(8)                    a source node, and a target node. And return a residual network
(8)                    that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)                    details). If flow_func is None, the default maximum flow function
(8)                    (:meth:`edmonds_karp`) is used. See below for details. The
(8)                    choice of the default function may change from version
(8)                    to version and should not be relied on. Default value: None.
(4)                Returns
(4)                -------
(4)                K : integer
(8)                    Node connectivity of G, or local node connectivity if source
(8)                    and target are provided.
(4)                Examples
(4)                --------
(4)                >>> # Platonic icosahedral graph is 5-node-connected
(4)                >>> G = nx.icosahedral_graph()
(4)                >>> nx.node_connectivity(G)
(4)                5
(4)                You can use alternative flow algorithms for the underlying maximum
(4)                flow computation. In dense networks the algorithm
(4)                :meth:`shortest_augmenting_path` will usually perform better
(4)                than the default :meth:`edmonds_karp`, which is faster for
(4)                sparse networks with highly skewed degree distributions. Alternative
(4)                flow functions have to be explicitly imported from the flow package.
(4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)                >>> nx.node_connectivity(G, flow_func=shortest_augmenting_path)
(4)                5
(4)                If you specify a pair of nodes (source and target) as parameters,
(4)                this function returns the value of local node connectivity.
(4)                >>> nx.node_connectivity(G, 3, 7)
(4)                5
(4)                If you need to perform several local computations among different
(4)                pairs of nodes on the same graph, it is recommended that you reuse
(4)                the data structures used in the maximum flow computations. See
(4)                :meth:`local_node_connectivity` for details.
(4)                Notes
(4)                -----
(4)                This is a flow based implementation of node connectivity. The
(4)                algorithm works by solving $O((n-\delta-1+\delta(\delta-1)/2))$
(4)                maximum flow problems on an auxiliary digraph. Where $\delta$
(4)                is the minimum degree of G. For details about the auxiliary
(4)                digraph and the computation of local node connectivity see
(4)                :meth:`local_node_connectivity`. This implementation is based
(4)                on algorithm 11 in [1]_.
(4)                See also
(4)                --------
(4)                :meth:`local_node_connectivity`
(4)                :meth:`edge_connectivity`
(4)                :meth:`maximum_flow`
(4)                :meth:`edmonds_karp`
(4)                :meth:`preflow_push`
(4)                :meth:`shortest_augmenting_path`
```

```
(4)                 References
(4)                 ----------
(4)                 .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms.
(8)                     http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
(4)                 """
(4)                 if (s is not None and t is None) or (s is None and t is not None):
(8)                     raise nx.NetworkXError("Both source and target must be specified.")
(4)                 if s is not None and t is not None:
(8)                     if s not in G:
(12)                        raise nx.NetworkXError(f"node {s} not in graph")
(8)                     if t not in G:
(12)                        raise nx.NetworkXError(f"node {t} not in graph")
(8)                     return local_node_connectivity(G, s, t, flow_func=flow_func)
(4)                 if G.is_directed():
(8)                     if not nx.is_weakly_connected(G):
(12)                        return 0
(8)                     iter_func = itertools.permutations
(8)                     def neighbors(v):
(12)                        return itertools.chain.from_iterable([G.predecessors(v),
G.successors(v)])
(4)                 else:
(8)                     if not nx.is_connected(G):
(12)                        return 0
(8)                     iter_func = itertools.combinations
(8)                     neighbors = G.neighbors
(4)                 H = build_auxiliary_node_connectivity(G)
(4)                 R = build_residual_network(H, "capacity")
(4)                 kwargs = {"flow_func": flow_func, "auxiliary": H, "residual": R}
(4)                 v, K = min(G.degree(), key=itemgetter(1))
(4)                 for w in set(G) - set(neighbors(v)) - {v}:
(8)                     kwargs["cutoff"] = K
(8)                     K = min(K, local_node_connectivity(G, v, w, **kwargs))
(4)                 for x, y in iter_func(neighbors(v), 2):
(8)                     if y in G[x]:
(12)                        continue
(8)                     kwargs["cutoff"] = K
(8)                     K = min(K, local_node_connectivity(G, x, y, **kwargs))
(4)                 return K
(0)             @nx._dispatchable
(0)             def average_node_connectivity(G, flow_func=None):
(4)                 r"""Returns the average connectivity of a graph G.
(4)                 The average connectivity `\bar{\kappa}` of a graph G is the average
(4)                 of local node connectivity over all pairs of nodes of G [1]_ .
(4)                 .. math::
(8)                     \bar{\kappa}(G) = \frac{\sum_{u,v} \kappa_{G}(u,v)}{{n \choose 2}}
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Undirected graph
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes.
(8)                     The function has to accept at least three parameters: a Digraph,
(8)                     a source node, and a target node. And return a residual network
(8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)                     details). If flow_func is None, the default maximum flow function
(8)                     (:meth:`edmonds_karp`) is used. See :meth:`local_node_connectivity`
(8)                     for details. The choice of the default function may change from
(8)                     version to version and should not be relied on. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 K : float
(8)                     Average node connectivity
(4)                 See also
(4)                 --------
(4)                 :meth:`local_node_connectivity`
(4)                 :meth:`node_connectivity`
(4)                 :meth:`edge_connectivity`
(4)                 :meth:`maximum_flow`
(4)                 :meth:`edmonds_karp`
```

```
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 References
(4)                 ----------
(4)                 .. [1]  Beineke, L., O. Oellermann, and R. Pippert (2002). The average
(12)                        connectivity of a graph. Discrete mathematics 252(1-3), 31-45.
(12)                        http://www.sciencedirect.com/science/article/pii/S0012365X01001807
(4)                 """
(4)                 if G.is_directed():
(8)                     iter_func = itertools.permutations
(4)                 else:
(8)                     iter_func = itertools.combinations
(4)                 H = build_auxiliary_node_connectivity(G)
(4)                 R = build_residual_network(H, "capacity")
(4)                 kwargs = {"flow_func": flow_func, "auxiliary": H, "residual": R}
(4)                 num, den = 0, 0
(4)                 for u, v in iter_func(G, 2):
(8)                     num += local_node_connectivity(G, u, v, **kwargs)
(8)                     den += 1
(4)                 if den == 0:  # Null Graph
(8)                     return 0
(4)                 return num / den
(0)             @nx._dispatchable
(0)             def all_pairs_node_connectivity(G, nbunch=None, flow_func=None):
(4)                 """Compute node connectivity between all pairs of nodes of G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Undirected graph
(4)                 nbunch: container
(8)                     Container of nodes. If provided node connectivity will be computed
(8)                     only over pairs of nodes in nbunch.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes.
(8)                     The function has to accept at least three parameters: a Digraph,
(8)                     a source node, and a target node. And return a residual network
(8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)                     details). If flow_func is None, the default maximum flow function
(8)                     (:meth:`edmonds_karp`) is used. See below for details. The
(8)                     choice of the default function may change from version
(8)                     to version and should not be relied on. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 all_pairs : dict
(8)                     A dictionary with node connectivity between all pairs of nodes
(8)                     in G, or in nbunch if provided.
(4)                 See also
(4)                 --------
(4)                 :meth:`local_node_connectivity`
(4)                 :meth:`edge_connectivity`
(4)                 :meth:`local_edge_connectivity`
(4)                 :meth:`maximum_flow`
(4)                 :meth:`edmonds_karp`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 """
(4)                 if nbunch is None:
(8)                     nbunch = G
(4)                 else:
(8)                     nbunch = set(nbunch)
(4)                 directed = G.is_directed()
(4)                 if directed:
(8)                     iter_func = itertools.permutations
(4)                 else:
(8)                     iter_func = itertools.combinations
(4)                 all_pairs = {n: {} for n in nbunch}
(4)                 H = build_auxiliary_node_connectivity(G)
(4)                 mapping = H.graph["mapping"]
(4)                 R = build_residual_network(H, "capacity")
```

```
(4)     kwargs = {"flow_func": flow_func, "auxiliary": H, "residual": R}
(4)     for u, v in iter_func(nbunch, 2):
(8)         K = local_node_connectivity(G, u, v, **kwargs)
(8)         all_pairs[u][v] = K
(8)         if not directed:
(12)            all_pairs[v][u] = K
(4)     return all_pairs
(0)  @nx._dispatchable(graphs={"G": 0, "auxiliary?": 4})
(0)  def local_edge_connectivity(
(4)     G, s, t, flow_func=None, auxiliary=None, residual=None, cutoff=None
(0)  ):
(4)     r"""Returns local edge connectivity for nodes s and t in G.
(4)     Local edge connectivity for two nodes s and t is the minimum number
(4)     of edges that must be removed to disconnect them.
(4)     This is a flow based implementation of edge connectivity. We compute the
(4)     maximum flow on an auxiliary digraph build from the original
(4)     network (see below for details). This is equal to the local edge
(4)     connectivity because the value of a maximum s-t-flow is equal to the
(4)     capacity of a minimum s-t-cut (Ford and Fulkerson theorem) [1]_ .
(4)     Parameters
(4)     ----------
(4)     G : NetworkX graph
(8)         Undirected or directed graph
(4)     s : node
(8)         Source node
(4)     t : node
(8)         Target node
(4)     flow_func : function
(8)         A function for computing the maximum flow among a pair of nodes.
(8)         The function has to accept at least three parameters: a Digraph,
(8)         a source node, and a target node. And return a residual network
(8)         that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)         details). If flow_func is None, the default maximum flow function
(8)         (:meth:`edmonds_karp`) is used. See below for details. The
(8)         choice of the default function may change from version
(8)         to version and should not be relied on. Default value: None.
(4)     auxiliary : NetworkX DiGraph
(8)         Auxiliary digraph for computing flow based edge connectivity. If
(8)         provided it will be reused instead of recreated. Default value: None.
(4)     residual : NetworkX DiGraph
(8)         Residual network to compute maximum flow. If provided it will be
(8)         reused instead of recreated. Default value: None.
(4)     cutoff : integer, float, or None (default: None)
(8)         If specified, the maximum flow algorithm will terminate when the
(8)         flow value reaches or exceeds the cutoff. This only works for flows
(8)         that support the cutoff parameter (most do) and is ignored otherwise.
(4)     Returns
(4)     -------
(4)     K : integer
(8)         local edge connectivity for nodes s and t.
(4)     Examples
(4)     --------
(4)     This function is not imported in the base NetworkX namespace, so you
(4)     have to explicitly import it from the connectivity package:
(4)     >>> from networkx.algorithms.connectivity import local_edge_connectivity
(4)     We use in this example the platonic icosahedral graph, which has edge
(4)     connectivity 5.
(4)     >>> G = nx.icosahedral_graph()
(4)     >>> local_edge_connectivity(G, 0, 6)
(4)     5
(4)     If you need to compute local connectivity on several pairs of
(4)     nodes in the same graph, it is recommended that you reuse the
(4)     data structures that NetworkX uses in the computation: the
(4)     auxiliary digraph for edge connectivity, and the residual
(4)     network for the underlying maximum flow computation.
(4)     Example of how to compute local edge connectivity among
(4)     all pairs of nodes of the platonic icosahedral graph reusing
(4)     the data structures.
(4)     >>> import itertools
```

```
(4)                 >>> # You also have to explicitly import the function for
(4)                 >>> # building the auxiliary digraph from the connectivity package
(4)                 >>> from networkx.algorithms.connectivity import
build_auxiliary_edge_connectivity
(4)                 >>> H = build_auxiliary_edge_connectivity(G)
(4)                 >>> # And the function for building the residual network from the
(4)                 >>> # flow package
(4)                 >>> from networkx.algorithms.flow import build_residual_network
(4)                 >>> # Note that the auxiliary digraph has an edge attribute named capacity
(4)                 >>> R = build_residual_network(H, "capacity")
(4)                 >>> result = dict.fromkeys(G, dict())
(4)                 >>> # Reuse the auxiliary digraph and the residual network by passing them
(4)                 >>> # as parameters
(4)                 >>> for u, v in itertools.combinations(G, 2):
(4)                 ...     k = local_edge_connectivity(G, u, v, auxiliary=H, residual=R)
(4)                 ...     result[u][v] = k
(4)                 >>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
(4)                 True
(4)                 You can also use alternative flow algorithms for computing edge
(4)                 connectivity. For instance, in dense networks the algorithm
(4)                 :meth:`shortest_augmenting_path` will usually perform better than
(4)                 the default :meth:`edmonds_karp` which is faster for sparse
(4)                 networks with highly skewed degree distributions. Alternative flow
(4)                 functions have to be explicitly imported from the flow package.
(4)                 >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)                 >>> local_edge_connectivity(G, 0, 6, flow_func=shortest_augmenting_path)
(4)                 5
(4)                 Notes
(4)                 -----
(4)                 This is a flow based implementation of edge connectivity. We compute the
(4)                 maximum flow using, by default, the :meth:`edmonds_karp` algorithm on an
(4)                 auxiliary digraph build from the original input graph:
(4)                 If the input graph is undirected, we replace each edge (`u`,`v`) with
(4)                 two reciprocal arcs (`u`, `v`) and (`v`, `u`) and then we set the
attribute
(4)                 'capacity' for each arc to 1. If the input graph is directed we simply
(4)                 add the 'capacity' attribute. This is an implementation of algorithm 1
(4)                 in [1]_.
(4)                 The maximum flow in the auxiliary network is equal to the local edge
(4)                 connectivity because the value of a maximum s-t-flow is equal to the
(4)                 capacity of a minimum s-t-cut (Ford and Fulkerson theorem).
(4)                 See also
(4)                 --------
(4)                 :meth:`edge_connectivity`
(4)                 :meth:`local_node_connectivity`
(4)                 :meth:`node_connectivity`
(4)                 :meth:`maximum_flow`
(4)                 :meth:`edmonds_karp`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 References
(4)                 ----------
(4)                 .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms.
(8)                     http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
(4)                 """
(4)                 if flow_func is None:
(8)                     flow_func = default_flow_func
(4)                 if auxiliary is None:
(8)                     H = build_auxiliary_edge_connectivity(G)
(4)                 else:
(8)                     H = auxiliary
(4)                 kwargs = {"flow_func": flow_func, "residual": residual}
(4)                 if flow_func is shortest_augmenting_path:
(8)                     kwargs["cutoff"] = cutoff
(8)                     kwargs["two_phase"] = True
(4)                 elif flow_func is edmonds_karp:
(8)                     kwargs["cutoff"] = cutoff
(4)                 elif flow_func is dinitz:
(8)                     kwargs["cutoff"] = cutoff
```

```
(4)         elif flow_func is boykov_kolmogorov:
(8)             kwargs["cutoff"] = cutoff
(4)         return nx.maximum_flow_value(H, s, t, **kwargs)
(0)     @nx._dispatchable
(0)     def edge_connectivity(G, s=None, t=None, flow_func=None, cutoff=None):
(4)         r"""Returns the edge connectivity of the graph or digraph G.
(4)         The edge connectivity is equal to the minimum number of edges that
(4)         must be removed to disconnect G or render it trivial. If source
(4)         and target nodes are provided, this function returns the local edge
(4)         connectivity: the minimum number of edges that must be removed to
(4)         break all paths from source to target in G.
(4)         Parameters
(4)         ----------
(4)         G : NetworkX graph
(8)             Undirected or directed graph
(4)         s : node
(8)             Source node. Optional. Default value: None.
(4)         t : node
(8)             Target node. Optional. Default value: None.
(4)         flow_func : function
(8)             A function for computing the maximum flow among a pair of nodes.
(8)             The function has to accept at least three parameters: a Digraph,
(8)             a source node, and a target node. And return a residual network
(8)             that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)             details). If flow_func is None, the default maximum flow function
(8)             (:meth:`edmonds_karp`) is used. See below for details. The
(8)             choice of the default function may change from version
(8)             to version and should not be relied on. Default value: None.
(4)         cutoff : integer, float, or None (default: None)
(8)             If specified, the maximum flow algorithm will terminate when the
(8)             flow value reaches or exceeds the cutoff. This only works for flows
(8)             that support the cutoff parameter (most do) and is ignored otherwise.
(4)         Returns
(4)         -------
(4)         K : integer
(8)             Edge connectivity for G, or local edge connectivity if source
(8)             and target were provided
(4)         Examples
(4)         --------
(4)         >>> # Platonic icosahedral graph is 5-edge-connected
(4)         >>> G = nx.icosahedral_graph()
(4)         >>> nx.edge_connectivity(G)
(4)         5
(4)         You can use alternative flow algorithms for the underlying
(4)         maximum flow computation. In dense networks the algorithm
(4)         :meth:`shortest_augmenting_path` will usually perform better
(4)         than the default :meth:`edmonds_karp`, which is faster for
(4)         sparse networks with highly skewed degree distributions.
(4)         Alternative flow functions have to be explicitly imported
(4)         from the flow package.
(4)         >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)         >>> nx.edge_connectivity(G, flow_func=shortest_augmenting_path)
(4)         5
(4)         If you specify a pair of nodes (source and target) as parameters,
(4)         this function returns the value of local edge connectivity.
(4)         >>> nx.edge_connectivity(G, 3, 7)
(4)         5
(4)         If you need to perform several local computations among different
(4)         pairs of nodes on the same graph, it is recommended that you reuse
(4)         the data structures used in the maximum flow computations. See
(4)         :meth:`local_edge_connectivity` for details.
(4)         Notes
(4)         -----
(4)         This is a flow based implementation of global edge connectivity.
(4)         For undirected graphs the algorithm works by finding a 'small'
(4)         dominating set of nodes of G (see algorithm 7 in [1]_ ) and
(4)         computing local maximum flow (see :meth:`local_edge_connectivity`)
(4)         between an arbitrary node in the dominating set and the rest of
(4)         nodes in it. This is an implementation of algorithm 6 in [1]_ .
```

```
    For directed graphs, the algorithm does n calls to the maximum
    flow function. This is an implementation of algorithm 8 in [1]_ .
    See also
    --------
    :meth:`local_edge_connectivity`
    :meth:`local_node_connectivity`
    :meth:`node_connectivity`
    :meth:`maximum_flow`
    :meth:`edmonds_karp`
    :meth:`preflow_push`
    :meth:`shortest_augmenting_path`
    :meth:`k_edge_components`
    :meth:`k_edge_subgraphs`
    References
    ----------
    .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms.
        http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
    """
    if (s is not None and t is None) or (s is None and t is not None):
        raise nx.NetworkXError("Both source and target must be specified.")
    if s is not None and t is not None:
        if s not in G:
            raise nx.NetworkXError(f"node {s} not in graph")
        if t not in G:
            raise nx.NetworkXError(f"node {t} not in graph")
        return local_edge_connectivity(G, s, t, flow_func=flow_func, cutoff=cutoff)
    H = build_auxiliary_edge_connectivity(G)
    R = build_residual_network(H, "capacity")
    kwargs = {"flow_func": flow_func, "auxiliary": H, "residual": R}
    if G.is_directed():
        if not nx.is_weakly_connected(G):
            return 0
        L = min(d for n, d in G.degree())
        nodes = list(G)
        n = len(nodes)
        if cutoff is not None:
            L = min(cutoff, L)
        for i in range(n):
            kwargs["cutoff"] = L
            try:
                L = min(L, local_edge_connectivity(G, nodes[i], nodes[i + 1], **kwargs))
            except IndexError:  # last node!
                L = min(L, local_edge_connectivity(G, nodes[i], nodes[0], **kwargs))
        return L
    else:  # undirected
        if not nx.is_connected(G):
            return 0
        L = min(d for n, d in G.degree())
        if cutoff is not None:
            L = min(cutoff, L)
        for node in G:
            D = nx.dominating_set(G, start_with=node)
            v = D.pop()
            if D:
                break
        else:
            return L
        for w in D:
            kwargs["cutoff"] = L
            L = min(L, local_edge_connectivity(G, v, w, **kwargs))
        return L
```

File 219 - cuts.py:

```
1: (0)              """
2: (0)              Flow based cut algorithms
3: (0)              """
4: (0)              import itertools
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.flow import build_residual_network, edmonds_karp
7: (0)              default_flow_func = edmonds_karp
8: (0)              from .utils import build_auxiliary_edge_connectivity,
build_auxiliary_node_connectivity
9: (0)              __all__ = [
10: (4)                 "minimum_st_node_cut",
11: (4)                 "minimum_node_cut",
12: (4)                 "minimum_st_edge_cut",
13: (4)                 "minimum_edge_cut",
14: (0)             ]
15: (0)             @nx._dispatchable(
16: (4)                 graphs={"G": 0, "auxiliary?": 4},
17: (4)                 preserve_edge_attrs={"auxiliary": {"capacity": float("inf")}},
18: (4)                 preserve_graph_attrs={"auxiliary"},
19: (0)             )
20: (0)             def minimum_st_edge_cut(G, s, t, flow_func=None, auxiliary=None,
residual=None):
21: (4)                 """Returns the edges of the cut-set of a minimum (s, t)-cut.
22: (4)                 This function returns the set of edges of minimum cardinality that,
23: (4)                 if removed, would destroy all paths among source and target in G.
24: (4)                 Edge weights are not considered. See :meth:`minimum_cut` for
25: (4)                 computing minimum cuts considering edge weights.
26: (4)                 Parameters
27: (4)                 ----------
28: (4)                 G : NetworkX graph
29: (4)                 s : node
30: (8)                     Source node for the flow.
31: (4)                 t : node
32: (8)                     Sink node for the flow.
33: (4)                 auxiliary : NetworkX DiGraph
34: (8)                     Auxiliary digraph to compute flow based node connectivity. It has
35: (8)                     to have a graph attribute called mapping with a dictionary mapping
36: (8)                     node names in G and in the auxiliary digraph. If provided
37: (8)                     it will be reused instead of recreated. Default value: None.
38: (4)                 flow_func : function
39: (8)                     A function for computing the maximum flow among a pair of nodes.
40: (8)                     The function has to accept at least three parameters: a Digraph,
41: (8)                     a source node, and a target node. And return a residual network
42: (8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
43: (8)                     details). If flow_func is None, the default maximum flow function
44: (8)                     (:meth:`edmonds_karp`) is used. See :meth:`node_connectivity` for
45: (8)                     details. The choice of the default function may change from version
46: (8)                     to version and should not be relied on. Default value: None.
47: (4)                 residual : NetworkX DiGraph
48: (8)                     Residual network to compute maximum flow. If provided it will be
49: (8)                     reused instead of recreated. Default value: None.
50: (4)                 Returns
51: (4)                 -------
52: (4)                 cutset : set
53: (8)                     Set of edges that, if removed from the graph, will disconnect it.
54: (4)                 See also
55: (4)                 --------
56: (4)                 :meth:`minimum_cut`
57: (4)                 :meth:`minimum_node_cut`
58: (4)                 :meth:`minimum_edge_cut`
59: (4)                 :meth:`stoer_wagner`
60: (4)                 :meth:`node_connectivity`
61: (4)                 :meth:`edge_connectivity`
62: (4)                 :meth:`maximum_flow`
63: (4)                 :meth:`edmonds_karp`
64: (4)                 :meth:`preflow_push`
65: (4)                 :meth:`shortest_augmenting_path`
66: (4)                 Examples
67: (4)                 --------
```

```
68: (4)                 This function is not imported in the base NetworkX namespace, so you
69: (4)                 have to explicitly import it from the connectivity package:
70: (4)                 >>> from networkx.algorithms.connectivity import minimum_st_edge_cut
71: (4)                 We use in this example the platonic icosahedral graph, which has edge
72: (4)                 connectivity 5.
73: (4)                 >>> G = nx.icosahedral_graph()
74: (4)                 >>> len(minimum_st_edge_cut(G, 0, 6))
75: (4)                 5
76: (4)                 If you need to compute local edge cuts on several pairs of
77: (4)                 nodes in the same graph, it is recommended that you reuse the
78: (4)                 data structures that NetworkX uses in the computation: the
79: (4)                 auxiliary digraph for edge connectivity, and the residual
80: (4)                 network for the underlying maximum flow computation.
81: (4)                 Example of how to compute local edge cuts among all pairs of
82: (4)                 nodes of the platonic icosahedral graph reusing the data
83: (4)                 structures.
84: (4)                 >>> import itertools
85: (4)                 >>> # You also have to explicitly import the function for
86: (4)                 >>> # building the auxiliary digraph from the connectivity package
87: (4)                 >>> from networkx.algorithms.connectivity import
build_auxiliary_edge_connectivity
88: (4)                 >>> H = build_auxiliary_edge_connectivity(G)
89: (4)                 >>> # And the function for building the residual network from the
90: (4)                 >>> # flow package
91: (4)                 >>> from networkx.algorithms.flow import build_residual_network
92: (4)                 >>> # Note that the auxiliary digraph has an edge attribute named capacity
93: (4)                 >>> R = build_residual_network(H, "capacity")
94: (4)                 >>> result = dict.fromkeys(G, dict())
95: (4)                 >>> # Reuse the auxiliary digraph and the residual network by passing them
96: (4)                 >>> # as parameters
97: (4)                 >>> for u, v in itertools.combinations(G, 2):
98: (4)                 ...     k = len(minimum_st_edge_cut(G, u, v, auxiliary=H, residual=R))
99: (4)                 ...     result[u][v] = k
100: (4)                >>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
101: (4)                True
102: (4)                You can also use alternative flow algorithms for computing edge
103: (4)                cuts. For instance, in dense networks the algorithm
104: (4)                :meth:`shortest_augmenting_path` will usually perform better than
105: (4)                the default :meth:`edmonds_karp` which is faster for sparse
106: (4)                networks with highly skewed degree distributions. Alternative flow
107: (4)                functions have to be explicitly imported from the flow package.
108: (4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
109: (4)                >>> len(minimum_st_edge_cut(G, 0, 6, flow_func=shortest_augmenting_path))
110: (4)                5
111: (4)                """
112: (4)                if flow_func is None:
113: (8)                    flow_func = default_flow_func
114: (4)                if auxiliary is None:
115: (8)                    H = build_auxiliary_edge_connectivity(G)
116: (4)                else:
117: (8)                    H = auxiliary
118: (4)                kwargs = {"capacity": "capacity", "flow_func": flow_func, "residual":
residual}
119: (4)                cut_value, partition = nx.minimum_cut(H, s, t, **kwargs)
120: (4)                reachable, non_reachable = partition
121: (4)                cutset = set()
122: (4)                for u, nbrs in ((n, G[n]) for n in reachable):
123: (8)                    cutset.update((u, v) for v in nbrs if v in non_reachable)
124: (4)                return cutset
125: (0)            @nx._dispatchable(
126: (4)                graphs={"G": 0, "auxiliary?": 4},
127: (4)                preserve_node_attrs={"auxiliary": {"id": None}},
128: (4)                preserve_graph_attrs={"auxiliary"},
129: (0)            )
130: (0)            def minimum_st_node_cut(G, s, t, flow_func=None, auxiliary=None,
residual=None):
131: (4)                r"""Returns a set of nodes of minimum cardinality that disconnect source
132: (4)                from target in G.
133: (4)                This function returns the set of nodes of minimum cardinality that,
```

```
(4)                 if removed, would destroy all paths among source and target in G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 s : node
(8)                     Source node.
(4)                 t : node
(8)                     Target node.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes.
(8)                     The function has to accept at least three parameters: a Digraph,
(8)                     a source node, and a target node. And return a residual network
(8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)                     details). If flow_func is None, the default maximum flow function
(8)                     (:meth:`edmonds_karp`) is used. See below for details. The choice
(8)                     of the default function may change from version to version and
(8)                     should not be relied on. Default value: None.
(4)                 auxiliary : NetworkX DiGraph
(8)                     Auxiliary digraph to compute flow based node connectivity. It has
(8)                     to have a graph attribute called mapping with a dictionary mapping
(8)                     node names in G and in the auxiliary digraph. If provided
(8)                     it will be reused instead of recreated. Default value: None.
(4)                 residual : NetworkX DiGraph
(8)                     Residual network to compute maximum flow. If provided it will be
(8)                     reused instead of recreated. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 cutset : set
(8)                     Set of nodes that, if removed, would destroy all paths between
(8)                     source and target in G.
(4)                 Examples
(4)                 --------
(4)                 This function is not imported in the base NetworkX namespace, so you
(4)                 have to explicitly import it from the connectivity package:
(4)                 >>> from networkx.algorithms.connectivity import minimum_st_node_cut
(4)                 We use in this example the platonic icosahedral graph, which has node
(4)                 connectivity 5.
(4)                 >>> G = nx.icosahedral_graph()
(4)                 >>> len(minimum_st_node_cut(G, 0, 6))
(4)                 5
(4)                 If you need to compute local st cuts between several pairs of
(4)                 nodes in the same graph, it is recommended that you reuse the
(4)                 data structures that NetworkX uses in the computation: the
(4)                 auxiliary digraph for node connectivity and node cuts, and the
(4)                 residual network for the underlying maximum flow computation.
(4)                 Example of how to compute local st node cuts reusing the data
(4)                 structures:
(4)                 >>> # You also have to explicitly import the function for
(4)                 >>> # building the auxiliary digraph from the connectivity package
(4)                 >>> from networkx.algorithms.connectivity import
build_auxiliary_node_connectivity
(4)                 >>> H = build_auxiliary_node_connectivity(G)
(4)                 >>> # And the function for building the residual network from the
(4)                 >>> # flow package
(4)                 >>> from networkx.algorithms.flow import build_residual_network
(4)                 >>> # Note that the auxiliary digraph has an edge attribute named capacity
(4)                 >>> R = build_residual_network(H, "capacity")
(4)                 >>> # Reuse the auxiliary digraph and the residual network by passing them
(4)                 >>> # as parameters
(4)                 >>> len(minimum_st_node_cut(G, 0, 6, auxiliary=H, residual=R))
(4)                 5
(4)                 You can also use alternative flow algorithms for computing minimum st
(4)                 node cuts. For instance, in dense networks the algorithm
(4)                 :meth:`shortest_augmenting_path` will usually perform better than
(4)                 the default :meth:`edmonds_karp` which is faster for sparse
(4)                 networks with highly skewed degree distributions. Alternative flow
(4)                 functions have to be explicitly imported from the flow package.
(4)                 >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)                 >>> len(minimum_st_node_cut(G, 0, 6, flow_func=shortest_augmenting_path))
```

```
(4)                 5
(4)                 Notes
(4)                 -----
(4)                 This is a flow based implementation of minimum node cut. The algorithm
(4)                 is based in solving a number of maximum flow computations to determine
(4)                 the capacity of the minimum cut on an auxiliary directed network that
(4)                 corresponds to the minimum node cut of G. It handles both directed
(4)                 and undirected graphs. This implementation is based on algorithm 11
(4)                 in [1]_.
(4)                 See also
(4)                 --------
(4)                 :meth:`minimum_node_cut`
(4)                 :meth:`minimum_edge_cut`
(4)                 :meth:`stoer_wagner`
(4)                 :meth:`node_connectivity`
(4)                 :meth:`edge_connectivity`
(4)                 :meth:`maximum_flow`
(4)                 :meth:`edmonds_karp`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 References
(4)                 ----------
(4)                 .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms.
(8)                     http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
(4)                 """
(4)                 if auxiliary is None:
(8)                     H = build_auxiliary_node_connectivity(G)
(4)                 else:
(8)                     H = auxiliary
(4)                 mapping = H.graph.get("mapping", None)
(4)                 if mapping is None:
(8)                     raise nx.NetworkXError("Invalid auxiliary digraph.")
(4)                 if G.has_edge(s, t) or G.has_edge(t, s):
(8)                     return {}
(4)                 kwargs = {"flow_func": flow_func, "residual": residual, "auxiliary": H}
(4)                 edge_cut = minimum_st_edge_cut(H, f"{mapping[s]}B", f"{mapping[t]}A",
**kwargs)
(4)                 node_cut = {H.nodes[node]["id"] for edge in edge_cut for node in edge}
(4)                 return node_cut - {s, t}
(0)             @nx._dispatchable
(0)             def minimum_node_cut(G, s=None, t=None, flow_func=None):
(4)                 r"""Returns a set of nodes of minimum cardinality that disconnects G.
(4)                 If source and target nodes are provided, this function returns the
(4)                 set of nodes of minimum cardinality that, if removed, would destroy
(4)                 all paths among source and target in G. If not, it returns a set
(4)                 of nodes of minimum cardinality that disconnects G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 s : node
(8)                     Source node. Optional. Default value: None.
(4)                 t : node
(8)                     Target node. Optional. Default value: None.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes.
(8)                     The function has to accept at least three parameters: a Digraph,
(8)                     a source node, and a target node. And return a residual network
(8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)                     details). If flow_func is None, the default maximum flow function
(8)                     (:meth:`edmonds_karp`) is used. See below for details. The
(8)                     choice of the default function may change from version
(8)                     to version and should not be relied on. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 cutset : set
(8)                     Set of nodes that, if removed, would disconnect G. If source
(8)                     and target nodes are provided, the set contains the nodes that
(8)                     if removed, would destroy all paths between source and target.
(4)                 Examples
```

```
    --------
    >>> # Platonic icosahedral graph has node connectivity 5
    >>> G = nx.icosahedral_graph()
    >>> node_cut = nx.minimum_node_cut(G)
    >>> len(node_cut)
    5
    You can use alternative flow algorithms for the underlying maximum
    flow computation. In dense networks the algorithm
    :meth:`shortest_augmenting_path` will usually perform better
    than the default :meth:`edmonds_karp`, which is faster for
    sparse networks with highly skewed degree distributions. Alternative
    flow functions have to be explicitly imported from the flow package.
    >>> from networkx.algorithms.flow import shortest_augmenting_path
    >>> node_cut == nx.minimum_node_cut(G, flow_func=shortest_augmenting_path)
    True
    If you specify a pair of nodes (source and target) as parameters,
    this function returns a local st node cut.
    >>> len(nx.minimum_node_cut(G, 3, 7))
    5
    If you need to perform several local st cuts among different
    pairs of nodes on the same graph, it is recommended that you reuse
    the data structures used in the maximum flow computations. See
    :meth:`minimum_st_node_cut` for details.
    Notes
    -----
    This is a flow based implementation of minimum node cut. The algorithm
    is based in solving a number of maximum flow computations to determine
    the capacity of the minimum cut on an auxiliary directed network that
    corresponds to the minimum node cut of G. It handles both directed
    and undirected graphs. This implementation is based on algorithm 11
    in [1]_.
    See also
    --------
    :meth:`minimum_st_node_cut`
    :meth:`minimum_cut`
    :meth:`minimum_edge_cut`
    :meth:`stoer_wagner`
    :meth:`node_connectivity`
    :meth:`edge_connectivity`
    :meth:`maximum_flow`
    :meth:`edmonds_karp`
    :meth:`preflow_push`
    :meth:`shortest_augmenting_path`
    References
    ----------
    .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms.
        http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
    """
    if (s is not None and t is None) or (s is None and t is not None):
        raise nx.NetworkXError("Both source and target must be specified.")
    if s is not None and t is not None:
        if s not in G:
            raise nx.NetworkXError(f"node {s} not in graph")
        if t not in G:
            raise nx.NetworkXError(f"node {t} not in graph")
        return minimum_st_node_cut(G, s, t, flow_func=flow_func)
    if G.is_directed():
        if not nx.is_weakly_connected(G):
            raise nx.NetworkXError("Input graph is not connected")
        iter_func = itertools.permutations
        def neighbors(v):
            return itertools.chain.from_iterable([G.predecessors(v), G.successors(v)])
    else:
        if not nx.is_connected(G):
            raise nx.NetworkXError("Input graph is not connected")
        iter_func = itertools.combinations
        neighbors = G.neighbors
    H = build_auxiliary_node_connectivity(G)
```

```
(4)                 R = build_residual_network(H, "capacity")
(4)                 kwargs = {"flow_func": flow_func, "auxiliary": H, "residual": R}
(4)                 v = min(G, key=G.degree)
(4)                 min_cut = set(G[v])
(4)                 for w in set(G) - set(neighbors(v)) - {v}:
(8)                     this_cut = minimum_st_node_cut(G, v, w, **kwargs)
(8)                     if len(min_cut) >= len(this_cut):
(12)                        min_cut = this_cut
(4)                 for x, y in iter_func(neighbors(v), 2):
(8)                     if y in G[x]:
(12)                        continue
(8)                     this_cut = minimum_st_node_cut(G, x, y, **kwargs)
(8)                     if len(min_cut) >= len(this_cut):
(12)                        min_cut = this_cut
(4)                 return min_cut
(0)             @nx._dispatchable
(0)             def minimum_edge_cut(G, s=None, t=None, flow_func=None):
(4)                 r"""Returns a set of edges of minimum cardinality that disconnects G.
(4)                 If source and target nodes are provided, this function returns the
(4)                 set of edges of minimum cardinality that, if removed, would break
(4)                 all paths among source and target in G. If not, it returns a set of
(4)                 edges of minimum cardinality that disconnects G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 s : node
(8)                     Source node. Optional. Default value: None.
(4)                 t : node
(8)                     Target node. Optional. Default value: None.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes.
(8)                     The function has to accept at least three parameters: a Digraph,
(8)                     a source node, and a target node. And return a residual network
(8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
(8)                     details). If flow_func is None, the default maximum flow function
(8)                     (:meth:`edmonds_karp`) is used. See below for details. The
(8)                     choice of the default function may change from version
(8)                     to version and should not be relied on. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 cutset : set
(8)                     Set of edges that, if removed, would disconnect G. If source
(8)                     and target nodes are provided, the set contains the edges that
(8)                     if removed, would destroy all paths between source and target.
(4)                 Examples
(4)                 --------
(4)                 >>> # Platonic icosahedral graph has edge connectivity 5
(4)                 >>> G = nx.icosahedral_graph()
(4)                 >>> len(nx.minimum_edge_cut(G))
(4)                 5
(4)                 You can use alternative flow algorithms for the underlying
(4)                 maximum flow computation. In dense networks the algorithm
(4)                 :meth:`shortest_augmenting_path` will usually perform better
(4)                 than the default :meth:`edmonds_karp`, which is faster for
(4)                 sparse networks with highly skewed degree distributions.
(4)                 Alternative flow functions have to be explicitly imported
(4)                 from the flow package.
(4)                 >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)                 >>> len(nx.minimum_edge_cut(G, flow_func=shortest_augmenting_path))
(4)                 5
(4)                 If you specify a pair of nodes (source and target) as parameters,
(4)                 this function returns the value of local edge connectivity.
(4)                 >>> nx.edge_connectivity(G, 3, 7)
(4)                 5
(4)                 If you need to perform several local computations among different
(4)                 pairs of nodes on the same graph, it is recommended that you reuse
(4)                 the data structures used in the maximum flow computations. See
(4)                 :meth:`local_edge_connectivity` for details.
(4)                 Notes
```

```
    -----
    This is a flow based implementation of minimum edge cut. For
    undirected graphs the algorithm works by finding a 'small' dominating
    set of nodes of G (see algorithm 7 in [1]_) and computing the maximum
    flow between an arbitrary node in the dominating set and the rest of
    nodes in it. This is an implementation of algorithm 6 in [1]_. For
    directed graphs, the algorithm does n calls to the max flow function.
    The function raises an error if the directed graph is not weakly
    connected and returns an empty set if it is weakly connected.
    It is an implementation of algorithm 8 in [1]_.
    See also
    --------
    :meth:`minimum_st_edge_cut`
    :meth:`minimum_node_cut`
    :meth:`stoer_wagner`
    :meth:`node_connectivity`
    :meth:`edge_connectivity`
    :meth:`maximum_flow`
    :meth:`edmonds_karp`
    :meth:`preflow_push`
    :meth:`shortest_augmenting_path`
    References
    ----------
    .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms.
        http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
    """
    if (s is not None and t is None) or (s is None and t is not None):
        raise nx.NetworkXError("Both source and target must be specified.")
    H = build_auxiliary_edge_connectivity(G)
    R = build_residual_network(H, "capacity")
    kwargs = {"flow_func": flow_func, "residual": R, "auxiliary": H}
    if s is not None and t is not None:
        if s not in G:
            raise nx.NetworkXError(f"node {s} not in graph")
        if t not in G:
            raise nx.NetworkXError(f"node {t} not in graph")
        return minimum_st_edge_cut(H, s, t, **kwargs)
    if G.is_directed():
        if not nx.is_weakly_connected(G):
            raise nx.NetworkXError("Input graph is not connected")
        node = min(G, key=G.degree)
        min_cut = set(G.edges(node))
        nodes = list(G)
        n = len(nodes)
        for i in range(n):
            try:
                this_cut = minimum_st_edge_cut(H, nodes[i], nodes[i + 1], **kwargs)
                if len(this_cut) <= len(min_cut):
                    min_cut = this_cut
            except IndexError:  # Last node!
                this_cut = minimum_st_edge_cut(H, nodes[i], nodes[0], **kwargs)
                if len(this_cut) <= len(min_cut):
                    min_cut = this_cut
        return min_cut
    else:  # undirected
        if not nx.is_connected(G):
            raise nx.NetworkXError("Input graph is not connected")
        node = min(G, key=G.degree)
        min_cut = set(G.edges(node))
        for node in G:
            D = nx.dominating_set(G, start_with=node)
            v = D.pop()
            if D:
                break
        else:
            return min_cut
        for w in D:
```

```
474: (12)                        this_cut = minimum_st_edge_cut(H, v, w, **kwargs)
475: (12)                        if len(this_cut) <= len(min_cut):
476: (16)                            min_cut = this_cut
477: (8)                    return min_cut
```

File 220 - disjoint_paths.py:

```
1: (0)              """Flow based node and edge disjoint paths."""
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.flow import (
4: (4)                  edmonds_karp,
5: (4)                  preflow_push,
6: (4)                  shortest_augmenting_path,
7: (0)              )
8: (0)              from networkx.exception import NetworkXNoPath
9: (0)              default_flow_func = edmonds_karp
10: (0)             from itertools import filterfalse as _filterfalse
11: (0)             from .utils import build_auxiliary_edge_connectivity,
build_auxiliary_node_connectivity
12: (0)             __all__ = ["edge_disjoint_paths", "node_disjoint_paths"]
13: (0)             @nx._dispatchable(
14: (4)                 graphs={"G": 0, "auxiliary?": 5},
15: (4)                 preserve_edge_attrs={"auxiliary": {"capacity": float("inf")}},
16: (0)             )
17: (0)             def edge_disjoint_paths(
18: (4)                 G, s, t, flow_func=None, cutoff=None, auxiliary=None, residual=None
19: (0)             ):
20: (4)                 """Returns the edges disjoint paths between source and target.
21: (4)                 Edge disjoint paths are paths that do not share any edge. The
22: (4)                 number of edge disjoint paths between source and target is equal
23: (4)                 to their edge connectivity.
24: (4)                 Parameters
25: (4)                 ----------
26: (4)                 G : NetworkX graph
27: (4)                 s : node
28: (8)                     Source node for the flow.
29: (4)                 t : node
30: (8)                     Sink node for the flow.
31: (4)                 flow_func : function
32: (8)                     A function for computing the maximum flow among a pair of nodes.
33: (8)                     The function has to accept at least three parameters: a Digraph,
34: (8)                     a source node, and a target node. And return a residual network
35: (8)                     that follows NetworkX conventions (see :meth:`maximum_flow` for
36: (8)                     details). If flow_func is None, the default maximum flow function
37: (8)                     (:meth:`edmonds_karp`) is used. The choice of the default function
38: (8)                     may change from version to version and should not be relied on.
39: (8)                     Default value: None.
40: (4)                 cutoff : integer or None (default: None)
41: (8)                     Maximum number of paths to yield. If specified, the maximum flow
42: (8)                     algorithm will terminate when the flow value reaches or exceeds the
43: (8)                     cutoff. This only works for flows that support the cutoff parameter
44: (8)                     (most do) and is ignored otherwise.
45: (4)                 auxiliary : NetworkX DiGraph
46: (8)                     Auxiliary digraph to compute flow based edge connectivity. It has
47: (8)                     to have a graph attribute called mapping with a dictionary mapping
48: (8)                     node names in G and in the auxiliary digraph. If provided
49: (8)                     it will be reused instead of recreated. Default value: None.
50: (4)                 residual : NetworkX DiGraph
51: (8)                     Residual network to compute maximum flow. If provided it will be
52: (8)                     reused instead of recreated. Default value: None.
53: (4)                 Returns
54: (4)                 -------
55: (4)                 paths : generator
56: (8)                     A generator of edge independent paths.
57: (4)                 Raises
58: (4)                 ------
59: (4)                 NetworkXNoPath
```

```
60: (8)                     If there is no path between source and target.
61: (4)                 NetworkXError
62: (8)                     If source or target are not in the graph G.
63: (4)                 See also
64: (4)                 --------
65: (4)                 :meth:`node_disjoint_paths`
66: (4)                 :meth:`edge_connectivity`
67: (4)                 :meth:`maximum_flow`
68: (4)                 :meth:`edmonds_karp`
69: (4)                 :meth:`preflow_push`
70: (4)                 :meth:`shortest_augmenting_path`
71: (4)                 Examples
72: (4)                 --------
73: (4)                 We use in this example the platonic icosahedral graph, which has node
74: (4)                 edge connectivity 5, thus there are 5 edge disjoint paths between any
75: (4)                 pair of nodes.
76: (4)                 >>> G = nx.icosahedral_graph()
77: (4)                 >>> len(list(nx.edge_disjoint_paths(G, 0, 6)))
78: (4)                 5
79: (4)                 If you need to compute edge disjoint paths on several pairs of
80: (4)                 nodes in the same graph, it is recommended that you reuse the
81: (4)                 data structures that NetworkX uses in the computation: the
82: (4)                 auxiliary digraph for edge connectivity, and the residual
83: (4)                 network for the underlying maximum flow computation.
84: (4)                 Example of how to compute edge disjoint paths among all pairs of
85: (4)                 nodes of the platonic icosahedral graph reusing the data
86: (4)                 structures.
87: (4)                 >>> import itertools
88: (4)                 >>> # You also have to explicitly import the function for
89: (4)                 >>> # building the auxiliary digraph from the connectivity package
90: (4)                 >>> from networkx.algorithms.connectivity import
build_auxiliary_edge_connectivity
91: (4)                 >>> H = build_auxiliary_edge_connectivity(G)
92: (4)                 >>> # And the function for building the residual network from the
93: (4)                 >>> # flow package
94: (4)                 >>> from networkx.algorithms.flow import build_residual_network
95: (4)                 >>> # Note that the auxiliary digraph has an edge attribute named capacity
96: (4)                 >>> R = build_residual_network(H, "capacity")
97: (4)                 >>> result = {n: {} for n in G}
98: (4)                 >>> # Reuse the auxiliary digraph and the residual network by passing them
99: (4)                 >>> # as arguments
100: (4)                >>> for u, v in itertools.combinations(G, 2):
101: (4)                ...     k = len(list(nx.edge_disjoint_paths(G, u, v, auxiliary=H,
residual=R)))
102: (4)                ...     result[u][v] = k
103: (4)                >>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
104: (4)                True
105: (4)                You can also use alternative flow algorithms for computing edge disjoint
106: (4)                paths. For instance, in dense networks the algorithm
107: (4)                :meth:`shortest_augmenting_path` will usually perform better than
108: (4)                the default :meth:`edmonds_karp` which is faster for sparse
109: (4)                networks with highly skewed degree distributions. Alternative flow
110: (4)                functions have to be explicitly imported from the flow package.
111: (4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
112: (4)                >>> len(list(nx.edge_disjoint_paths(G, 0, 6,
flow_func=shortest_augmenting_path)))
113: (4)                5
114: (4)                Notes
115: (4)                -----
116: (4)                This is a flow based implementation of edge disjoint paths. We compute
117: (4)                the maximum flow between source and target on an auxiliary directed
118: (4)                network. The saturated edges in the residual network after running the
119: (4)                maximum flow algorithm correspond to edge disjoint paths between source
120: (4)                and target in the original network. This function handles both directed
121: (4)                and undirected graphs, and can use all flow algorithms from NetworkX flow
122: (4)                package.
123: (4)                """
124: (4)                if s not in G:
125: (8)                    raise nx.NetworkXError(f"node {s} not in graph")
```

```
(4)                 if t not in G:
(8)                     raise nx.NetworkXError(f"node {t} not in graph")
(4)                 if flow_func is None:
(8)                     flow_func = default_flow_func
(4)                 if auxiliary is None:
(8)                     H = build_auxiliary_edge_connectivity(G)
(4)                 else:
(8)                     H = auxiliary
(4)                 possible = min(H.out_degree(s), H.in_degree(t))
(4)                 if not possible:
(8)                     raise NetworkXNoPath
(4)                 if cutoff is None:
(8)                     cutoff = possible
(4)                 else:
(8)                     cutoff = min(cutoff, possible)
(4)                 kwargs = {
(8)                     "capacity": "capacity",
(8)                     "residual": residual,
(8)                     "cutoff": cutoff,
(8)                     "value_only": True,
(4)                 }
(4)                 if flow_func is preflow_push:
(8)                     del kwargs["cutoff"]
(4)                 if flow_func is shortest_augmenting_path:
(8)                     kwargs["two_phase"] = True
(4)                 R = flow_func(H, s, t, **kwargs)
(4)                 if R.graph["flow_value"] == 0:
(8)                     raise NetworkXNoPath
(4)                 cutset = [
(8)                     (u, v)
(8)                     for u, v, d in R.edges(data=True)
(8)                     if d["capacity"] == d["flow"] and d["flow"] > 0
(4)                 ]
(4)                 flow_dict = {n: {} for edge in cutset for n in edge}
(4)                 for u, v in cutset:
(8)                     flow_dict[u][v] = 1
(4)                 paths_found = 0
(4)                 for v in list(flow_dict[s]):
(8)                     if paths_found >= cutoff:
(12)                        break
(8)                     path = [s]
(8)                     if v == t:
(12)                        path.append(v)
(12)                        yield path
(12)                        continue
(8)                     u = v
(8)                     while u != t:
(12)                        path.append(u)
(12)                        try:
(16)                            u, _ = flow_dict[u].popitem()
(12)                        except KeyError:
(16)                            break
(8)                     else:
(12)                        path.append(t)
(12)                        yield path
(12)                        paths_found += 1
(0)             @nx._dispatchable(
(4)                 graphs={"G": 0, "auxiliary?": 5},
(4)                 preserve_node_attrs={"auxiliary": {"id": None}},
(4)                 preserve_graph_attrs={"auxiliary"},
(0)             )
(0)             def node_disjoint_paths(
(4)                 G, s, t, flow_func=None, cutoff=None, auxiliary=None, residual=None
(0)             ):
(4)                 r"""Computes node disjoint paths between source and target.
(4)                 Node disjoint paths are paths that only share their first and last
(4)                 nodes. The number of node independent paths between two nodes is
(4)                 equal to their local node connectivity.
(4)                 Parameters
```

```
    ----------
    G : NetworkX graph
    s : node
        Source node.
    t : node
        Target node.
    flow_func : function
        A function for computing the maximum flow among a pair of nodes.
        The function has to accept at least three parameters: a Digraph,
        a source node, and a target node. And return a residual network
        that follows NetworkX conventions (see :meth:`maximum_flow` for
        details). If flow_func is None, the default maximum flow function
        (:meth:`edmonds_karp`) is used. See below for details. The choice
        of the default function may change from version to version and
        should not be relied on. Default value: None.
    cutoff : integer or None (default: None)
        Maximum number of paths to yield. If specified, the maximum flow
        algorithm will terminate when the flow value reaches or exceeds the
        cutoff. This only works for flows that support the cutoff parameter
        (most do) and is ignored otherwise.
    auxiliary : NetworkX DiGraph
        Auxiliary digraph to compute flow based node connectivity. It has
        to have a graph attribute called mapping with a dictionary mapping
        node names in G and in the auxiliary digraph. If provided
        it will be reused instead of recreated. Default value: None.
    residual : NetworkX DiGraph
        Residual network to compute maximum flow. If provided it will be
        reused instead of recreated. Default value: None.
    Returns
    -------
    paths : generator
        Generator of node disjoint paths.
    Raises
    ------
    NetworkXNoPath
        If there is no path between source and target.
    NetworkXError
        If source or target are not in the graph G.
    Examples
    --------
    We use in this example the platonic icosahedral graph, which has node
    connectivity 5, thus there are 5 node disjoint paths between any pair
    of non neighbor nodes.
    >>> G = nx.icosahedral_graph()
    >>> len(list(nx.node_disjoint_paths(G, 0, 6)))
    5
    If you need to compute node disjoint paths between several pairs of
    nodes in the same graph, it is recommended that you reuse the
    data structures that NetworkX uses in the computation: the
    auxiliary digraph for node connectivity and node cuts, and the
    residual network for the underlying maximum flow computation.
    Example of how to compute node disjoint paths reusing the data
    structures:
    >>> # You also have to explicitly import the function for
    >>> # building the auxiliary digraph from the connectivity package
    >>> from networkx.algorithms.connectivity import
build_auxiliary_node_connectivity
    >>> H = build_auxiliary_node_connectivity(G)
    >>> # And the function for building the residual network from the
    >>> # flow package
    >>> from networkx.algorithms.flow import build_residual_network
    >>> # Note that the auxiliary digraph has an edge attribute named capacity
    >>> R = build_residual_network(H, "capacity")
    >>> # Reuse the auxiliary digraph and the residual network by passing them
    >>> # as arguments
    >>> len(list(nx.node_disjoint_paths(G, 0, 6, auxiliary=H, residual=R)))
    5
    You can also use alternative flow algorithms for computing node disjoint
    paths. For instance, in dense networks the algorithm
```

```
263: (4)                :meth:`shortest_augmenting_path` will usually perform better than
264: (4)                the default :meth:`edmonds_karp` which is faster for sparse
265: (4)                networks with highly skewed degree distributions. Alternative flow
266: (4)                functions have to be explicitly imported from the flow package.
267: (4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
268: (4)                >>> len(list(nx.node_disjoint_paths(G, 0, 6,
flow_func=shortest_augmenting_path)))
269: (4)                5
270: (4)                Notes
271: (4)                -----
272: (4)                This is a flow based implementation of node disjoint paths. We compute
273: (4)                the maximum flow between source and target on an auxiliary directed
274: (4)                network. The saturated edges in the residual network after running the
275: (4)                maximum flow algorithm correspond to node disjoint paths between source
276: (4)                and target in the original network. This function handles both directed
277: (4)                and undirected graphs, and can use all flow algorithms from NetworkX flow
278: (4)                package.
279: (4)                See also
280: (4)                --------
281: (4)                :meth:`edge_disjoint_paths`
282: (4)                :meth:`node_connectivity`
283: (4)                :meth:`maximum_flow`
284: (4)                :meth:`edmonds_karp`
285: (4)                :meth:`preflow_push`
286: (4)                :meth:`shortest_augmenting_path`
287: (4)                """
288: (4)                if s not in G:
289: (8)                    raise nx.NetworkXError(f"node {s} not in graph")
290: (4)                if t not in G:
291: (8)                    raise nx.NetworkXError(f"node {t} not in graph")
292: (4)                if auxiliary is None:
293: (8)                    H = build_auxiliary_node_connectivity(G)
294: (4)                else:
295: (8)                    H = auxiliary
296: (4)                mapping = H.graph.get("mapping", None)
297: (4)                if mapping is None:
298: (8)                    raise nx.NetworkXError("Invalid auxiliary digraph.")
299: (4)                possible = min(H.out_degree(f"{mapping[s]}B"), H.in_degree(f"
{mapping[t]}A"))
300: (4)                if not possible:
301: (8)                    raise NetworkXNoPath
302: (4)                if cutoff is None:
303: (8)                    cutoff = possible
304: (4)                else:
305: (8)                    cutoff = min(cutoff, possible)
306: (4)                kwargs = {
307: (8)                    "flow_func": flow_func,
308: (8)                    "residual": residual,
309: (8)                    "auxiliary": H,
310: (8)                    "cutoff": cutoff,
311: (4)                }
312: (4)                paths_edges = edge_disjoint_paths(H, f"{mapping[s]}B", f"{mapping[t]}A",
**kwargs)
313: (4)                for path in paths_edges:
314: (8)                    yield list(_unique_everseen(H.nodes[node]["id"] for node in path))
315: (0)            def _unique_everseen(iterable):
316: (4)                "List unique elements, preserving order. Remember all elements ever seen."
317: (4)                seen = set()
318: (4)                seen_add = seen.add
319: (4)                for element in _filterfalse(seen.__contains__, iterable):
320: (8)                    seen_add(element)
321: (8)                    yield element

----------------------------------------

File 221 - edge_augmentation.py:

1: (0)              """
2: (0)              Algorithms for finding k-edge-augmentations
```

```
3: (0)              A k-edge-augmentation is a set of edges, that once added to a graph, ensures
4: (0)              that the graph is k-edge-connected; i.e. the graph cannot be disconnected
5: (0)              unless k or more edges are removed.  Typically, the goal is to find the
6: (0)              augmentation with minimum weight.  In general, it is not guaranteed that a
7: (0)              k-edge-augmentation exists.
8: (0)              See Also
9: (0)              --------
10: (0)             :mod:`edge_kcomponents` : algorithms for finding k-edge-connected components
11: (0)             :mod:`connectivity` : algorithms for determining edge connectivity.
12: (0)             """
13: (0)             import itertools as it
14: (0)             import math
15: (0)             from collections import defaultdict, namedtuple
16: (0)             import networkx as nx
17: (0)             from networkx.utils import not_implemented_for, py_random_state
18: (0)             __all__ = ["k_edge_augmentation", "is_k_edge_connected",
"is_locally_k_edge_connected"]
19: (0)             @not_implemented_for("directed")
20: (0)             @not_implemented_for("multigraph")
21: (0)             @nx._dispatchable
22: (0)             def is_k_edge_connected(G, k):
23: (4)                 """Tests to see if a graph is k-edge-connected.
24: (4)                 Is it impossible to disconnect the graph by removing fewer than k edges?
25: (4)                 If so, then G is k-edge-connected.
26: (4)                 Parameters
27: (4)                 ----------
28: (4)                 G : NetworkX graph
29: (7)                    An undirected graph.
30: (4)                 k : integer
31: (8)                     edge connectivity to test for
32: (4)                 Returns
33: (4)                 -------
34: (4)                 boolean
35: (8)                     True if G is k-edge-connected.
36: (4)                 See Also
37: (4)                 --------
38: (4)                 :func:`is_locally_k_edge_connected`
39: (4)                 Examples
40: (4)                 --------
41: (4)                 >>> G = nx.barbell_graph(10, 0)
42: (4)                 >>> nx.is_k_edge_connected(G, k=1)
43: (4)                 True
44: (4)                 >>> nx.is_k_edge_connected(G, k=2)
45: (4)                 False
46: (4)                 """
47: (4)                 if k < 1:
48: (8)                     raise ValueError(f"k must be positive, not {k}")
49: (4)                 if G.number_of_nodes() < k + 1:
50: (8)                     return False
51: (4)                 elif any(d < k for n, d in G.degree()):
52: (8)                     return False
53: (4)                 else:
54: (8)                     if k == 1:
55: (12)                        return nx.is_connected(G)
56: (8)                     elif k == 2:
57: (12)                        return nx.is_connected(G) and not nx.has_bridges(G)
58: (8)                     else:
59: (12)                        return nx.edge_connectivity(G, cutoff=k) >= k
60: (0)             @not_implemented_for("directed")
61: (0)             @not_implemented_for("multigraph")
62: (0)             @nx._dispatchable
63: (0)             def is_locally_k_edge_connected(G, s, t, k):
64: (4)                 """Tests to see if an edge in a graph is locally k-edge-connected.
65: (4)                 Is it impossible to disconnect s and t by removing fewer than k edges?
66: (4)                 If so, then s and t are locally k-edge-connected in G.
67: (4)                 Parameters
68: (4)                 ----------
69: (4)                 G : NetworkX graph
70: (7)                    An undirected graph.
```

```
71: (4)                 s : node
72: (8)                     Source node
73: (4)                 t : node
74: (8)                     Target node
75: (4)                 k : integer
76: (8)                     local edge connectivity for nodes s and t
77: (4)                 Returns
78: (4)                 -------
79: (4)                 boolean
80: (8)                     True if s and t are locally k-edge-connected in G.
81: (4)                 See Also
82: (4)                 --------
83: (4)                 :func:`is_k_edge_connected`
84: (4)                 Examples
85: (4)                 --------
86: (4)                 >>> from networkx.algorithms.connectivity import
is_locally_k_edge_connected
87: (4)                 >>> G = nx.barbell_graph(10, 0)
88: (4)                 >>> is_locally_k_edge_connected(G, 5, 15, k=1)
89: (4)                 True
90: (4)                 >>> is_locally_k_edge_connected(G, 5, 15, k=2)
91: (4)                 False
92: (4)                 >>> is_locally_k_edge_connected(G, 1, 5, k=2)
93: (4)                 True
94: (4)                 """
95: (4)                 if k < 1:
96: (8)                     raise ValueError(f"k must be positive, not {k}")
97: (4)                 if G.degree(s) < k or G.degree(t) < k:
98: (8)                     return False
99: (4)                 else:
100: (8)                    if k == 1:
101: (12)                       return nx.has_path(G, s, t)
102: (8)                    else:
103: (12)                       localk = nx.connectivity.local_edge_connectivity(G, s, t,
cutoff=k)
104: (12)                       return localk >= k
105: (0)            @not_implemented_for("directed")
106: (0)            @not_implemented_for("multigraph")
107: (0)            @nx._dispatchable
108: (0)            def k_edge_augmentation(G, k, avail=None, weight=None, partial=False):
109: (4)                """Finds set of edges to k-edge-connect G.
110: (4)                Adding edges from the augmentation to G make it impossible to disconnect G
111: (4)                unless k or more edges are removed. This function uses the most efficient
112: (4)                function available (depending on the value of k and if the problem is
113: (4)                weighted or unweighted) to search for a minimum weight subset of available
114: (4)                edges that k-edge-connects G. In general, finding a k-edge-augmentation is
115: (4)                NP-hard, so solutions are not guaranteed to be minimal. Furthermore, a
116: (4)                k-edge-augmentation may not exist.
117: (4)                Parameters
118: (4)                ----------
119: (4)                G : NetworkX graph
120: (7)                   An undirected graph.
121: (4)                k : integer
122: (8)                    Desired edge connectivity
123: (4)                avail : dict or a set of 2 or 3 tuples
124: (8)                    The available edges that can be used in the augmentation.
125: (8)                    If unspecified, then all edges in the complement of G are available.
126: (8)                    Otherwise, each item is an available edge (with an optional weight).
127: (8)                    In the unweighted case, each item is an edge ``(u, v)``.
128: (8)                    In the weighted case, each item is a 3-tuple ``(u, v, d)`` or a dict
129: (8)                    with items ``(u, v): d``.  The third item, ``d``, can be a dictionary
130: (8)                    or a real number.  If ``d`` is a dictionary ``d[weight]``
131: (8)                    correspondings to the weight.
132: (4)                weight : string
133: (8)                    key to use to find weights if ``avail`` is a set of 3-tuples where the
134: (8)                    third item in each tuple is a dictionary.
135: (4)                partial : boolean
136: (8)                    If partial is True and no feasible k-edge-augmentation exists, then
all
```

```
    a partial k-edge-augmentation is generated. Adding the edges in a
    partial augmentation to G, minimizes the number of k-edge-connected
    components and maximizes the edge connectivity between those
    components. For details, see :func:`partial_k_edge_augmentation`.
Yields
------
edge : tuple
    Edges that, once added to G, would cause G to become k-edge-connected.
    If partial is False, an error is raised if this is not possible.
    Otherwise, generated edges form a partial augmentation, which
    k-edge-connects any part of G where it is possible, and maximally
    connects the remaining parts.
Raises
------
NetworkXUnfeasible
    If partial is False and no k-edge-augmentation exists.
NetworkXNotImplemented
    If the input graph is directed or a multigraph.
ValueError:
    If k is less than 1
Notes
-----
When k=1 this returns an optimal solution.
When k=2 and ``avail`` is None, this returns an optimal solution.
Otherwise when k=2, this returns a 2-approximation of the optimal solution.

For k>3, this problem is NP-hard and this uses a randomized algorithm that
    produces a feasible solution, but provides no guarantees on the
    solution weight.
Examples
--------
>>> # Unweighted cases
>>> G = nx.path_graph((1, 2, 3, 4))
>>> G.add_node(5)
>>> sorted(nx.k_edge_augmentation(G, k=1))
[(1, 5)]
>>> sorted(nx.k_edge_augmentation(G, k=2))
[(1, 5), (5, 4)]
>>> sorted(nx.k_edge_augmentation(G, k=3))
[(1, 4), (1, 5), (2, 5), (3, 5), (4, 5)]
>>> complement = list(nx.k_edge_augmentation(G, k=5, partial=True))
>>> G.add_edges_from(complement)
>>> nx.edge_connectivity(G)
4
>>> # Weighted cases
>>> G = nx.path_graph((1, 2, 3, 4))
>>> G.add_node(5)
>>> # avail can be a tuple with a dict
>>> avail = [(1, 5, {"weight": 11}), (2, 5, {"weight": 10})]
>>> sorted(nx.k_edge_augmentation(G, k=1, avail=avail, weight="weight"))
[(2, 5)]
>>> # or avail can be a 3-tuple with a real number
>>> avail = [(1, 5, 11), (2, 5, 10), (4, 3, 1), (4, 5, 51)]
>>> sorted(nx.k_edge_augmentation(G, k=2, avail=avail))
[(1, 5), (2, 5), (4, 5)]
>>> # or avail can be a dict
>>> avail = {(1, 5): 11, (2, 5): 10, (4, 3): 1, (4, 5): 51}
>>> sorted(nx.k_edge_augmentation(G, k=2, avail=avail))
[(1, 5), (2, 5), (4, 5)]
>>> # If augmentation is infeasible, then a partial solution can be found
>>> avail = {(1, 5): 11}
>>> sorted(nx.k_edge_augmentation(G, k=2, avail=avail, partial=True))
[(1, 5)]
"""
try:
    if k <= 0:
        raise ValueError(f"k must be a positive integer, not {k}")
    elif G.number_of_nodes() < k + 1:
        msg = f"impossible to {k} connect in graph with less than {k + 1}
```

```
nodes"
205: (12)                        raise nx.NetworkXUnfeasible(msg)
206: (8)                    elif avail is not None and len(avail) == 0:
207: (12)                        if not nx.is_k_edge_connected(G, k):
208: (16)                            raise nx.NetworkXUnfeasible("no available edges")
209: (12)                        aug_edges = []
210: (8)                    elif k == 1:
211: (12)                        aug_edges = one_edge_augmentation(
212: (16)                            G, avail=avail, weight=weight, partial=partial
213: (12)                        )
214: (8)                    elif k == 2:
215: (12)                        aug_edges = bridge_augmentation(G, avail=avail, weight=weight)
216: (8)                    else:
217: (12)                        aug_edges = greedy_k_edge_augmentation(
218: (16)                            G, k=k, avail=avail, weight=weight, seed=0
219: (12)                        )
220: (8)                    yield from list(aug_edges)
221: (4)                except nx.NetworkXUnfeasible:
222: (8)                    if partial:
223: (12)                        if avail is None:
224: (16)                            aug_edges = complement_edges(G)
225: (12)                        else:
226: (16)                            aug_edges = partial_k_edge_augmentation(
227: (20)                                G, k=k, avail=avail, weight=weight
228: (16)                            )
229: (12)                        yield from aug_edges
230: (8)                    else:
231: (12)                        raise
232: (0)            @nx._dispatchable
233: (0)            def partial_k_edge_augmentation(G, k, avail, weight=None):
234: (4)                """Finds augmentation that k-edge-connects as much of the graph as
possible.
235: (4)                When a k-edge-augmentation is not possible, we can still try to find a
236: (4)                small set of edges that partially k-edge-connects as much of the graph as
237: (4)                possible. All possible edges are generated between remaining parts.
238: (4)                This minimizes the number of k-edge-connected subgraphs in the resulting
239: (4)                graph and maximizes the edge connectivity between those subgraphs.
240: (4)                Parameters
241: (4)                ----------
242: (4)                G : NetworkX graph
243: (7)                   An undirected graph.
244: (4)                k : integer
245: (8)                    Desired edge connectivity
246: (4)                avail : dict or a set of 2 or 3 tuples
247: (8)                    For more details, see :func:`k_edge_augmentation`.
248: (4)                weight : string
249: (8)                    key to use to find weights if ``avail`` is a set of 3-tuples.
250: (8)                    For more details, see :func:`k_edge_augmentation`.
251: (4)                Yields
252: (4)                ------
253: (4)                edge : tuple
254: (8)                    Edges in the partial augmentation of G. These edges k-edge-connect any
255: (8)                    part of G where it is possible, and maximally connects the remaining
256: (8)                    parts. In other words, all edges from avail are generated except for
257: (8)                    those within subgraphs that have already become k-edge-connected.
258: (4)                Notes
259: (4)                -----
260: (4)                Construct H that augments G with all edges in avail.
261: (4)                Find the k-edge-subgraphs of H.
262: (4)                For each k-edge-subgraph, if the number of nodes is more than k, then find
263: (4)                the k-edge-augmentation of that graph and add it to the solution. Then add
264: (4)                all edges in avail between k-edge subgraphs to the solution.
265: (4)                See Also
266: (4)                --------
267: (4)                :func:`k_edge_augmentation`
268: (4)                Examples
269: (4)                --------
270: (4)                >>> G = nx.path_graph((1, 2, 3, 4, 5, 6, 7))
271: (4)                >>> G.add_node(8)
```

```
    >>> avail = [(1, 3), (1, 4), (1, 5), (2, 4), (2, 5), (3, 5), (1, 8)]
    >>> sorted(partial_k_edge_augmentation(G, k=2, avail=avail))
    [(1, 5), (1, 8)]
    """
    def _edges_between_disjoint(H, only1, only2):
        """finds edges between disjoint nodes"""
        only1_adj = {u: set(H.adj[u]) for u in only1}
        for u, neighbs in only1_adj.items():
            neighbs12 = neighbs.intersection(only2)
            for v in neighbs12:
                yield (u, v)
    avail_uv, avail_w = _unpack_available_edges(avail, weight=weight, G=G)
    H = G.copy()
    H.add_edges_from(
        (
            (u, v, {"weight": w, "generator": (u, v)})
            for (u, v), w in zip(avail, avail_w)
        )
    )
    k_edge_subgraphs = list(nx.k_edge_subgraphs(H, k=k))
    for nodes in k_edge_subgraphs:
        if len(nodes) > 1:
            C = H.subgraph(nodes).copy()
            sub_avail = {
                d["generator"]: d["weight"]
                for (u, v, d) in C.edges(data=True)
                if "generator" in d
            }
            C.remove_edges_from(sub_avail.keys())
            yield from nx.k_edge_augmentation(C, k=k, avail=sub_avail)
    for cc1, cc2 in it.combinations(k_edge_subgraphs, 2):
        for u, v in _edges_between_disjoint(H, cc1, cc2):
            d = H.get_edge_data(u, v)
            edge = d.get("generator", None)
            if edge is not None:
                yield edge
@not_implemented_for("multigraph")
@not_implemented_for("directed")
@nx._dispatchable
def one_edge_augmentation(G, avail=None, weight=None, partial=False):
    """Finds minimum weight set of edges to connect G.
    Equivalent to :func:`k_edge_augmentation` when k=1. Adding the resulting
    edges to G will make it 1-edge-connected. The solution is optimal for both
    weighted and non-weighted variants.
    Parameters
    ----------
    G : NetworkX graph
       An undirected graph.
    avail : dict or a set of 2 or 3 tuples
        For more details, see :func:`k_edge_augmentation`.
    weight : string
        key to use to find weights if ``avail`` is a set of 3-tuples.
        For more details, see :func:`k_edge_augmentation`.
    partial : boolean
        If partial is True and no feasible k-edge-augmentation exists, then the
        augmenting edges minimize the number of connected components.
    Yields
    ------
    edge : tuple
        Edges in the one-augmentation of G
    Raises
    ------
    NetworkXUnfeasible
        If partial is False and no one-edge-augmentation exists.
    Notes
    -----
    Uses either :func:`unconstrained_one_edge_augmentation` or
    :func:`weighted_one_edge_augmentation` depending on whether ``avail`` is
```

```
(4)                 specified. Both algorithms are based on finding a minimum spanning tree.
(4)                 As such both algorithms find optimal solutions and run in linear time.
(4)                 See Also
(4)                 --------
(4)                 :func:`k_edge_augmentation`
(4)                 """
(4)                 if avail is None:
(8)                     return unconstrained_one_edge_augmentation(G)
(4)                 else:
(8)                     return weighted_one_edge_augmentation(
(12)                        G, avail=avail, weight=weight, partial=partial
(8)                     )
(0)             @not_implemented_for("multigraph")
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable
(0)             def bridge_augmentation(G, avail=None, weight=None):
(4)                 """Finds the a set of edges that bridge connects G.
(4)                 Equivalent to :func:`k_edge_augmentation` when k=2, and partial=False.
(4)                 Adding the resulting edges to G will make it 2-edge-connected.  If no
(4)                 constraints are specified the returned set of edges is minimum an optimal,
(4)                 otherwise the solution is approximated.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    An undirected graph.
(4)                 avail : dict or a set of 2 or 3 tuples
(8)                     For more details, see :func:`k_edge_augmentation`.
(4)                 weight : string
(8)                     key to use to find weights if ``avail`` is a set of 3-tuples.
(8)                     For more details, see :func:`k_edge_augmentation`.
(4)                 Yields
(4)                 ------
(4)                 edge : tuple
(8)                     Edges in the bridge-augmentation of G
(4)                 Raises
(4)                 ------
(4)                 NetworkXUnfeasible
(8)                     If no bridge-augmentation exists.
(4)                 Notes
(4)                 -----
(4)                 If there are no constraints the solution can be computed in linear time
(4)                 using :func:`unconstrained_bridge_augmentation`. Otherwise, the problem
(4)                 becomes NP-hard and is the solution is approximated by
(4)                 :func:`weighted_bridge_augmentation`.
(4)                 See Also
(4)                 --------
(4)                 :func:`k_edge_augmentation`
(4)                 """
(4)                 if G.number_of_nodes() < 3:
(8)                     raise nx.NetworkXUnfeasible("impossible to bridge connect less than 3
nodes")
(4)                 if avail is None:
(8)                     return unconstrained_bridge_augmentation(G)
(4)                 else:
(8)                     return weighted_bridge_augmentation(G, avail, weight=weight)
(0)             def _ordered(u, v):
(4)                 """Returns the nodes in an undirected edge in lower-triangular order"""
(4)                 return (u, v) if u < v else (v, u)
(0)             def _unpack_available_edges(avail, weight=None, G=None):
(4)                 """Helper to separate avail into edges and corresponding weights"""
(4)                 if weight is None:
(8)                     weight = "weight"
(4)                 if isinstance(avail, dict):
(8)                     avail_uv = list(avail.keys())
(8)                     avail_w = list(avail.values())
(4)                 else:
(8)                     def _try_getitem(d):
(12)                        try:
(16)                            return d[weight]
```

```
408: (12)                        except TypeError:
409: (16)                            return d
410: (8)                    avail_uv = [tup[0:2] for tup in avail]
411: (8)                    avail_w = [1 if len(tup) == 2 else _try_getitem(tup[-1]) for tup in
avail]
412: (4)                if G is not None:
413: (8)                    flags = [not G.has_edge(u, v) for u, v in avail_uv]
414: (8)                    avail_uv = list(it.compress(avail_uv, flags))
415: (8)                    avail_w = list(it.compress(avail_w, flags))
416: (4)                return avail_uv, avail_w
417: (0)            MetaEdge = namedtuple("MetaEdge", ("meta_uv", "uv", "w"))
418: (0)            def _lightest_meta_edges(mapping, avail_uv, avail_w):
419: (4)                """Maps available edges in the original graph to edges in the metagraph.
420: (4)                Parameters
421: (4)                ----------
422: (4)                mapping : dict
423: (8)                    mapping produced by :func:`collapse`, that maps each node in the
424: (8)                    original graph to a node in the meta graph
425: (4)                avail_uv : list
426: (8)                    list of edges
427: (4)                avail_w : list
428: (8)                    list of edge weights
429: (4)                Notes
430: (4)                -----
431: (4)                Each node in the metagraph is a k-edge-connected component in the original
432: (4)                graph.  We don't care about any edge within the same k-edge-connected
433: (4)                component, so we ignore self edges.  We also are only interested in the
434: (4)                minimum weight edge bridging each k-edge-connected component so, we group
435: (4)                the edges by meta-edge and take the lightest in each group.
436: (4)                Examples
437: (4)                --------
438: (4)                >>> # Each group represents a meta-node
439: (4)                >>> groups = ([1, 2, 3], [4, 5], [6])
440: (4)                >>> mapping = {n: meta_n for meta_n, ns in enumerate(groups) for n in ns}
441: (4)                >>> avail_uv = [(1, 2), (3, 6), (1, 4), (5, 2), (6, 1), (2, 6), (3, 1)]
442: (4)                >>> avail_w = [20, 99, 20, 15, 50, 99, 20]
443: (4)                >>> sorted(_lightest_meta_edges(mapping, avail_uv, avail_w))
444: (4)                [MetaEdge(meta_uv=(0, 1), uv=(5, 2), w=15), MetaEdge(meta_uv=(0, 2), uv=
(6, 1), w=50)]
445: (4)                """
446: (4)                grouped_wuv = defaultdict(list)
447: (4)                for w, (u, v) in zip(avail_w, avail_uv):
448: (8)                    meta_uv = _ordered(mapping[u], mapping[v])
449: (8)                    grouped_wuv[meta_uv].append((w, u, v))
450: (4)                for (mu, mv), choices_wuv in grouped_wuv.items():
451: (8)                    if mu != mv:
452: (12)                       w, u, v = min(choices_wuv)
453: (12)                       yield MetaEdge((mu, mv), (u, v), w)
454: (0)            @nx._dispatchable
455: (0)            def unconstrained_one_edge_augmentation(G):
456: (4)                """Finds the smallest set of edges to connect G.
457: (4)                This is a variant of the unweighted MST problem.
458: (4)                If G is not empty, a feasible solution always exists.
459: (4)                Parameters
460: (4)                ----------
461: (4)                G : NetworkX graph
462: (7)                   An undirected graph.
463: (4)                Yields
464: (4)                ------
465: (4)                edge : tuple
466: (8)                    Edges in the one-edge-augmentation of G
467: (4)                See Also
468: (4)                --------
469: (4)                :func:`one_edge_augmentation`
470: (4)                :func:`k_edge_augmentation`
471: (4)                Examples
472: (4)                --------
473: (4)                >>> G = nx.Graph([(1, 2), (2, 3), (4, 5)])
474: (4)                >>> G.add_nodes_from([6, 7, 8])
```

```
475: (4)                >>> sorted(unconstrained_one_edge_augmentation(G))
476: (4)                [(1, 4), (4, 6), (6, 7), (7, 8)]
477: (4)                """
478: (4)                ccs1 = list(nx.connected_components(G))
479: (4)                C = collapse(G, ccs1)
480: (4)                meta_nodes = list(C.nodes())
481: (4)                meta_aug = list(zip(meta_nodes, meta_nodes[1:]))
482: (4)                inverse = defaultdict(list)
483: (4)                for k, v in C.graph["mapping"].items():
484: (8)                    inverse[v].append(k)
485: (4)                for mu, mv in meta_aug:
486: (8)                    yield (inverse[mu][0], inverse[mv][0])
487: (0)            @nx._dispatchable
488: (0)            def weighted_one_edge_augmentation(G, avail, weight=None, partial=False):
489: (4)                """Finds the minimum weight set of edges to connect G if one exists.
490: (4)                This is a variant of the weighted MST problem.
491: (4)                Parameters
492: (4)                ----------
493: (4)                G : NetworkX graph
494: (7)                   An undirected graph.
495: (4)                avail : dict or a set of 2 or 3 tuples
496: (8)                    For more details, see :func:`k_edge_augmentation`.
497: (4)                weight : string
498: (8)                    key to use to find weights if ``avail`` is a set of 3-tuples.
499: (8)                    For more details, see :func:`k_edge_augmentation`.
500: (4)                partial : boolean
501: (8)                    If partial is True and no feasible k-edge-augmentation exists, then
the
502: (8)                    augmenting edges minimize the number of connected components.
503: (4)                Yields
504: (4)                ------
505: (4)                edge : tuple
506: (8)                    Edges in the subset of avail chosen to connect G.
507: (4)                See Also
508: (4)                --------
509: (4)                :func:`one_edge_augmentation`
510: (4)                :func:`k_edge_augmentation`
511: (4)                Examples
512: (4)                --------
513: (4)                >>> G = nx.Graph([(1, 2), (2, 3), (4, 5)])
514: (4)                >>> G.add_nodes_from([6, 7, 8])
515: (4)                >>> # any edge not in avail has an implicit weight of infinity
516: (4)                >>> avail = [(1, 3), (1, 5), (4, 7), (4, 8), (6, 1), (8, 1), (8, 2)]
517: (4)                >>> sorted(weighted_one_edge_augmentation(G, avail))
518: (4)                [(1, 5), (4, 7), (6, 1), (8, 1)]
519: (4)                >>> # find another solution by giving large weights to edges in the
520: (4)                >>> # previous solution (note some of the old edges must be used)
521: (4)                >>> avail = [(1, 3), (1, 5, 99), (4, 7, 9), (6, 1, 99), (8, 1, 99), (8,
2)]
522: (4)                >>> sorted(weighted_one_edge_augmentation(G, avail))
523: (4)                [(1, 5), (4, 7), (6, 1), (8, 2)]
524: (4)                """
525: (4)                avail_uv, avail_w = _unpack_available_edges(avail, weight=weight, G=G)
526: (4)                C = collapse(G, nx.connected_components(G))
527: (4)                mapping = C.graph["mapping"]
528: (4)                candidate_mapping = _lightest_meta_edges(mapping, avail_uv, avail_w)
529: (4)                C.add_edges_from(
530: (8)                    (mu, mv, {"weight": w, "generator": uv})
531: (8)                    for (mu, mv), uv, w in candidate_mapping
532: (4)                )
533: (4)                meta_mst = nx.minimum_spanning_tree(C)
534: (4)                if not partial and not nx.is_connected(meta_mst):
535: (8)                    raise nx.NetworkXUnfeasible("Not possible to connect G with available
edges")
536: (4)                for mu, mv, d in meta_mst.edges(data=True):
537: (8)                    if "generator" in d:
538: (12)                       edge = d["generator"]
539: (12)                       yield edge
540: (0)            @nx._dispatchable
```

```
(0)             def unconstrained_bridge_augmentation(G):
(4)                 """Finds an optimal 2-edge-augmentation of G using the fewest edges.
(4)                 This is an implementation of the algorithm detailed in [1]_.
(4)                 The basic idea is to construct a meta-graph of bridge-ccs, connect leaf
(4)                 nodes of the trees to connect the entire graph, and finally connect the
(4)                 leafs of the tree in dfs-preorder to bridge connect the entire graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    An undirected graph.
(4)                 Yields
(4)                 ------
(4)                 edge : tuple
(8)                     Edges in the bridge augmentation of G
(4)                 Notes
(4)                 -----
(4)                 Input: a graph G.
(4)                 First find the bridge components of G and collapse each bridge-cc into a
(4)                 node of a metagraph graph C, which is guaranteed to be a forest of trees.
(4)                 C contains p "leafs" --- nodes with exactly one incident edge.
(4)                 C contains q "isolated nodes" --- nodes with no incident edges.
(4)                 Theorem: If p + q > 1, then at least :math:`ceil(p / 2) + q` edges are
(8)                     needed to bridge connect C. This algorithm achieves this min number.
(4)                 The method first adds enough edges to make G into a tree and then pairs
(4)                 leafs in a simple fashion.
(4)                 Let n be the number of trees in C. Let v(i) be an isolated vertex in the
(4)                 i-th tree if one exists, otherwise it is a pair of distinct leafs nodes
(4)                 in the i-th tree. Alternating edges from these sets (i.e.  adding edges
(4)                 A1 = [(v(i)[0], v(i + 1)[1]), v(i + 1)[0], v(i + 2)[1])...]) connects C
(4)                 into a tree T. This tree has p' = p + 2q - 2(n -1) leafs and no isolated
(4)                 vertices. A1 has n - 1 edges. The next step finds ceil(p' / 2) edges to
(4)                 biconnect any tree with p' leafs.
(4)                 Convert T into an arborescence T' by picking an arbitrary root node with
(4)                 degree >= 2 and directing all edges away from the root. Note the
(4)                 implementation implicitly constructs T'.
(4)                 The leafs of T are the nodes with no existing edges in T'.
(4)                 Order the leafs of T' by DFS preorder. Then break this list in half
(4)                 and add the zipped pairs to A2.
(4)                 The set A = A1 + A2 is the minimum augmentation in the metagraph.
(4)                 To convert this to edges in the original graph
(4)                 References
(4)                 ----------
(4)                 .. [1] Eswaran, Kapali P., and R. Endre Tarjan. (1975) Augmentation
problems.
(8)                     http://epubs.siam.org/doi/abs/10.1137/0205044
(4)                 See Also
(4)                 --------
(4)                 :func:`bridge_augmentation`
(4)                 :func:`k_edge_augmentation`
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph((1, 2, 3, 4, 5, 6, 7))
(4)                 >>> sorted(unconstrained_bridge_augmentation(G))
(4)                 [(1, 7)]
(4)                 >>> G = nx.path_graph((1, 2, 3, 2, 4, 5, 6, 7))
(4)                 >>> sorted(unconstrained_bridge_augmentation(G))
(4)                 [(1, 3), (3, 7)]
(4)                 >>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
(4)                 >>> G.add_node(4)
(4)                 >>> sorted(unconstrained_bridge_augmentation(G))
(4)                 [(1, 4), (4, 0)]
(4)                 """
(4)                 bridge_ccs = list(nx.connectivity.bridge_components(G))
(4)                 C = collapse(G, bridge_ccs)
(4)                 vset1 = [
(8)                     tuple(cc) * 2  # case1: an isolated node
(8)                     if len(cc) == 1
(8)                     else sorted(cc, key=C.degree)[0:2]  # case2: pair of leaf nodes
(8)                     for cc in nx.connected_components(C)
```

```
(4)                 ]
(4)                 if len(vset1) > 1:
(8)                     nodes1 = [vs[0] for vs in vset1]
(8)                     nodes2 = [vs[1] for vs in vset1]
(8)                     A1 = list(zip(nodes1[1:], nodes2))
(4)                 else:
(8)                     A1 = []
(4)                 T = C.copy()
(4)                 T.add_edges_from(A1)
(4)                 leafs = [n for n, d in T.degree() if d == 1]
(4)                 if len(leafs) == 1:
(8)                     A2 = []
(4)                 if len(leafs) == 2:
(8)                     A2 = [tuple(leafs)]
(4)                 else:
(8)                     try:
(12)                        root = next(n for n, d in T.degree() if d > 1)
(8)                     except StopIteration:  # no nodes found with degree > 1
(12)                        return
(8)                     v2 = [n for n in nx.dfs_preorder_nodes(T, root) if T.degree(n) == 1]
(8)                     half = math.ceil(len(v2) / 2)
(8)                     A2 = list(zip(v2[:half], v2[-half:]))
(4)                 aug_tree_edges = A1 + A2
(4)                 inverse = defaultdict(list)
(4)                 for k, v in C.graph["mapping"].items():
(8)                     inverse[v].append(k)
(4)                 inverse = {
(8)                     mu: sorted(mapped, key=lambda u: (G.degree(u), u))
(8)                     for mu, mapped in inverse.items()
(4)                 }
(4)                 G2 = G.copy()
(4)                 for mu, mv in aug_tree_edges:
(8)                     for u, v in it.product(inverse[mu], inverse[mv]):
(12)                        if not G2.has_edge(u, v):
(16)                            G2.add_edge(u, v)
(16)                            yield u, v
(16)                            break
(0)             @nx._dispatchable
(0)             def weighted_bridge_augmentation(G, avail, weight=None):
(4)                 """Finds an approximate min-weight 2-edge-augmentation of G.
(4)                 This is an implementation of the approximation algorithm detailed in [1]_.
(4)                 It chooses a set of edges from avail to add to G that renders it
(4)                 2-edge-connected if such a subset exists.  This is done by finding a
(4)                 minimum spanning arborescence of a specially constructed metagraph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(7)                    An undirected graph.
(4)                 avail : set of 2 or 3 tuples.
(8)                     candidate edges (with optional weights) to choose from
(4)                 weight : string
(8)                     key to use to find weights if avail is a set of 3-tuples where the
(8)                     third item in each tuple is a dictionary.
(4)                 Yields
(4)                 ------
(4)                 edge : tuple
(8)                     Edges in the subset of avail chosen to bridge augment G.
(4)                 Notes
(4)                 -----
(4)                 Finding a weighted 2-edge-augmentation is NP-hard.
(4)                 Any edge not in ``avail`` is considered to have a weight of infinity.
(4)                 The approximation factor is 2 if ``G`` is connected and 3 if it is not.
(4)                 Runs in :math:`O(m + n log(n))` time
(4)                 References
(4)                 ----------
(4)                 .. [1] Khuller, Samir, and Ramakrishna Thurimella. (1993) Approximation
(8)                     algorithms for graph augmentation.
(8)                     http://www.sciencedirect.com/science/article/pii/S0196677483710102
(4)                 See Also
```

```
    --------
    :func:`bridge_augmentation`
    :func:`k_edge_augmentation`
    Examples
    --------
    >>> G = nx.path_graph((1, 2, 3, 4))
    >>> # When the weights are equal, (1, 4) is the best
    >>> avail = [(1, 4, 1), (1, 3, 1), (2, 4, 1)]
    >>> sorted(weighted_bridge_augmentation(G, avail))
    [(1, 4)]
    >>> # Giving (1, 4) a high weight makes the two edge solution the best.
    >>> avail = [(1, 4, 1000), (1, 3, 1), (2, 4, 1)]
    >>> sorted(weighted_bridge_augmentation(G, avail))
    [(1, 3), (2, 4)]
    >>> # ------
    >>> G = nx.path_graph((1, 2, 3, 4))
    >>> G.add_node(5)
    >>> avail = [(1, 5, 11), (2, 5, 10), (4, 3, 1), (4, 5, 1)]
    >>> sorted(weighted_bridge_augmentation(G, avail=avail))
    [(1, 5), (4, 5)]
    >>> avail = [(1, 5, 11), (2, 5, 10), (4, 3, 1), (4, 5, 51)]
    >>> sorted(weighted_bridge_augmentation(G, avail=avail))
    [(1, 5), (2, 5), (4, 5)]
    """
    if weight is None:
        weight = "weight"
    if not nx.is_connected(G):
        H = G.copy()
        connectors = list(one_edge_augmentation(H, avail=avail, weight=weight))
        H.add_edges_from(connectors)
        yield from connectors
    else:
        connectors = []
        H = G
    if len(avail) == 0:
        if nx.has_bridges(H):
            raise nx.NetworkXUnfeasible("no augmentation possible")
    avail_uv, avail_w = _unpack_available_edges(avail, weight=weight, G=H)
    bridge_ccs = nx.connectivity.bridge_components(H)
    C = collapse(H, bridge_ccs)
    mapping = C.graph["mapping"]
    meta_to_wuv = {
        (mu, mv): (w, uv)
        for (mu, mv), uv, w in _lightest_meta_edges(mapping, avail_uv, avail_w)
    }
    try:
        root = next(n for n, d in C.degree() if d == 1)
    except StopIteration:  # no nodes found with degree == 1
        return
    TR = nx.dfs_tree(C, root)
    D = nx.reverse(TR).copy()
    nx.set_edge_attributes(D, name="weight", values=0)
    lca_gen = nx.tree_all_pairs_lowest_common_ancestor(
        TR, root=root, pairs=meta_to_wuv.keys()
    )
    for (mu, mv), lca in lca_gen:
        w, uv = meta_to_wuv[(mu, mv)]
        if lca == mu:
            D.add_edge(lca, mv, weight=w, generator=uv)
        elif lca == mv:
            D.add_edge(lca, mu, weight=w, generator=uv)
        else:
            D.add_edge(lca, mu, weight=w, generator=uv)
            D.add_edge(lca, mv, weight=w, generator=uv)
    try:
        A = _minimum_rooted_branching(D, root)
    except nx.NetworkXException as err:
```

```
(8)                     raise nx.NetworkXUnfeasible("no 2-edge-augmentation possible") from
err
(4)                 bridge_connectors = set()
(4)                 for mu, mv in A.edges():
(8)                     data = D.get_edge_data(mu, mv)
(8)                     if "generator" in data:
(12)                        edge = data["generator"]
(12)                        bridge_connectors.add(edge)
(4)                 yield from bridge_connectors
(0)             def _minimum_rooted_branching(D, root):
(4)                 """Helper function to compute a minimum rooted branching (aka rooted
(4)                 arborescence)
(4)                 Before the branching can be computed, the directed graph must be rooted by
(4)                 removing the predecessors of root.
(4)                 A branching / arborescence of rooted graph G is a subgraph that contains a
(4)                 directed path from the root to every other vertex. It is the directed
(4)                 analog of the minimum spanning tree problem.
(4)                 References
(4)                 ----------
(4)                 [1] Khuller, Samir (2002) Advanced Algorithms Lecture 24 Notes.
(4)
https://web.archive.org/web/20121030033722/https://www.cs.umd.edu/class/spring2011/cmsc651/lec07.p
df
(4)                 """
(4)                 rooted = D.copy()
(4)                 rooted.remove_edges_from([(u, root) for u in D.predecessors(root)])
(4)                 A = nx.minimum_spanning_arborescence(rooted)
(4)                 return A
(0)             @nx._dispatchable(returns_graph=True)
(0)             def collapse(G, grouped_nodes):
(4)                 """Collapses each group of nodes into a single node.
(4)                 This is similar to condensation, but works on undirected graphs.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 grouped_nodes:  list or generator
(7)                    Grouping of nodes to collapse. The grouping must be disjoint.
(7)                    If grouped_nodes are strongly_connected_components then this is
(7)                    equivalent to :func:`condensation`.
(4)                 Returns
(4)                 -------
(4)                 C : NetworkX Graph
(7)                    The collapsed graph C of G with respect to the node grouping.  The node
(7)                    labels are integers corresponding to the index of the component in the
(7)                    list of grouped_nodes.  C has a graph attribute named 'mapping' with a
(7)                    dictionary mapping the original nodes to the nodes in C to which they
(7)                    belong.  Each node in C also has a node attribute 'members' with the
set
(7)                    of original nodes in G that form the group that the node in C
(7)                    represents.
(4)                 Examples
(4)                 --------
(4)                 >>> # Collapses a graph using disjoint groups, but not necessarily
connected
(4)                 >>> G = nx.Graph([(1, 0), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (5, 7)])
(4)                 >>> G.add_node("A")
(4)                 >>> grouped_nodes = [{0, 1, 2, 3}, {5, 6, 7}]
(4)                 >>> C = collapse(G, grouped_nodes)
(4)                 >>> members = nx.get_node_attributes(C, "members")
(4)                 >>> sorted(members.keys())
(4)                 [0, 1, 2, 3]
(4)                 >>> member_values = set(map(frozenset, members.values()))
(4)                 >>> assert {0, 1, 2, 3} in member_values
(4)                 >>> assert {4} in member_values
(4)                 >>> assert {5, 6, 7} in member_values
(4)                 >>> assert {"A"} in member_values
(4)                 """
(4)                 mapping = {}
(4)                 members = {}
```

```
(4)                 C = G.__class__()
(4)                 i = 0  # required if G is empty
(4)                 remaining = set(G.nodes())
(4)                 for i, group in enumerate(grouped_nodes):
(8)                     group = set(group)
(8)                     assert remaining.issuperset(
(12)                        group
(8)                     ), "grouped nodes must exist in G and be disjoint"
(8)                     remaining.difference_update(group)
(8)                     members[i] = group
(8)                     mapping.update((n, i) for n in group)
(4)                 for i, node in enumerate(remaining, start=i + 1):
(8)                     group = {node}
(8)                     members[i] = group
(8)                     mapping.update((n, i) for n in group)
(4)                 number_of_groups = i + 1
(4)                 C.add_nodes_from(range(number_of_groups))
(4)                 C.add_edges_from(
(8)                     (mapping[u], mapping[v]) for u, v in G.edges() if mapping[u] !=
mapping[v]
(4)                 )
(4)                 nx.set_node_attributes(C, name="members", values=members)
(4)                 C.graph["mapping"] = mapping
(4)                 return C
(0)             @nx._dispatchable
(0)             def complement_edges(G):
(4)                 """Returns only the edges in the complement of G
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 Yields
(4)                 ------
(4)                 edge : tuple
(8)                     Edges in the complement of G
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph((1, 2, 3, 4))
(4)                 >>> sorted(complement_edges(G))
(4)                 [(1, 3), (1, 4), (2, 4)]
(4)                 >>> G = nx.path_graph((1, 2, 3, 4), nx.DiGraph())
(4)                 >>> sorted(complement_edges(G))
(4)                 [(1, 3), (1, 4), (2, 1), (2, 4), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3)]
(4)                 >>> G = nx.complete_graph(1000)
(4)                 >>> sorted(complement_edges(G))
(4)                 []
(4)                 """
(4)                 G_adj = G._adj  # Store as a variable to eliminate attribute lookup
(4)                 if G.is_directed():
(8)                     for u, v in it.combinations(G.nodes(), 2):
(12)                        if v not in G_adj[u]:
(16)                            yield (u, v)
(12)                        if u not in G_adj[v]:
(16)                            yield (v, u)
(4)                 else:
(8)                     for u, v in it.combinations(G.nodes(), 2):
(12)                        if v not in G_adj[u]:
(16)                            yield (u, v)
(0)             def _compat_shuffle(rng, input):
(4)                 """wrapper around rng.shuffle for python 2 compatibility reasons"""
(4)                 rng.shuffle(input)
(0)             @not_implemented_for("multigraph")
(0)             @not_implemented_for("directed")
(0)             @py_random_state(4)
(0)             @nx._dispatchable
(0)             def greedy_k_edge_augmentation(G, k, avail=None, weight=None, seed=None):
(4)                 """Greedy algorithm for finding a k-edge-augmentation
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
```

```
(7)                 An undirected graph.
(4)             k : integer
(8)                 Desired edge connectivity
(4)             avail : dict or a set of 2 or 3 tuples
(8)                 For more details, see :func:`k_edge_augmentation`.
(4)             weight : string
(8)                 key to use to find weights if ``avail`` is a set of 3-tuples.
(8)                 For more details, see :func:`k_edge_augmentation`.
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Yields
(4)             ------
(4)             edge : tuple
(8)                 Edges in the greedy augmentation of G
(4)             Notes
(4)             -----
(4)             The algorithm is simple. Edges are incrementally added between parts of
the
(4)             graph that are not yet locally k-edge-connected. Then edges are from the
(4)             augmenting set are pruned as long as local-edge-connectivity is not
broken.
(4)             This algorithm is greedy and does not provide optimality guarantees. It
(4)             exists only to provide :func:`k_edge_augmentation` with the ability to
(4)             generate a feasible solution for arbitrary k.
(4)             See Also
(4)             --------
(4)             :func:`k_edge_augmentation`
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph((1, 2, 3, 4, 5, 6, 7))
(4)             >>> sorted(greedy_k_edge_augmentation(G, k=2))
(4)             [(1, 7)]
(4)             >>> sorted(greedy_k_edge_augmentation(G, k=1, avail=[]))
(4)             []
(4)             >>> G = nx.path_graph((1, 2, 3, 4, 5, 6, 7))
(4)             >>> avail = {(u, v): 1 for (u, v) in complement_edges(G)}
(4)             >>> # randomized pruning process can produce different solutions
(4)             >>> sorted(greedy_k_edge_augmentation(G, k=4, avail=avail, seed=2))
(4)             [(1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (2, 4), (2, 6), (3, 7), (5, 7)]
(4)             >>> sorted(greedy_k_edge_augmentation(G, k=4, avail=avail, seed=3))
(4)             [(1, 3), (1, 5), (1, 6), (2, 4), (2, 6), (3, 7), (4, 7), (5, 7)]
(4)             """
(4)             aug_edges = []
(4)             done = is_k_edge_connected(G, k)
(4)             if done:
(8)                 return
(4)             if avail is None:
(8)                 avail_uv = list(complement_edges(G))
(8)                 avail_w = [1] * len(avail_uv)
(4)             else:
(8)                 avail_uv, avail_w = _unpack_available_edges(avail, weight=weight, G=G)
(4)             tiebreaker = [sum(map(G.degree, uv)) for uv in avail_uv]
(4)             avail_wduv = sorted(zip(avail_w, tiebreaker, avail_uv))
(4)             avail_uv = [uv for w, d, uv in avail_wduv]
(4)             H = G.copy()
(4)             for u, v in avail_uv:
(8)                 done = False
(8)                 if not is_locally_k_edge_connected(H, u, v, k=k):
(12)                    aug_edges.append((u, v))
(12)                    H.add_edge(u, v)
(12)                    if H.degree(u) >= k and H.degree(v) >= k:
(16)                        done = is_k_edge_connected(H, k)
(8)                 if done:
(12)                    break
(4)             if not done:
(8)                 raise nx.NetworkXUnfeasible("not able to k-edge-connect with available
edges")
(4)             _compat_shuffle(seed, aug_edges)
```

```
943: (4)                for u, v in list(aug_edges):
944: (8)                    if H.degree(u) <= k or H.degree(v) <= k:
945: (12)                       continue
946: (8)                    H.remove_edge(u, v)
947: (8)                    aug_edges.remove((u, v))
948: (8)                    if not is_k_edge_connected(H, k=k):
949: (12)                       H.add_edge(u, v)
950: (12)                       aug_edges.append((u, v))
951: (4)                yield from aug_edges
```

File 222 - edge_kcomponents.py:

```
1: (0)              """
2: (0)              Algorithms for finding k-edge-connected components and subgraphs.
3: (0)              A k-edge-connected component (k-edge-cc) is a maximal set of nodes in G, such
4: (0)              that all pairs of node have an edge-connectivity of at least k.
5: (0)              A k-edge-connected subgraph (k-edge-subgraph) is a maximal set of nodes in G,
6: (0)              such that the subgraph of G defined by the nodes has an edge-connectivity at
7: (0)              least k.
8: (0)              """
9: (0)              import itertools as it
10: (0)             from functools import partial
11: (0)             import networkx as nx
12: (0)             from networkx.utils import arbitrary_element, not_implemented_for
13: (0)             __all__ = [
14: (4)                 "k_edge_components",
15: (4)                 "k_edge_subgraphs",
16: (4)                 "bridge_components",
17: (4)                 "EdgeComponentAuxGraph",
18: (0)             ]
19: (0)             @not_implemented_for("multigraph")
20: (0)             @nx._dispatchable
21: (0)             def k_edge_components(G, k):
22: (4)                 """Generates nodes in each maximal k-edge-connected component in G.
23: (4)                 Parameters
24: (4)                 ----------
25: (4)                 G : NetworkX graph
26: (4)                 k : Integer
27: (8)                     Desired edge connectivity
28: (4)                 Returns
29: (4)                 -------
30: (4)                 k_edge_components : a generator of k-edge-ccs. Each set of returned nodes
31: (7)                    will have k-edge-connectivity in the graph G.
32: (4)                 See Also
33: (4)                 --------
34: (4)                 :func:`local_edge_connectivity`
35: (4)                 :func:`k_edge_subgraphs` : similar to this function, but the subgraph
36: (8)                     defined by the nodes must also have k-edge-connectivity.
37: (4)                 :func:`k_components` : similar to this function, but uses node-
connectivity
38: (8)                     instead of edge-connectivity
39: (4)                 Raises
40: (4)                 ------
41: (4)                 NetworkXNotImplemented
42: (8)                     If the input graph is a multigraph.
43: (4)                 ValueError:
44: (8)                     If k is less than 1
45: (4)                 Notes
46: (4)                 -----
47: (4)                 Attempts to use the most efficient implementation available based on k.
48: (4)                 If k=1, this is simply connected components for directed graphs and
49: (4)                 connected components for undirected graphs.
50: (4)                 If k=2 on an efficient bridge connected component algorithm from _[1] is
51: (4)                 run based on the chain decomposition.
52: (4)                 Otherwise, the algorithm from _[2] is used.
53: (4)                 Examples
54: (4)                 --------
```

```
55: (4)                 >>> import itertools as it
56: (4)                 >>> from networkx.utils import pairwise
57: (4)                 >>> paths = [
58: (4)                 ...     (1, 2, 4, 3, 1, 4),
59: (4)                 ...     (5, 6, 7, 8, 5, 7, 8, 6),
60: (4)                 ... ]
61: (4)                 >>> G = nx.Graph()
62: (4)                 >>> G.add_nodes_from(it.chain(*paths))
63: (4)                 >>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
64: (4)                 >>> # note this returns {1, 4} unlike k_edge_subgraphs
65: (4)                 >>> sorted(map(sorted, nx.k_edge_components(G, k=3)))
66: (4)                 [[1, 4], [2], [3], [5, 6, 7, 8]]
67: (4)                 References
68: (4)                 ----------
69: (4)                 .. [1] https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29
70: (4)                 .. [2] Wang, Tianhao, et al. (2015) A simple algorithm for finding all
71: (8)                     k-edge-connected components.
72: (8)                     http://journals.plos.org/plosone/article?
id=10.1371/journal.pone.0136264
73: (4)                 """
74: (4)                 if k < 1:
75: (8)                     raise ValueError("k cannot be less than 1")
76: (4)                 if G.is_directed():
77: (8)                     if k == 1:
78: (12)                        return nx.strongly_connected_components(G)
79: (8)                     else:
80: (12)                        aux_graph = EdgeComponentAuxGraph.construct(G)
81: (12)                        return aux_graph.k_edge_components(k)
82: (4)                 else:
83: (8)                     if k == 1:
84: (12)                        return nx.connected_components(G)
85: (8)                     elif k == 2:
86: (12)                        return bridge_components(G)
87: (8)                     else:
88: (12)                        aux_graph = EdgeComponentAuxGraph.construct(G)
89: (12)                        return aux_graph.k_edge_components(k)
90: (0)             @not_implemented_for("multigraph")
91: (0)             @nx._dispatchable
92: (0)             def k_edge_subgraphs(G, k):
93: (4)                 """Generates nodes in each maximal k-edge-connected subgraph in G.
94: (4)                 Parameters
95: (4)                 ----------
96: (4)                 G : NetworkX graph
97: (4)                 k : Integer
98: (8)                     Desired edge connectivity
99: (4)                 Returns
100: (4)                -------
101: (4)                k_edge_subgraphs : a generator of k-edge-subgraphs
102: (8)                    Each k-edge-subgraph is a maximal set of nodes that defines a subgraph
103: (8)                    of G that is k-edge-connected.
104: (4)                See Also
105: (4)                --------
106: (4)                :func:`edge_connectivity`
107: (4)                :func:`k_edge_components` : similar to this function, but nodes only
108: (8)                    need to have k-edge-connectivity within the graph G and the subgraphs
109: (8)                    might not be k-edge-connected.
110: (4)                Raises
111: (4)                ------
112: (4)                NetworkXNotImplemented
113: (8)                    If the input graph is a multigraph.
114: (4)                ValueError:
115: (8)                    If k is less than 1
116: (4)                Notes
117: (4)                -----
118: (4)                Attempts to use the most efficient implementation available based on k.
119: (4)                If k=1, or k=2 and the graph is undirected, then this simply calls
120: (4)                `k_edge_components`.  Otherwise the algorithm from _[1] is used.
121: (4)                Examples
122: (4)                --------
```

```
(4)                 >>> import itertools as it
(4)                 >>> from networkx.utils import pairwise
(4)                 >>> paths = [
(4)                 ...     (1, 2, 4, 3, 1, 4),
(4)                 ...     (5, 6, 7, 8, 5, 7, 8, 6),
(4)                 ... ]
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_nodes_from(it.chain(*paths))
(4)                 >>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
(4)                 >>> # note this does not return {1, 4} unlike k_edge_components
(4)                 >>> sorted(map(sorted, nx.k_edge_subgraphs(G, k=3)))
(4)                 [[1], [2], [3], [4], [5, 6, 7, 8]]
(4)                 References
(4)                 ----------
(4)                 .. [1] Zhou, Liu, et al. (2012) Finding maximal k-edge-connected subgraphs
(8)                     from a large graph.  ACM International Conference on Extending
Database
(8)                     Technology 2012 480-–491.
(8)                     https://openproceedings.org/2012/conf/edbt/ZhouLYLCL12.pdf
(4)                 """
(4)                 if k < 1:
(8)                     raise ValueError("k cannot be less than 1")
(4)                 if G.is_directed():
(8)                     if k <= 1:
(12)                        return k_edge_components(G, k)
(8)                     else:
(12)                        return _k_edge_subgraphs_nodes(G, k)
(4)                 else:
(8)                     if k <= 2:
(12)                        return k_edge_components(G, k)
(8)                     else:
(12)                        return _k_edge_subgraphs_nodes(G, k)
(0)             def _k_edge_subgraphs_nodes(G, k):
(4)                 """Helper to get the nodes from the subgraphs.
(4)                 This allows k_edge_subgraphs to return a generator.
(4)                 """
(4)                 for C in general_k_edge_subgraphs(G, k):
(8)                     yield set(C.nodes())
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable
(0)             def bridge_components(G):
(4)                 """Finds all bridge-connected components G.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX undirected graph
(4)                 Returns
(4)                 -------
(4)                 bridge_components : a generator of 2-edge-connected components
(4)                 See Also
(4)                 --------
(4)                 :func:`k_edge_subgraphs` : this function is a special case for an
(8)                     undirected graph where k=2.
(4)                 :func:`biconnected_components` : similar to this function, but is defined
(8)                     using 2-node-connectivity instead of 2-edge-connectivity.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If the input graph is directed or a multigraph.
(4)                 Notes
(4)                 -----
(4)                 Bridge-connected components are also known as 2-edge-connected components.
(4)                 Examples
(4)                 --------
(4)                 >>> # The barbell graph with parameter zero has a single bridge
(4)                 >>> G = nx.barbell_graph(5, 0)
(4)                 >>> from networkx.algorithms.connectivity.edge_kcomponents import
bridge_components
(4)                 >>> sorted(map(sorted, bridge_components(G)))
```

```
190: (4)                [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
191: (4)                """
192: (4)                H = G.copy()
193: (4)                H.remove_edges_from(nx.bridges(G))
194: (4)                yield from nx.connected_components(H)
195: (0)            class EdgeComponentAuxGraph:
196: (4)                r"""A simple algorithm to find all k-edge-connected components in a graph.
197: (4)                Constructing the auxiliary graph (which may take some time) allows for the
198: (4)                k-edge-ccs to be found in linear time for arbitrary k.
199: (4)                Notes
200: (4)                -----
201: (4)                This implementation is based on [1]_. The idea is to construct an
auxiliary
202: (4)                graph from which the k-edge-ccs can be extracted in linear time. The
203: (4)                auxiliary graph is constructed in $O(|V|\cdot F)$ operations, where F is
the
204: (4)                complexity of max flow. Querying the components takes an additional
$O(|V|)$
205: (4)                operations. This algorithm can be slow for large graphs, but it handles an
206: (4)                arbitrary k and works for both directed and undirected inputs.
207: (4)                The undirected case for k=1 is exactly connected components.
208: (4)                The undirected case for k=2 is exactly bridge connected components.
209: (4)                The directed case for k=1 is exactly strongly connected components.
210: (4)                References
211: (4)                ----------
212: (4)                .. [1] Wang, Tianhao, et al. (2015) A simple algorithm for finding all
213: (8)                    k-edge-connected components.
214: (8)                    http://journals.plos.org/plosone/article?
id=10.1371/journal.pone.0136264
215: (4)                Examples
216: (4)                --------
217: (4)                >>> import itertools as it
218: (4)                >>> from networkx.utils import pairwise
219: (4)                >>> from networkx.algorithms.connectivity import EdgeComponentAuxGraph
220: (4)                >>> # Build an interesting graph with multiple levels of k-edge-ccs
221: (4)                >>> paths = [
222: (4)                ...     (1, 2, 3, 4, 1, 3, 4, 2),  # a 3-edge-cc (a 4 clique)
223: (4)                ...     (5, 6, 7, 5),  # a 2-edge-cc (a 3 clique)
224: (4)                ...     (1, 5),  # combine first two ccs into a 1-edge-cc
225: (4)                ...     (0,),  # add an additional disconnected 1-edge-cc
226: (4)                ... ]
227: (4)                >>> G = nx.Graph()
228: (4)                >>> G.add_nodes_from(it.chain(*paths))
229: (4)                >>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
230: (4)                >>> # Constructing the AuxGraph takes about O(n ** 4)
231: (4)                >>> aux_graph = EdgeComponentAuxGraph.construct(G)
232: (4)                >>> # Once constructed, querying takes O(n)
233: (4)                >>> sorted(map(sorted, aux_graph.k_edge_components(k=1)))
234: (4)                [[0], [1, 2, 3, 4, 5, 6, 7]]
235: (4)                >>> sorted(map(sorted, aux_graph.k_edge_components(k=2)))
236: (4)                [[0], [1, 2, 3, 4], [5, 6, 7]]
237: (4)                >>> sorted(map(sorted, aux_graph.k_edge_components(k=3)))
238: (4)                [[0], [1, 2, 3, 4], [5], [6], [7]]
239: (4)                >>> sorted(map(sorted, aux_graph.k_edge_components(k=4)))
240: (4)                [[0], [1], [2], [3], [4], [5], [6], [7]]
241: (4)                The auxiliary graph is primarily used for k-edge-ccs but it
242: (4)                can also speed up the queries of k-edge-subgraphs by refining the
243: (4)                search space.
244: (4)                >>> import itertools as it
245: (4)                >>> from networkx.utils import pairwise
246: (4)                >>> from networkx.algorithms.connectivity import EdgeComponentAuxGraph
247: (4)                >>> paths = [
248: (4)                ...     (1, 2, 4, 3, 1, 4),
249: (4)                ... ]
250: (4)                >>> G = nx.Graph()
251: (4)                >>> G.add_nodes_from(it.chain(*paths))
252: (4)                >>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
253: (4)                >>> aux_graph = EdgeComponentAuxGraph.construct(G)
254: (4)                >>> sorted(map(sorted, aux_graph.k_edge_subgraphs(k=3)))
```

```
    [[1], [2], [3], [4]]
    >>> sorted(map(sorted, aux_graph.k_edge_components(k=3)))
    [[1, 4], [2], [3]]
    """
    @classmethod
    def construct(EdgeComponentAuxGraph, G):
        """Builds an auxiliary graph encoding edge-connectivity between nodes.
        Notes
        -----
        Given G=(V, E), initialize an empty auxiliary graph A.
        Choose an arbitrary source node s.  Initialize a set N of available
        nodes (that can be used as the sink). The algorithm picks an
        arbitrary node t from N - {s}, and then computes the minimum st-cut
        (S, T) with value w. If G is directed the minimum of the st-cut or
        the ts-cut is used instead. Then, the edge (s, t) is added to the
        auxiliary graph with weight w. The algorithm is called recursively
        first using S as the available nodes and s as the source, and then
        using T and t. Recursion stops when the source is the only available
        node.
        Parameters
        ----------
        G : NetworkX graph
        """
        not_implemented_for("multigraph")(lambda G: G)(G)
        def _recursive_build(H, A, source, avail):
            if {source} == avail:
                return
            sink = arbitrary_element(avail - {source})
            value, (S, T) = nx.minimum_cut(H, source, sink)
            if H.is_directed():
                value_, (T_, S_) = nx.minimum_cut(H, sink, source)
                if value_ < value:
                    value, S, T = value_, S_, T_
            A.add_edge(source, sink, weight=value)
            _recursive_build(H, A, source, avail.intersection(S))
            _recursive_build(H, A, sink, avail.intersection(T))
        H = G.__class__()
        H.add_nodes_from(G.nodes())
        H.add_edges_from(G.edges(), capacity=1)
        A = nx.Graph()
        if H.number_of_nodes() > 0:
            source = arbitrary_element(H.nodes())
            avail = set(H.nodes())
            _recursive_build(H, A, source, avail)
        self = EdgeComponentAuxGraph()
        self.A = A
        self.H = H
        return self
    def k_edge_components(self, k):
        """Queries the auxiliary graph for k-edge-connected components.
        Parameters
        ----------
        k : Integer
            Desired edge connectivity
        Returns
        -------
        k_edge_components : a generator of k-edge-ccs
        Notes
        -----
        Given the auxiliary graph, the k-edge-connected components can be
        determined in linear time by removing all edges with weights less than
        k from the auxiliary graph.  The resulting connected components are the
        k-edge-ccs in the original graph.
        """
        if k < 1:
            raise ValueError("k cannot be less than 1")
        A = self.A
        aux_weights = nx.get_edge_attributes(A, "weight")
```

```
(8)                 R = nx.Graph()
(8)                 R.add_nodes_from(A.nodes())
(8)                 R.add_edges_from(e for e, w in aux_weights.items() if w >= k)
(8)                 yield from nx.connected_components(R)
(4)             def k_edge_subgraphs(self, k):
(8)                 """Queries the auxiliary graph for k-edge-connected subgraphs.
(8)                 Parameters
(8)                 ----------
(8)                 k : Integer
(12)                    Desired edge connectivity
(8)                 Returns
(8)                 -------
(8)                 k_edge_subgraphs : a generator of k-edge-subgraphs
(8)                 Notes
(8)                 -----
(8)                 Refines the k-edge-ccs into k-edge-subgraphs. The running time is more
(8)                 than $O(|V|)$.
(8)                 For single values of k it is faster to use `nx.k_edge_subgraphs`.
(8)                 But for multiple values of k, it can be faster to build AuxGraph and
(8)                 then use this method.
(8)                 """
(8)                 if k < 1:
(12)                    raise ValueError("k cannot be less than 1")
(8)                 H = self.H
(8)                 A = self.A
(8)                 aux_weights = nx.get_edge_attributes(A, "weight")
(8)                 R = nx.Graph()
(8)                 R.add_nodes_from(A.nodes())
(8)                 R.add_edges_from(e for e, w in aux_weights.items() if w >= k)
(8)                 for cc in nx.connected_components(R):
(12)                    if len(cc) < k:
(16)                        for node in cc:
(20)                            yield {node}
(12)                    else:
(16)                        C = H.subgraph(cc)
(16)                        yield from k_edge_subgraphs(C, k)
(0)         def _low_degree_nodes(G, k, nbunch=None):
(4)             """Helper for finding nodes with degree less than k."""
(4)             if G.is_directed():
(8)                 seen = set()
(8)                 for node, degree in G.out_degree(nbunch):
(12)                    if degree < k:
(16)                        seen.add(node)
(16)                        yield node
(8)                 for node, degree in G.in_degree(nbunch):
(12)                    if node not in seen and degree < k:
(16)                        seen.add(node)
(16)                        yield node
(4)             else:
(8)                 for node, degree in G.degree(nbunch):
(12)                    if degree < k:
(16)                        yield node
(0)         def _high_degree_components(G, k):
(4)             """Helper for filtering components that can't be k-edge-connected.
(4)             Removes and generates each node with degree less than k.  Then generates
(4)             remaining components where all nodes have degree at least k.
(4)             """
(4)             H = G.copy()
(4)             singletons = set(_low_degree_nodes(H, k))
(4)             while singletons:
(8)                 nbunch = set(it.chain.from_iterable(map(H.neighbors, singletons)))
(8)                 nbunch.difference_update(singletons)
(8)                 H.remove_nodes_from(singletons)
(8)                 for node in singletons:
(12)                    yield {node}
(8)                 singletons = set(_low_degree_nodes(H, k, nbunch))
(4)             if G.is_directed():
(8)                 yield from nx.strongly_connected_components(H)
(4)             else:
```

```
(8)                 yield from nx.connected_components(H)
(0)         @nx._dispatchable(returns_graph=True)
(0)         def general_k_edge_subgraphs(G, k):
(4)             """General algorithm to find all maximal k-edge-connected subgraphs in
`G`.
(4)             Parameters
(4)             ----------
(4)             G : nx.Graph
(7)                Graph in which all maximal k-edge-connected subgraphs will be found.
(4)             k : int
(4)             Yields
(4)             ------
(4)             k_edge_subgraphs : Graph instances that are k-edge-subgraphs
(8)                 Each k-edge-subgraph contains a maximal set of nodes that defines a
(8)                 subgraph of `G` that is k-edge-connected.
(4)             Notes
(4)             -----
(4)             Implementation of the basic algorithm from [1]_.  The basic idea is to
find
(4)             a global minimum cut of the graph. If the cut value is at least k, then
the
(4)             graph is a k-edge-connected subgraph and can be added to the results.
(4)             Otherwise, the cut is used to split the graph in two and the procedure is
(4)             applied recursively. If the graph is just a single node, then it is also
(4)             added to the results. At the end, each result is either guaranteed to be
(4)             a single node or a subgraph of G that is k-edge-connected.
(4)             This implementation contains optimizations for reducing the number of
calls
(4)             to max-flow, but there are other optimizations in [1]_ that could be
(4)             implemented.
(4)             References
(4)             ----------
(4)             .. [1] Zhou, Liu, et al. (2012) Finding maximal k-edge-connected subgraphs
(8)                 from a large graph.  ACM International Conference on Extending
Database
(8)                 Technology 2012 480-–491.
(8)                 https://openproceedings.org/2012/conf/edbt/ZhouLYLCL12.pdf
(4)             Examples
(4)             --------
(4)             >>> from networkx.utils import pairwise
(4)             >>> paths = [
(4)             ...     (11, 12, 13, 14, 11, 13, 14, 12),  # a 4-clique
(4)             ...     (21, 22, 23, 24, 21, 23, 24, 22),  # another 4-clique
(4)             ...     # connect the cliques with high degree but low connectivity
(4)             ...     (50, 13),
(4)             ...     (12, 50, 22),
(4)             ...     (13, 102, 23),
(4)             ...     (14, 101, 24),
(4)             ... ]
(4)             >>> G = nx.Graph(it.chain(*[pairwise(path) for path in paths]))
(4)             >>> sorted(len(k_sg) for k_sg in k_edge_subgraphs(G, k=3))
(4)             [1, 1, 1, 4, 4]
(4)             """
(4)             if k < 1:
(8)                 raise ValueError("k cannot be less than 1")
(4)             find_ccs = partial(_high_degree_components, k=k)
(4)             if G.number_of_nodes() < k:
(8)                 for node in G.nodes():
(12)                    yield G.subgraph([node]).copy()
(8)                 return
(4)             R0 = {G.subgraph(cc).copy() for cc in find_ccs(G)}
(4)             while R0:
(8)                 G1 = R0.pop()
(8)                 if G1.number_of_nodes() == 1:
(12)                    yield G1
(8)                 else:
(12)                    cut_edges = nx.minimum_edge_cut(G1)
(12)                    cut_value = len(cut_edges)
(12)                    if cut_value < k:
```

```
456: (16)                           G1.remove_edges_from(cut_edges)
457: (16)                           for cc in find_ccs(G1):
458: (20)                               R0.add(G1.subgraph(cc).copy())
459: (12)                       else:
460: (16)                           yield G1
```

File 223 - kcomponents.py:

```
1: (0)              """
2: (0)              Moody and White algorithm for k-components
3: (0)              """
4: (0)              from collections import defaultdict
5: (0)              from itertools import combinations
6: (0)              from operator import itemgetter
7: (0)              import networkx as nx
8: (0)              from networkx.algorithms.flow import edmonds_karp
9: (0)              from networkx.utils import not_implemented_for
10: (0)             default_flow_func = edmonds_karp
11: (0)             __all__ = ["k_components"]
12: (0)             @not_implemented_for("directed")
13: (0)             @nx._dispatchable
14: (0)             def k_components(G, flow_func=None):
15: (4)                 r"""Returns the k-component structure of a graph G.
16: (4)                 A `k`-component is a maximal subgraph of a graph G that has, at least,
17: (4)                 node connectivity `k`: we need to remove at least `k` nodes to break it
18: (4)                 into more components. `k`-components have an inherent hierarchical
19: (4)                 structure because they are nested in terms of connectivity: a connected
20: (4)                 graph can contain several 2-components, each of which can contain
21: (4)                 one or more 3-components, and so forth.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 G : NetworkX graph
25: (4)                 flow_func : function
26: (8)                     Function to perform the underlying flow computations. Default value
27: (8)                     :meth:`edmonds_karp`. This function performs better in sparse graphs
with
28: (8)                     right tailed degree distributions. :meth:`shortest_augmenting_path`
will
29: (8)                     perform better in denser graphs.
30: (4)                 Returns
31: (4)                 -------
32: (4)                 k_components : dict
33: (8)                     Dictionary with all connectivity levels `k` in the input Graph as keys
34: (8)                     and a list of sets of nodes that form a k-component of level `k` as
35: (8)                     values.
36: (4)                 Raises
37: (4)                 ------
38: (4)                 NetworkXNotImplemented
39: (8)                     If the input graph is directed.
40: (4)                 Examples
41: (4)                 --------
42: (4)                 >>> # Petersen graph has 10 nodes and it is triconnected, thus all
43: (4)                 >>> # nodes are in a single component on all three connectivity levels
44: (4)                 >>> G = nx.petersen_graph()
45: (4)                 >>> k_components = nx.k_components(G)
46: (4)                 Notes
47: (4)                 -----
48: (4)                 Moody and White [1]_ (appendix A) provide an algorithm for identifying
49: (4)                 k-components in a graph, which is based on Kanevsky's algorithm [2]_
50: (4)                 for finding all minimum-size node cut-sets of a graph (implemented in
51: (4)                 :meth:`all_node_cuts` function):
52: (8)                     1. Compute node connectivity, k, of the input graph G.
53: (8)                     2. Identify all k-cutsets at the current level of connectivity using
54: (11)                       Kanevsky's algorithm.
55: (8)                     3. Generate new graph components based on the removal of
56: (11)                       these cutsets. Nodes in a cutset belong to both sides
57: (11)                       of the induced cut.
```

```
58: (8)                     4. If the graph is neither complete nor trivial, return to 1;
59: (11)                       else end.
60: (4)                 This implementation also uses some heuristics (see [3]_ for details)
61: (4)                 to speed up the computation.
62: (4)                 See also
63: (4)                 --------
64: (4)                 node_connectivity
65: (4)                 all_node_cuts
66: (4)                 biconnected_components : special case of this function when k=2
67: (4)                 k_edge_components : similar to this function, but uses edge-connectivity
68: (8)                     instead of node-connectivity
69: (4)                 References
70: (4)                 ----------
71: (4)                 .. [1]  Moody, J. and D. White (2003). Social cohesion and embeddedness:
72: (12)                        A hierarchical conception of social groups.
73: (12)                        American Sociological Review 68(1), 103--28.
74: (12)                        http://www2.asanet.org/journals/ASRFeb03MoodyWhite.pdf
75: (4)                 .. [2]  Kanevsky, A. (1993). Finding all minimum-size separating vertex
76: (12)                        sets in a graph. Networks 23(6), 533--541.
77: (12)                        http://onlinelibrary.wiley.com/doi/10.1002/net.3230230604/abstract
78: (4)                 .. [3]  Torrents, J. and F. Ferraro (2015). Structural Cohesion:
79: (12)                        Visualization and Heuristics for Fast Computation.
80: (12)                        https://arxiv.org/pdf/1503.04476v1
81: (4)                 """
82: (4)                 k_components = defaultdict(list)
83: (4)                 if flow_func is None:
84: (8)                     flow_func = default_flow_func
85: (4)                 for component in nx.connected_components(G):
86: (8)                     comp = set(component)
87: (8)                     if len(comp) > 1:
88: (12)                        k_components[1].append(comp)
89: (4)                 bicomponents = [G.subgraph(c) for c in nx.biconnected_components(G)]
90: (4)                 for bicomponent in bicomponents:
91: (8)                     bicomp = set(bicomponent)
92: (8)                     if len(bicomp) > 2:
93: (12)                        k_components[2].append(bicomp)
94: (4)                 for B in bicomponents:
95: (8)                     if len(B) <= 2:
96: (12)                        continue
97: (8)                     k = nx.node_connectivity(B, flow_func=flow_func)
98: (8)                     if k > 2:
99: (12)                        k_components[k].append(set(B))
100: (8)                    cuts = list(nx.all_node_cuts(B, k=k, flow_func=flow_func))
101: (8)                    stack = [(k, _generate_partition(B, cuts, k))]
102: (8)                    while stack:
103: (12)                       (parent_k, partition) = stack[-1]
104: (12)                       try:
105: (16)                           nodes = next(partition)
106: (16)                           C = B.subgraph(nodes)
107: (16)                           this_k = nx.node_connectivity(C, flow_func=flow_func)
108: (16)                           if this_k > parent_k and this_k > 2:
109: (20)                               k_components[this_k].append(set(C))
110: (16)                           cuts = list(nx.all_node_cuts(C, k=this_k,
flow_func=flow_func))
111: (16)                           if cuts:
112: (20)                               stack.append((this_k, _generate_partition(C, cuts,
this_k)))
113: (12)                       except StopIteration:
114: (16)                           stack.pop()
115: (4)                return _reconstruct_k_components(k_components)
116: (0)            def _consolidate(sets, k):
117: (4)                """Merge sets that share k or more elements.
118: (4)                See: http://rosettacode.org/wiki/Set_consolidation
119: (4)                The iterative python implementation posted there is
120: (4)                faster than this because of the overhead of building a
121: (4)                Graph and calling nx.connected_components, but it's not
122: (4)                clear for us if we can use it in NetworkX because there
123: (4)                is no licence for the code.
124: (4)                """
```

```
125: (4)                G = nx.Graph()
126: (4)                nodes = dict(enumerate(sets))
127: (4)                G.add_nodes_from(nodes)
128: (4)                G.add_edges_from(
129: (8)                    (u, v) for u, v in combinations(nodes, 2) if len(nodes[u] & nodes[v])
>= k
130: (4)                )
131: (4)                for component in nx.connected_components(G):
132: (8)                    yield set.union(*[nodes[n] for n in component])
133: (0)            def _generate_partition(G, cuts, k):
134: (4)                def has_nbrs_in_partition(G, node, partition):
135: (8)                    return any(n in partition for n in G[node])
136: (4)                components = []
137: (4)                nodes = {n for n, d in G.degree() if d > k} - {n for cut in cuts for n in
cut}
138: (4)                H = G.subgraph(nodes)
139: (4)                for cc in nx.connected_components(H):
140: (8)                    component = set(cc)
141: (8)                    for cut in cuts:
142: (12)                       for node in cut:
143: (16)                           if has_nbrs_in_partition(G, node, cc):
144: (20)                               component.add(node)
145: (8)                    if len(component) < G.order():
146: (12)                       components.append(component)
147: (4)                yield from _consolidate(components, k + 1)
148: (0)            def _reconstruct_k_components(k_comps):
149: (4)                result = {}
150: (4)                max_k = max(k_comps)
151: (4)                for k in reversed(range(1, max_k + 1)):
152: (8)                    if k == max_k:
153: (12)                       result[k] = list(_consolidate(k_comps[k], k))
154: (8)                    elif k not in k_comps:
155: (12)                       result[k] = list(_consolidate(result[k + 1], k))
156: (8)                    else:
157: (12)                       nodes_at_k = set.union(*k_comps[k])
158: (12)                       to_add = [c for c in result[k + 1] if any(n not in nodes_at_k for
n in c)]
159: (12)                       if to_add:
160: (16)                           result[k] = list(_consolidate(k_comps[k] + to_add, k))
161: (12)                       else:
162: (16)                           result[k] = list(_consolidate(k_comps[k], k))
163: (4)                return result
164: (0)            def build_k_number_dict(kcomps):
165: (4)                result = {}
166: (4)                for k, comps in sorted(kcomps.items(), key=itemgetter(0)):
167: (8)                    for comp in comps:
168: (12)                       for node in comp:
169: (16)                           result[node] = k
170: (4)                return result
```

File 224 - kcutsets.py:

```
1: (0)              """
2: (0)              Kanevsky all minimum node k cutsets algorithm.
3: (0)              """
4: (0)              import copy
5: (0)              from collections import defaultdict
6: (0)              from itertools import combinations
7: (0)              from operator import itemgetter
8: (0)              import networkx as nx
9: (0)              from networkx.algorithms.flow import (
10: (4)                 build_residual_network,
11: (4)                 edmonds_karp,
12: (4)                 shortest_augmenting_path,
13: (0)             )
14: (0)             from .utils import build_auxiliary_node_connectivity
15: (0)             default_flow_func = edmonds_karp
```

```
__all__ = ["all_node_cuts"]
@nx._dispatchable
def all_node_cuts(G, k=None, flow_func=None):
    r"""Returns all minimum k cutsets of an undirected graph G.
    This implementation is based on Kanevsky's algorithm [1]_ for finding all
    minimum-size node cut-sets of an undirected graph G; ie the set (or sets)
    of nodes of cardinality equal to the node connectivity of G. Thus if
    removed, would break G into two or more connected components.
    Parameters
    ----------
    G : NetworkX graph
        Undirected graph
    k : Integer
        Node connectivity of the input graph. If k is None, then it is
        computed. Default value: None.
    flow_func : function
        Function to perform the underlying flow computations. Default value is
        :func:`~networkx.algorithms.flow.edmonds_karp`. This function performs
        better in sparse graphs with right tailed degree distributions.
        :func:`~networkx.algorithms.flow.shortest_augmenting_path` will
        perform better in denser graphs.
    Returns
    -------
    cuts : a generator of node cutsets
        Each node cutset has cardinality equal to the node connectivity of
        the input graph.
    Examples
    --------
    >>> # A two-dimensional grid graph has 4 cutsets of cardinality 2
    >>> G = nx.grid_2d_graph(5, 5)
    >>> cutsets = list(nx.all_node_cuts(G))
    >>> len(cutsets)
    4
    >>> all(2 == len(cutset) for cutset in cutsets)
    True
    >>> nx.node_connectivity(G)
    2
    Notes
    -----
    This implementation is based on the sequential algorithm for finding all
    minimum-size separating vertex sets in a graph [1]_. The main idea is to
    compute minimum cuts using local maximum flow computations among a set
    of nodes of highest degree and all other non-adjacent nodes in the Graph.
    Once we find a minimum cut, we add an edge between the high degree
    node and the target node of the local maximum flow computation to make
    sure that we will not find that minimum cut again.
    See also
    --------
    node_connectivity
    edmonds_karp
    shortest_augmenting_path
    References
    ----------
    .. [1]  Kanevsky, A. (1993). Finding all minimum-size separating vertex
            sets in a graph. Networks 23(6), 533--541.
            http://onlinelibrary.wiley.com/doi/10.1002/net.3230230604/abstract
    """
    if not nx.is_connected(G):
        raise nx.NetworkXError("Input graph is disconnected.")
    if nx.density(G) == 1:
        yield from ()
        return
    seen = []
    H = build_auxiliary_node_connectivity(G)
    H_nodes = H.nodes  # for speed
    mapping = H.graph["mapping"]
    original_H_pred = copy.copy(H._pred)
    R = build_residual_network(H, "capacity")
    kwargs = {"capacity": "capacity", "residual": R}
```

```
85: (4)                 if flow_func is None:
86: (8)                     flow_func = default_flow_func
87: (4)                 if flow_func is shortest_augmenting_path:
88: (8)                     kwargs["two_phase"] = True
89: (4)                 if k is None:
90: (8)                     k = nx.node_connectivity(G, flow_func=flow_func)
91: (4)                 X = {n for n, d in sorted(G.degree(), key=itemgetter(1), reverse=True)
[:k]}
92: (4)                 if _is_separating_set(G, X):
93: (8)                     seen.append(X)
94: (8)                     yield X
95: (4)                 for x in X:
96: (8)                     non_adjacent = set(G) - {x} - set(G[x])
97: (8)                     for v in non_adjacent:
98: (12)                        R = flow_func(H, f"{mapping[x]}B", f"{mapping[v]}A", **kwargs)
99: (12)                        flow_value = R.graph["flow_value"]
100: (12)                       if flow_value == k:
101: (16)                           E1 = flowed_edges = [
102: (20)                               (u, w) for (u, w, d) in R.edges(data=True) if d["flow"] !=
0
103: (16)                           ]
104: (16)                           VE1 = incident_nodes = {n for edge in E1 for n in edge}
105: (16)                           saturated_edges = [
106: (20)                               (u, w, d)
107: (20)                               for (u, w, d) in R.edges(data=True)
108: (20)                               if d["capacity"] == d["flow"] or d["capacity"] == 0
109: (16)                           ]
110: (16)                           R.remove_edges_from(saturated_edges)
111: (16)                           R_closure = nx.transitive_closure(R)
112: (16)                           L = nx.condensation(R)
113: (16)                           cmap = L.graph["mapping"]
114: (16)                           inv_cmap = defaultdict(list)
115: (16)                           for n, scc in cmap.items():
116: (20)                               inv_cmap[scc].append(n)
117: (16)                           VE1 = {cmap[n] for n in VE1}
118: (16)                           for antichain in nx.antichains(L):
119: (20)                               if not set(antichain).issubset(VE1):
120: (24)                                   continue
121: (20)                               S = set()
122: (20)                               for scc in antichain:
123: (24)                                   S.update(inv_cmap[scc])
124: (20)                               S_ancestors = set()
125: (20)                               for n in S:
126: (24)                                   S_ancestors.update(R_closure._pred[n])
127: (20)                               S.update(S_ancestors)
128: (20)                               if f"{mapping[x]}B" not in S or f"{mapping[v]}A" in S:
129: (24)                                   continue
130: (20)                               cutset = set()
131: (20)                               for u in S:
132: (24)                                   cutset.update((u, w) for w in original_H_pred[u] if w
not in S)
133: (20)                               if any(H_nodes[u]["id"] != H_nodes[w]["id"] for u, w in
cutset):
134: (24)                                   continue
135: (20)                               node_cut = {H_nodes[u]["id"] for u, _ in cutset}
136: (20)                               if len(node_cut) == k:
137: (24)                                   if x in node_cut or v in node_cut:
138: (28)                                       continue
139: (24)                                   if node_cut not in seen:
140: (28)                                       yield node_cut
141: (28)                                       seen.append(node_cut)
142: (16)                           H.add_edge(f"{mapping[x]}B", f"{mapping[v]}A", capacity=1)
143: (16)                           H.add_edge(f"{mapping[v]}B", f"{mapping[x]}A", capacity=1)
144: (16)                           R.add_edge(f"{mapping[x]}B", f"{mapping[v]}A", capacity=1)
145: (16)                           R.add_edge(f"{mapping[v]}A", f"{mapping[x]}B", capacity=0)
146: (16)                           R.add_edge(f"{mapping[v]}B", f"{mapping[x]}A", capacity=1)
147: (16)                           R.add_edge(f"{mapping[x]}A", f"{mapping[v]}B", capacity=0)
148: (16)                           R.add_edges_from(saturated_edges)
149: (0)            def _is_separating_set(G, cut):
```

```
150: (4)                """Assumes that the input graph is connected"""
151: (4)                if len(cut) == len(G) - 1:
152: (8)                    return True
153: (4)                H = nx.restricted_view(G, cut, [])
154: (4)                if nx.is_connected(H):
155: (8)                    return False
156: (4)                return True
```

File 225 - stoerwagner.py:

```
1: (0)              """
2: (0)              Stoer-Wagner minimum cut algorithm.
3: (0)              """
4: (0)              from itertools import islice
5: (0)              import networkx as nx
6: (0)              from ...utils import BinaryHeap, arbitrary_element, not_implemented_for
7: (0)              __all__ = ["stoer_wagner"]
8: (0)              @not_implemented_for("directed")
9: (0)              @not_implemented_for("multigraph")
10: (0)             @nx._dispatchable(edge_attrs="weight")
11: (0)             def stoer_wagner(G, weight="weight", heap=BinaryHeap):
12: (4)                 r"""Returns the weighted minimum edge cut using the Stoer-Wagner
algorithm.
13: (4)                 Determine the minimum edge cut of a connected graph using the
14: (4)                 Stoer-Wagner algorithm. In weighted cases, all weights must be
15: (4)                 nonnegative.
16: (4)                 The running time of the algorithm depends on the type of heaps used:
17: (4)                 ============== ============================================
18: (4)                 Type of heap   Running time
19: (4)                 ============== ============================================
20: (4)                 Binary heap    $O(n (m + n) \log n)$
21: (4)                 Fibonacci heap $O(nm + n^2 \log n)$
22: (4)                 Pairing heap   $O(2^{2 \sqrt{\log \log n}} nm + n^2 \log n)$
23: (4)                 ============== ============================================
24: (4)                 Parameters
25: (4)                 ----------
26: (4)                 G : NetworkX graph
27: (8)                     Edges of the graph are expected to have an attribute named by the
28: (8)                     weight parameter below. If this attribute is not present, the edge is
29: (8)                     considered to have unit weight.
30: (4)                 weight : string
31: (8)                     Name of the weight attribute of the edges. If the attribute is not
32: (8)                     present, unit weight is assumed. Default value: 'weight'.
33: (4)                 heap : class
34: (8)                     Type of heap to be used in the algorithm. It should be a subclass of
35: (8)                     :class:`MinHeap` or implement a compatible interface.
36: (8)                     If a stock heap implementation is to be used, :class:`BinaryHeap` is
37: (8)                     recommended over :class:`PairingHeap` for Python implementations
without
38: (8)                     optimized attribute accesses (e.g., CPython) despite a slower
39: (8)                     asymptotic running time. For Python implementations with optimized
40: (8)                     attribute accesses (e.g., PyPy), :class:`PairingHeap` provides better
41: (8)                     performance. Default value: :class:`BinaryHeap`.
42: (4)                 Returns
43: (4)                 -------
44: (4)                 cut_value : integer or float
45: (8)                     The sum of weights of edges in a minimum cut.
46: (4)                 partition : pair of node lists
47: (8)                     A partitioning of the nodes that defines a minimum cut.
48: (4)                 Raises
49: (4)                 ------
50: (4)                 NetworkXNotImplemented
51: (8)                     If the graph is directed or a multigraph.
52: (4)                 NetworkXError
53: (8)                     If the graph has less than two nodes, is not connected or has a
54: (8)                     negative-weighted edge.
55: (4)                 Examples
```

```
56: (4)                 --------
57: (4)                 >>> G = nx.Graph()
58: (4)                 >>> G.add_edge("x", "a", weight=3)
59: (4)                 >>> G.add_edge("x", "b", weight=1)
60: (4)                 >>> G.add_edge("a", "c", weight=3)
61: (4)                 >>> G.add_edge("b", "c", weight=5)
62: (4)                 >>> G.add_edge("b", "d", weight=4)
63: (4)                 >>> G.add_edge("d", "e", weight=2)
64: (4)                 >>> G.add_edge("c", "y", weight=2)
65: (4)                 >>> G.add_edge("e", "y", weight=3)
66: (4)                 >>> cut_value, partition = nx.stoer_wagner(G)
67: (4)                 >>> cut_value
68: (4)                 4
69: (4)                 """
70: (4)                 n = len(G)
71: (4)                 if n < 2:
72: (8)                     raise nx.NetworkXError("graph has less than two nodes.")
73: (4)                 if not nx.is_connected(G):
74: (8)                     raise nx.NetworkXError("graph is not connected.")
75: (4)                 G = nx.Graph(
76: (8)                     (u, v, {"weight": e.get(weight, 1)}) for u, v, e in G.edges(data=True)
if u != v
77: (4)                 )
78: (4)                 G.__networkx_cache__ = None  # Disable caching
79: (4)                 for u, v, e in G.edges(data=True):
80: (8)                     if e["weight"] < 0:
81: (12)                        raise nx.NetworkXError("graph has a negative-weighted edge.")
82: (4)                 cut_value = float("inf")
83: (4)                 nodes = set(G)
84: (4)                 contractions = []  # contracted node pairs
85: (4)                 for i in range(n - 1):
86: (8)                     u = arbitrary_element(G)
87: (8)                     A = {u}
88: (8)                     h = heap()  # min-heap emulating a max-heap
89: (8)                     for v, e in G[u].items():
90: (12)                        h.insert(v, -e["weight"])
91: (8)                     for j in range(n - i - 2):
92: (12)                        u = h.pop()[0]
93: (12)                        A.add(u)
94: (12)                        for v, e in G[u].items():
95: (16)                            if v not in A:
96: (20)                                h.insert(v, h.get(v, 0) - e["weight"])
97: (8)                     v, w = h.min()
98: (8)                     w = -w
99: (8)                     if w < cut_value:
100: (12)                       cut_value = w
101: (12)                       best_phase = i
102: (8)                    contractions.append((u, v))
103: (8)                    for w, e in G[v].items():
104: (12)                       if w != u:
105: (16)                           if w not in G[u]:
106: (20)                               G.add_edge(u, w, weight=e["weight"])
107: (16)                           else:
108: (20)                               G[u][w]["weight"] += e["weight"]
109: (8)                    G.remove_node(v)
110: (4)                G = nx.Graph(islice(contractions, best_phase))
111: (4)                v = contractions[best_phase][1]
112: (4)                G.add_node(v)
113: (4)                reachable = set(nx.single_source_shortest_path_length(G, v))
114: (4)                partition = (list(reachable), list(nodes - reachable))
115: (4)                return cut_value, partition

----------------------------------------

File 226 - utils.py:

1: (0)              """
2: (0)              Utilities for connectivity package
3: (0)              """
```

```
4: (0)              import networkx as nx
5: (0)              __all__ = ["build_auxiliary_node_connectivity",
"build_auxiliary_edge_connectivity"]
6: (0)              @nx._dispatchable(returns_graph=True)
7: (0)              def build_auxiliary_node_connectivity(G):
8: (4)                  r"""Creates a directed graph D from an undirected graph G to compute flow
9: (4)                  based node connectivity.
10: (4)                 For an undirected graph G having `n` nodes and `m` edges we derive a
11: (4)                 directed graph D with `2n` nodes and `2m+n` arcs by replacing each
12: (4)                 original node `v` with two nodes `vA`, `vB` linked by an (internal)
13: (4)                 arc in D. Then for each edge (`u`, `v`) in G we add two arcs (`uB`, `vA`)
14: (4)                 and (`vB`, `uA`) in D. Finally we set the attribute capacity = 1 for each
15: (4)                 arc in D [1]_.
16: (4)                 For a directed graph having `n` nodes and `m` arcs we derive a
17: (4)                 directed graph D with `2n` nodes and `m+n` arcs by replacing each
18: (4)                 original node `v` with two nodes `vA`, `vB` linked by an (internal)
19: (4)                 arc (`vA`, `vB`) in D. Then for each arc (`u`, `v`) in G we add one
20: (4)                 arc (`uB`, `vA`) in D. Finally we set the attribute capacity = 1 for
21: (4)                 each arc in D.
22: (4)                 A dictionary with a mapping between nodes in the original graph and the
23: (4)                 auxiliary digraph is stored as a graph attribute: D.graph['mapping'].
24: (4)                 References
25: (4)                 ----------
26: (4)                 .. [1] Kammer, Frank and Hanjo Taubig. Graph Connectivity. in Brandes and
27: (8)                     Erlebach, 'Network Analysis: Methodological Foundations', Lecture
28: (8)                     Notes in Computer Science, Volume 3418, Springer-Verlag, 2005.
29: (8)                     https://doi.org/10.1007/978-3-540-31955-9_7
30: (4)                 """
31: (4)                 directed = G.is_directed()
32: (4)                 mapping = {}
33: (4)                 H = nx.DiGraph()
34: (4)                 for i, node in enumerate(G):
35: (8)                     mapping[node] = i
36: (8)                     H.add_node(f"{i}A", id=node)
37: (8)                     H.add_node(f"{i}B", id=node)
38: (8)                     H.add_edge(f"{i}A", f"{i}B", capacity=1)
39: (4)                 edges = []
40: (4)                 for source, target in G.edges():
41: (8)                     edges.append((f"{mapping[source]}B", f"{mapping[target]}A"))
42: (8)                     if not directed:
43: (12)                        edges.append((f"{mapping[target]}B", f"{mapping[source]}A"))
44: (4)                 H.add_edges_from(edges, capacity=1)
45: (4)                 H.graph["mapping"] = mapping
46: (4)                 return H
47: (0)             @nx._dispatchable(returns_graph=True)
48: (0)             def build_auxiliary_edge_connectivity(G):
49: (4)                 """Auxiliary digraph for computing flow based edge connectivity
50: (4)                 If the input graph is undirected, we replace each edge (`u`,`v`) with
51: (4)                 two reciprocal arcs (`u`, `v`) and (`v`, `u`) and then we set the
attribute
52: (4)                 'capacity' for each arc to 1. If the input graph is directed we simply
53: (4)                 add the 'capacity' attribute. Part of algorithm 1 in [1]_ .
54: (4)                 References
55: (4)                 ----------
56: (4)                 .. [1] Abdol-Hossein Esfahanian. Connectivity Algorithms. (this is a
57: (8)                     chapter, look for the reference of the book).
58: (8)                     http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf
59: (4)                 """
60: (4)                 if G.is_directed():
61: (8)                     H = nx.DiGraph()
62: (8)                     H.add_nodes_from(G.nodes())
63: (8)                     H.add_edges_from(G.edges(), capacity=1)
64: (8)                     return H
65: (4)                 else:
66: (8)                     H = nx.DiGraph()
67: (8)                     H.add_nodes_from(G.nodes())
68: (8)                     for source, target in G.edges():
69: (12)                        H.add_edges_from([(source, target), (target, source)], capacity=1)
70: (8)                     return H
```

```
----------------------------------------

File 227 - __init__.py:

1: (0)

----------------------------------------

File 228 - test_connectivity.py:

1: (0)              import itertools
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms import flow
5: (0)              from networkx.algorithms.connectivity import (
6: (4)                  local_edge_connectivity,
7: (4)                  local_node_connectivity,
8: (0)              )
9: (0)              flow_funcs = [
10: (4)                 flow.boykov_kolmogorov,
11: (4)                 flow.dinitz,
12: (4)                 flow.edmonds_karp,
13: (4)                 flow.preflow_push,
14: (4)                 flow.shortest_augmenting_path,
15: (0)             ]
16: (0)             def _generate_no_biconnected(max_attempts=50):
17: (4)                 attempts = 0
18: (4)                 while True:
19: (8)                     G = nx.fast_gnp_random_graph(100, 0.0575, seed=42)
20: (8)                     if nx.is_connected(G) and not nx.is_biconnected(G):
21: (12)                        attempts = 0
22: (12)                        yield G
23: (8)                     else:
24: (12)                        if attempts >= max_attempts:
25: (16)                            msg = f"Tried {max_attempts} times: no suitable Graph."
26: (16)                            raise Exception(msg)
27: (12)                        else:
28: (16)                            attempts += 1
29: (0)             def test_average_connectivity():
30: (4)                 G1 = nx.path_graph(3)
31: (4)                 G1.add_edges_from([(1, 3), (1, 4)])
32: (4)                 G2 = nx.path_graph(3)
33: (4)                 G2.add_edges_from([(1, 3), (1, 4), (0, 3), (0, 4), (3, 4)])
34: (4)                 G3 = nx.Graph()
35: (4)                 for flow_func in flow_funcs:
36: (8)                     kwargs = {"flow_func": flow_func}
37: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
38: (8)                     assert nx.average_node_connectivity(G1, **kwargs) == 1, errmsg
39: (8)                     assert nx.average_node_connectivity(G2, **kwargs) == 2.2, errmsg
40: (8)                     assert nx.average_node_connectivity(G3, **kwargs) == 0, errmsg
41: (0)             def test_average_connectivity_directed():
42: (4)                 G = nx.DiGraph([(1, 3), (1, 4), (1, 5)])
43: (4)                 for flow_func in flow_funcs:
44: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
45: (8)                     assert nx.average_node_connectivity(G) == 0.25, errmsg
46: (0)             def test_articulation_points():
47: (4)                 Ggen = _generate_no_biconnected()
48: (4)                 for flow_func in flow_funcs:
49: (8)                     for i in range(3):
50: (12)                        G = next(Ggen)
51: (12)                        errmsg = f"Assertion failed in function: {flow_func.__name__}"
52: (12)                        assert nx.node_connectivity(G, flow_func=flow_func) == 1, errmsg
53: (0)             def test_brandes_erlebach():
54: (4)                 G = nx.Graph()
55: (4)                 G.add_edges_from(
56: (8)                     [
57: (12)                        (1, 2),
58: (12)                        (1, 3),
```

```
59: (12)                        (1, 4),
60: (12)                        (1, 5),
61: (12)                        (2, 3),
62: (12)                        (2, 6),
63: (12)                        (3, 4),
64: (12)                        (3, 6),
65: (12)                        (4, 6),
66: (12)                        (4, 7),
67: (12)                        (5, 7),
68: (12)                        (6, 8),
69: (12)                        (6, 9),
70: (12)                        (7, 8),
71: (12)                        (7, 10),
72: (12)                        (8, 11),
73: (12)                        (9, 10),
74: (12)                        (9, 11),
75: (12)                        (10, 11),
76: (8)                     ]
77: (4)                 )
78: (4)                 for flow_func in flow_funcs:
79: (8)                     kwargs = {"flow_func": flow_func}
80: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
81: (8)                     assert 3 == local_edge_connectivity(G, 1, 11, **kwargs), errmsg
82: (8)                     assert 3 == nx.edge_connectivity(G, 1, 11, **kwargs), errmsg
83: (8)                     assert 2 == local_node_connectivity(G, 1, 11, **kwargs), errmsg
84: (8)                     assert 2 == nx.node_connectivity(G, 1, 11, **kwargs), errmsg
85: (8)                     assert 2 == nx.edge_connectivity(G, **kwargs), errmsg
86: (8)                     assert 2 == nx.node_connectivity(G, **kwargs), errmsg
87: (8)                     if flow_func is flow.preflow_push:
88: (12)                        assert 3 == nx.edge_connectivity(G, 1, 11, cutoff=2, **kwargs),
errmsg
89: (8)                     else:
90: (12)                        assert 2 == nx.edge_connectivity(G, 1, 11, cutoff=2, **kwargs),
errmsg
91: (0)             def test_white_harary_1():
92: (4)                 G = nx.disjoint_union(nx.complete_graph(4), nx.complete_graph(4))
93: (4)                 G.remove_node(7)
94: (4)                 for i in range(4, 7):
95: (8)                     G.add_edge(0, i)
96: (4)                 G = nx.disjoint_union(G, nx.complete_graph(4))
97: (4)                 G.remove_node(G.order() - 1)
98: (4)                 for i in range(7, 10):
99: (8)                     G.add_edge(0, i)
100: (4)                for flow_func in flow_funcs:
101: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
102: (8)                    assert 1 == nx.node_connectivity(G, flow_func=flow_func), errmsg
103: (8)                    assert 3 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
104: (0)            def test_white_harary_2():
105: (4)                G = nx.disjoint_union(nx.complete_graph(4), nx.complete_graph(4))
106: (4)                G.add_edge(0, 4)
107: (4)                assert 3 == min(nx.core_number(G).values())
108: (4)                for flow_func in flow_funcs:
109: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
110: (8)                    assert 1 == nx.node_connectivity(G, flow_func=flow_func), errmsg
111: (8)                    assert 1 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
112: (0)            def test_complete_graphs():
113: (4)                for n in range(5, 20, 5):
114: (8)                    for flow_func in flow_funcs:
115: (12)                       G = nx.complete_graph(n)
116: (12)                       errmsg = f"Assertion failed in function: {flow_func.__name__}"
117: (12)                       assert n - 1 == nx.node_connectivity(G, flow_func=flow_func),
errmsg
118: (12)                       assert n - 1 == nx.node_connectivity(
119: (16)                           G.to_directed(), flow_func=flow_func
120: (12)                       ), errmsg
121: (12)                       assert n - 1 == nx.edge_connectivity(G, flow_func=flow_func),
errmsg
122: (12)                       assert n - 1 == nx.edge_connectivity(
123: (16)                           G.to_directed(), flow_func=flow_func
```

```
(12)                        ), errmsg
(0)             def test_empty_graphs():
(4)                 for k in range(5, 25, 5):
(8)                     G = nx.empty_graph(k)
(8)                     for flow_func in flow_funcs:
(12)                        errmsg = f"Assertion failed in function: {flow_func.__name__}"
(12)                        assert 0 == nx.node_connectivity(G, flow_func=flow_func), errmsg
(12)                        assert 0 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
(0)             def test_petersen():
(4)                 G = nx.petersen_graph()
(4)                 for flow_func in flow_funcs:
(8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
(8)                     assert 3 == nx.node_connectivity(G, flow_func=flow_func), errmsg
(8)                     assert 3 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
(0)             def test_tutte():
(4)                 G = nx.tutte_graph()
(4)                 for flow_func in flow_funcs:
(8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
(8)                     assert 3 == nx.node_connectivity(G, flow_func=flow_func), errmsg
(8)                     assert 3 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
(0)             def test_dodecahedral():
(4)                 G = nx.dodecahedral_graph()
(4)                 for flow_func in flow_funcs:
(8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
(8)                     assert 3 == nx.node_connectivity(G, flow_func=flow_func), errmsg
(8)                     assert 3 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
(0)             def test_octahedral():
(4)                 G = nx.octahedral_graph()
(4)                 for flow_func in flow_funcs:
(8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
(8)                     assert 4 == nx.node_connectivity(G, flow_func=flow_func), errmsg
(8)                     assert 4 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
(0)             def test_icosahedral():
(4)                 G = nx.icosahedral_graph()
(4)                 for flow_func in flow_funcs:
(8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
(8)                     assert 5 == nx.node_connectivity(G, flow_func=flow_func), errmsg
(8)                     assert 5 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
(0)             def test_missing_source():
(4)                 G = nx.path_graph(4)
(4)                 for flow_func in flow_funcs:
(8)                     pytest.raises(
(12)                        nx.NetworkXError, nx.node_connectivity, G, 10, 1,
flow_func=flow_func
(8)                     )
(0)             def test_missing_target():
(4)                 G = nx.path_graph(4)
(4)                 for flow_func in flow_funcs:
(8)                     pytest.raises(
(12)                        nx.NetworkXError, nx.node_connectivity, G, 1, 10,
flow_func=flow_func
(8)                     )
(0)             def test_edge_missing_source():
(4)                 G = nx.path_graph(4)
(4)                 for flow_func in flow_funcs:
(8)                     pytest.raises(
(12)                        nx.NetworkXError, nx.edge_connectivity, G, 10, 1,
flow_func=flow_func
(8)                     )
(0)             def test_edge_missing_target():
(4)                 G = nx.path_graph(4)
(4)                 for flow_func in flow_funcs:
(8)                     pytest.raises(
(12)                        nx.NetworkXError, nx.edge_connectivity, G, 1, 10,
flow_func=flow_func
(8)                     )
(0)             def test_not_weakly_connected():
(4)                 G = nx.DiGraph()
(4)                 nx.add_path(G, [1, 2, 3])
```

```
189: (4)                nx.add_path(G, [4, 5])
190: (4)                for flow_func in flow_funcs:
191: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
192: (8)                    assert nx.node_connectivity(G) == 0, errmsg
193: (8)                    assert nx.edge_connectivity(G) == 0, errmsg
194: (0)            def test_not_connected():
195: (4)                G = nx.Graph()
196: (4)                nx.add_path(G, [1, 2, 3])
197: (4)                nx.add_path(G, [4, 5])
198: (4)                for flow_func in flow_funcs:
199: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
200: (8)                    assert nx.node_connectivity(G) == 0, errmsg
201: (8)                    assert nx.edge_connectivity(G) == 0, errmsg
202: (0)            def test_directed_edge_connectivity():
203: (4)                G = nx.cycle_graph(10, create_using=nx.DiGraph())  # only one direction
204: (4)                D = nx.cycle_graph(10).to_directed()  # 2 reciprocal edges
205: (4)                for flow_func in flow_funcs:
206: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
207: (8)                    assert 1 == nx.edge_connectivity(G, flow_func=flow_func), errmsg
208: (8)                    assert 1 == local_edge_connectivity(G, 1, 4, flow_func=flow_func),
errmsg
209: (8)                    assert 1 == nx.edge_connectivity(G, 1, 4, flow_func=flow_func), errmsg
210: (8)                    assert 2 == nx.edge_connectivity(D, flow_func=flow_func), errmsg
211: (8)                    assert 2 == local_edge_connectivity(D, 1, 4, flow_func=flow_func),
errmsg
212: (8)                    assert 2 == nx.edge_connectivity(D, 1, 4, flow_func=flow_func), errmsg
213: (0)            def test_cutoff():
214: (4)                G = nx.complete_graph(5)
215: (4)                for local_func in [local_edge_connectivity, local_node_connectivity]:
216: (8)                    for flow_func in flow_funcs:
217: (12)                       if flow_func is flow.preflow_push:
218: (16)                           continue
219: (12)                       for cutoff in [3, 2, 1]:
220: (16)                           result = local_func(G, 0, 4, flow_func=flow_func,
cutoff=cutoff)
221: (16)                           assert cutoff == result, f"cutoff error in
{flow_func.__name__}"
222: (0)            def test_invalid_auxiliary():
223: (4)                G = nx.complete_graph(5)
224: (4)                pytest.raises(nx.NetworkXError, local_node_connectivity, G, 0, 3,
auxiliary=G)
225: (0)            def test_interface_only_source():
226: (4)                G = nx.complete_graph(5)
227: (4)                for interface_func in [nx.node_connectivity, nx.edge_connectivity]:
228: (8)                    pytest.raises(nx.NetworkXError, interface_func, G, s=0)
229: (0)            def test_interface_only_target():
230: (4)                G = nx.complete_graph(5)
231: (4)                for interface_func in [nx.node_connectivity, nx.edge_connectivity]:
232: (8)                    pytest.raises(nx.NetworkXError, interface_func, G, t=3)
233: (0)            def test_edge_connectivity_flow_vs_stoer_wagner():
234: (4)                graph_funcs = [nx.icosahedral_graph, nx.octahedral_graph,
nx.dodecahedral_graph]
235: (4)                for graph_func in graph_funcs:
236: (8)                    G = graph_func()
237: (8)                    assert nx.stoer_wagner(G)[0] == nx.edge_connectivity(G)
238: (0)            class TestAllPairsNodeConnectivity:
239: (4)                @classmethod
240: (4)                def setup_class(cls):
241: (8)                    cls.path = nx.path_graph(7)
242: (8)                    cls.directed_path = nx.path_graph(7, create_using=nx.DiGraph())
243: (8)                    cls.cycle = nx.cycle_graph(7)
244: (8)                    cls.directed_cycle = nx.cycle_graph(7, create_using=nx.DiGraph())
245: (8)                    cls.gnp = nx.gnp_random_graph(30, 0.1, seed=42)
246: (8)                    cls.directed_gnp = nx.gnp_random_graph(30, 0.1, directed=True,
seed=42)
247: (8)                    cls.K20 = nx.complete_graph(20)
248: (8)                    cls.K10 = nx.complete_graph(10)
249: (8)                    cls.K5 = nx.complete_graph(5)
250: (8)                    cls.G_list = [
```

```
(12)                        cls.path,
(12)                        cls.directed_path,
(12)                        cls.cycle,
(12)                        cls.directed_cycle,
(12)                        cls.gnp,
(12)                        cls.directed_gnp,
(12)                        cls.K10,
(12)                        cls.K5,
(12)                        cls.K20,
(8)                     ]
(4)                 def test_cycles(self):
(8)                     K_undir = nx.all_pairs_node_connectivity(self.cycle)
(8)                     for source in K_undir:
(12)                        for target, k in K_undir[source].items():
(16)                            assert k == 2
(8)                     K_dir = nx.all_pairs_node_connectivity(self.directed_cycle)
(8)                     for source in K_dir:
(12)                        for target, k in K_dir[source].items():
(16)                            assert k == 1
(4)                 def test_complete(self):
(8)                     for G in [self.K10, self.K5, self.K20]:
(12)                        K = nx.all_pairs_node_connectivity(G)
(12)                        for source in K:
(16)                            for target, k in K[source].items():
(20)                                assert k == len(G) - 1
(4)                 def test_paths(self):
(8)                     K_undir = nx.all_pairs_node_connectivity(self.path)
(8)                     for source in K_undir:
(12)                        for target, k in K_undir[source].items():
(16)                            assert k == 1
(8)                     K_dir = nx.all_pairs_node_connectivity(self.directed_path)
(8)                     for source in K_dir:
(12)                        for target, k in K_dir[source].items():
(16)                            if source < target:
(20)                                assert k == 1
(16)                            else:
(20)                                assert k == 0
(4)                 def test_all_pairs_connectivity_nbunch(self):
(8)                     G = nx.complete_graph(5)
(8)                     nbunch = [0, 2, 3]
(8)                     C = nx.all_pairs_node_connectivity(G, nbunch=nbunch)
(8)                     assert len(C) == len(nbunch)
(4)                 def test_all_pairs_connectivity_icosahedral(self):
(8)                     G = nx.icosahedral_graph()
(8)                     C = nx.all_pairs_node_connectivity(G)
(8)                     assert all(5 == C[u][v] for u, v in itertools.combinations(G, 2))
(4)                 def test_all_pairs_connectivity(self):
(8)                     G = nx.Graph()
(8)                     nodes = [0, 1, 2, 3]
(8)                     nx.add_path(G, nodes)
(8)                     A = {n: {} for n in G}
(8)                     for u, v in itertools.combinations(nodes, 2):
(12)                        A[u][v] = A[v][u] = nx.node_connectivity(G, u, v)
(8)                     C = nx.all_pairs_node_connectivity(G)
(8)                     assert sorted((k, sorted(v)) for k, v in A.items()) == sorted(
(12)                        (k, sorted(v)) for k, v in C.items()
(8)                     )
(4)                 def test_all_pairs_connectivity_directed(self):
(8)                     G = nx.DiGraph()
(8)                     nodes = [0, 1, 2, 3]
(8)                     nx.add_path(G, nodes)
(8)                     A = {n: {} for n in G}
(8)                     for u, v in itertools.permutations(nodes, 2):
(12)                        A[u][v] = nx.node_connectivity(G, u, v)
(8)                     C = nx.all_pairs_node_connectivity(G)
(8)                     assert sorted((k, sorted(v)) for k, v in A.items()) == sorted(
(12)                        (k, sorted(v)) for k, v in C.items()
(8)                     )
(4)                 def test_all_pairs_connectivity_nbunch_combinations(self):
```

```
320: (8)                    G = nx.complete_graph(5)
321: (8)                    nbunch = [0, 2, 3]
322: (8)                    A = {n: {} for n in nbunch}
323: (8)                    for u, v in itertools.combinations(nbunch, 2):
324: (12)                       A[u][v] = A[v][u] = nx.node_connectivity(G, u, v)
325: (8)                    C = nx.all_pairs_node_connectivity(G, nbunch=nbunch)
326: (8)                    assert sorted((k, sorted(v)) for k, v in A.items()) == sorted(
327: (12)                       (k, sorted(v)) for k, v in C.items()
328: (8)                    )
329: (4)                def test_all_pairs_connectivity_nbunch_iter(self):
330: (8)                    G = nx.complete_graph(5)
331: (8)                    nbunch = [0, 2, 3]
332: (8)                    A = {n: {} for n in nbunch}
333: (8)                    for u, v in itertools.combinations(nbunch, 2):
334: (12)                       A[u][v] = A[v][u] = nx.node_connectivity(G, u, v)
335: (8)                    C = nx.all_pairs_node_connectivity(G, nbunch=iter(nbunch))
336: (8)                    assert sorted((k, sorted(v)) for k, v in A.items()) == sorted(
337: (12)                       (k, sorted(v)) for k, v in C.items()
338: (8)                    )
```

File 229 - test_cuts.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import flow
4: (0)              from networkx.algorithms.connectivity import minimum_st_edge_cut,
minimum_st_node_cut
5: (0)              from networkx.utils import arbitrary_element
6: (0)              flow_funcs = [
7: (4)                  flow.boykov_kolmogorov,
8: (4)                  flow.dinitz,
9: (4)                  flow.edmonds_karp,
10: (4)                 flow.preflow_push,
11: (4)                 flow.shortest_augmenting_path,
12: (0)             ]
13: (0)             def _generate_no_biconnected(max_attempts=50):
14: (4)                 attempts = 0
15: (4)                 while True:
16: (8)                     G = nx.fast_gnp_random_graph(100, 0.0575, seed=42)
17: (8)                     if nx.is_connected(G) and not nx.is_biconnected(G):
18: (12)                        attempts = 0
19: (12)                        yield G
20: (8)                     else:
21: (12)                        if attempts >= max_attempts:
22: (16)                            msg = f"Tried {attempts} times: no suitable Graph."
23: (16)                            raise Exception(msg)
24: (12)                        else:
25: (16)                            attempts += 1
26: (0)             def test_articulation_points():
27: (4)                 Ggen = _generate_no_biconnected()
28: (4)                 for flow_func in flow_funcs:
29: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
30: (8)                     for i in range(1):  # change 1 to 3 or more for more realizations.
31: (12)                        G = next(Ggen)
32: (12)                        cut = nx.minimum_node_cut(G, flow_func=flow_func)
33: (12)                        assert len(cut) == 1, errmsg
34: (12)                        assert cut.pop() in set(nx.articulation_points(G)), errmsg
35: (0)             def test_brandes_erlebach_book():
36: (4)                 G = nx.Graph()
37: (4)                 G.add_edges_from(
38: (8)                     [
39: (12)                        (1, 2),
40: (12)                        (1, 3),
41: (12)                        (1, 4),
42: (12)                        (1, 5),
43: (12)                        (2, 3),
44: (12)                        (2, 6),
```

```
45: (12)                        (3, 4),
46: (12)                        (3, 6),
47: (12)                        (4, 6),
48: (12)                        (4, 7),
49: (12)                        (5, 7),
50: (12)                        (6, 8),
51: (12)                        (6, 9),
52: (12)                        (7, 8),
53: (12)                        (7, 10),
54: (12)                        (8, 11),
55: (12)                        (9, 10),
56: (12)                        (9, 11),
57: (12)                        (10, 11),
58: (8)                     ]
59: (4)                 )
60: (4)                 for flow_func in flow_funcs:
61: (8)                     kwargs = {"flow_func": flow_func}
62: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
63: (8)                     assert 3 == len(nx.minimum_edge_cut(G, 1, 11, **kwargs)), errmsg
64: (8)                     edge_cut = nx.minimum_edge_cut(G, **kwargs)
65: (8)                     assert 2 == len(edge_cut), errmsg
66: (8)                     H = G.copy()
67: (8)                     H.remove_edges_from(edge_cut)
68: (8)                     assert not nx.is_connected(H), errmsg
69: (8)                     assert {6, 7} == minimum_st_node_cut(G, 1, 11, **kwargs), errmsg
70: (8)                     assert {6, 7} == nx.minimum_node_cut(G, 1, 11, **kwargs), errmsg
71: (8)                     node_cut = nx.minimum_node_cut(G, **kwargs)
72: (8)                     assert 2 == len(node_cut), errmsg
73: (8)                     H = G.copy()
74: (8)                     H.remove_nodes_from(node_cut)
75: (8)                     assert not nx.is_connected(H), errmsg
76: (0)             def test_white_harary_paper():
77: (4)                 G = nx.disjoint_union(nx.complete_graph(4), nx.complete_graph(4))
78: (4)                 G.remove_node(7)
79: (4)                 for i in range(4, 7):
80: (8)                     G.add_edge(0, i)
81: (4)                 G = nx.disjoint_union(G, nx.complete_graph(4))
82: (4)                 G.remove_node(G.order() - 1)
83: (4)                 for i in range(7, 10):
84: (8)                     G.add_edge(0, i)
85: (4)                 for flow_func in flow_funcs:
86: (8)                     kwargs = {"flow_func": flow_func}
87: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
88: (8)                     edge_cut = nx.minimum_edge_cut(G, **kwargs)
89: (8)                     assert 3 == len(edge_cut), errmsg
90: (8)                     H = G.copy()
91: (8)                     H.remove_edges_from(edge_cut)
92: (8)                     assert not nx.is_connected(H), errmsg
93: (8)                     node_cut = nx.minimum_node_cut(G, **kwargs)
94: (8)                     assert {0} == node_cut, errmsg
95: (8)                     H = G.copy()
96: (8)                     H.remove_nodes_from(node_cut)
97: (8)                     assert not nx.is_connected(H), errmsg
98: (0)             def test_petersen_cutset():
99: (4)                 G = nx.petersen_graph()
100: (4)                for flow_func in flow_funcs:
101: (8)                    kwargs = {"flow_func": flow_func}
102: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
103: (8)                    edge_cut = nx.minimum_edge_cut(G, **kwargs)
104: (8)                    assert 3 == len(edge_cut), errmsg
105: (8)                    H = G.copy()
106: (8)                    H.remove_edges_from(edge_cut)
107: (8)                    assert not nx.is_connected(H), errmsg
108: (8)                    node_cut = nx.minimum_node_cut(G, **kwargs)
109: (8)                    assert 3 == len(node_cut), errmsg
110: (8)                    H = G.copy()
111: (8)                    H.remove_nodes_from(node_cut)
112: (8)                    assert not nx.is_connected(H), errmsg
113: (0)            def test_octahedral_cutset():
```

```
114: (4)                G = nx.octahedral_graph()
115: (4)                for flow_func in flow_funcs:
116: (8)                    kwargs = {"flow_func": flow_func}
117: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
118: (8)                    edge_cut = nx.minimum_edge_cut(G, **kwargs)
119: (8)                    assert 4 == len(edge_cut), errmsg
120: (8)                    H = G.copy()
121: (8)                    H.remove_edges_from(edge_cut)
122: (8)                    assert not nx.is_connected(H), errmsg
123: (8)                    node_cut = nx.minimum_node_cut(G, **kwargs)
124: (8)                    assert 4 == len(node_cut), errmsg
125: (8)                    H = G.copy()
126: (8)                    H.remove_nodes_from(node_cut)
127: (8)                    assert not nx.is_connected(H), errmsg
128: (0)            def test_icosahedral_cutset():
129: (4)                G = nx.icosahedral_graph()
130: (4)                for flow_func in flow_funcs:
131: (8)                    kwargs = {"flow_func": flow_func}
132: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
133: (8)                    edge_cut = nx.minimum_edge_cut(G, **kwargs)
134: (8)                    assert 5 == len(edge_cut), errmsg
135: (8)                    H = G.copy()
136: (8)                    H.remove_edges_from(edge_cut)
137: (8)                    assert not nx.is_connected(H), errmsg
138: (8)                    node_cut = nx.minimum_node_cut(G, **kwargs)
139: (8)                    assert 5 == len(node_cut), errmsg
140: (8)                    H = G.copy()
141: (8)                    H.remove_nodes_from(node_cut)
142: (8)                    assert not nx.is_connected(H), errmsg
143: (0)            def test_node_cutset_exception():
144: (4)                G = nx.Graph()
145: (4)                G.add_edges_from([(1, 2), (3, 4)])
146: (4)                for flow_func in flow_funcs:
147: (8)                    pytest.raises(nx.NetworkXError, nx.minimum_node_cut, G,
flow_func=flow_func)
148: (0)            def test_node_cutset_random_graphs():
149: (4)                for flow_func in flow_funcs:
150: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
151: (8)                    for i in range(3):
152: (12)                       G = nx.fast_gnp_random_graph(50, 0.25, seed=42)
153: (12)                       if not nx.is_connected(G):
154: (16)                           ccs = iter(nx.connected_components(G))
155: (16)                           start = arbitrary_element(next(ccs))
156: (16)                           G.add_edges_from((start, arbitrary_element(c)) for c in ccs)
157: (12)                       cutset = nx.minimum_node_cut(G, flow_func=flow_func)
158: (12)                       assert nx.node_connectivity(G) == len(cutset), errmsg
159: (12)                       G.remove_nodes_from(cutset)
160: (12)                       assert not nx.is_connected(G), errmsg
161: (0)            def test_edge_cutset_random_graphs():
162: (4)                for flow_func in flow_funcs:
163: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
164: (8)                    for i in range(3):
165: (12)                       G = nx.fast_gnp_random_graph(50, 0.25, seed=42)
166: (12)                       if not nx.is_connected(G):
167: (16)                           ccs = iter(nx.connected_components(G))
168: (16)                           start = arbitrary_element(next(ccs))
169: (16)                           G.add_edges_from((start, arbitrary_element(c)) for c in ccs)
170: (12)                       cutset = nx.minimum_edge_cut(G, flow_func=flow_func)
171: (12)                       assert nx.edge_connectivity(G) == len(cutset), errmsg
172: (12)                       G.remove_edges_from(cutset)
173: (12)                       assert not nx.is_connected(G), errmsg
174: (0)            def test_empty_graphs():
175: (4)                G = nx.Graph()
176: (4)                D = nx.DiGraph()
177: (4)                for interface_func in [nx.minimum_node_cut, nx.minimum_edge_cut]:
178: (8)                    for flow_func in flow_funcs:
179: (12)                       pytest.raises(
180: (16)                           nx.NetworkXPointlessConcept, interface_func, G,
flow_func=flow_func
```

```
181: (12)                        )
182: (12)                        pytest.raises(
183: (16)                            nx.NetworkXPointlessConcept, interface_func, D,
flow_func=flow_func
184: (12)                        )
185: (0)            def test_unbounded():
186: (4)                G = nx.complete_graph(5)
187: (4)                for flow_func in flow_funcs:
188: (8)                    assert 4 == len(minimum_st_edge_cut(G, 1, 4, flow_func=flow_func))
189: (0)            def test_missing_source():
190: (4)                G = nx.path_graph(4)
191: (4)                for interface_func in [nx.minimum_edge_cut, nx.minimum_node_cut]:
192: (8)                    for flow_func in flow_funcs:
193: (12)                       pytest.raises(
194: (16)                           nx.NetworkXError, interface_func, G, 10, 1,
flow_func=flow_func
195: (12)                       )
196: (0)            def test_missing_target():
197: (4)                G = nx.path_graph(4)
198: (4)                for interface_func in [nx.minimum_edge_cut, nx.minimum_node_cut]:
199: (8)                    for flow_func in flow_funcs:
200: (12)                       pytest.raises(
201: (16)                           nx.NetworkXError, interface_func, G, 1, 10,
flow_func=flow_func
202: (12)                       )
203: (0)            def test_not_weakly_connected():
204: (4)                G = nx.DiGraph()
205: (4)                nx.add_path(G, [1, 2, 3])
206: (4)                nx.add_path(G, [4, 5])
207: (4)                for interface_func in [nx.minimum_edge_cut, nx.minimum_node_cut]:
208: (8)                    for flow_func in flow_funcs:
209: (12)                       pytest.raises(nx.NetworkXError, interface_func, G,
flow_func=flow_func)
210: (0)            def test_not_connected():
211: (4)                G = nx.Graph()
212: (4)                nx.add_path(G, [1, 2, 3])
213: (4)                nx.add_path(G, [4, 5])
214: (4)                for interface_func in [nx.minimum_edge_cut, nx.minimum_node_cut]:
215: (8)                    for flow_func in flow_funcs:
216: (12)                       pytest.raises(nx.NetworkXError, interface_func, G,
flow_func=flow_func)
217: (0)            def tests_min_cut_complete():
218: (4)                G = nx.complete_graph(5)
219: (4)                for interface_func in [nx.minimum_edge_cut, nx.minimum_node_cut]:
220: (8)                    for flow_func in flow_funcs:
221: (12)                       assert 4 == len(interface_func(G, flow_func=flow_func))
222: (0)            def tests_min_cut_complete_directed():
223: (4)                G = nx.complete_graph(5)
224: (4)                G = G.to_directed()
225: (4)                for interface_func in [nx.minimum_edge_cut, nx.minimum_node_cut]:
226: (8)                    for flow_func in flow_funcs:
227: (12)                       assert 4 == len(interface_func(G, flow_func=flow_func))
228: (0)            def tests_minimum_st_node_cut():
229: (4)                G = nx.Graph()
230: (4)                G.add_nodes_from([0, 1, 2, 3, 7, 8, 11, 12])
231: (4)                G.add_edges_from([(7, 11), (1, 11), (1, 12), (12, 8), (0, 1)])
232: (4)                nodelist = minimum_st_node_cut(G, 7, 11)
233: (4)                assert nodelist == {}
234: (0)            def test_invalid_auxiliary():
235: (4)                G = nx.complete_graph(5)
236: (4)                pytest.raises(nx.NetworkXError, minimum_st_node_cut, G, 0, 3, auxiliary=G)
237: (0)            def test_interface_only_source():
238: (4)                G = nx.complete_graph(5)
239: (4)                for interface_func in [nx.minimum_node_cut, nx.minimum_edge_cut]:
240: (8)                    pytest.raises(nx.NetworkXError, interface_func, G, s=0)
241: (0)            def test_interface_only_target():
242: (4)                G = nx.complete_graph(5)
243: (4)                for interface_func in [nx.minimum_node_cut, nx.minimum_edge_cut]:
244: (8)                    pytest.raises(nx.NetworkXError, interface_func, G, t=3)
```

```
----------------------------------------

File 230 - test_disjoint_paths.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import flow
4: (0)              from networkx.utils import pairwise
5: (0)              flow_funcs = [
6: (4)                  flow.boykov_kolmogorov,
7: (4)                  flow.edmonds_karp,
8: (4)                  flow.dinitz,
9: (4)                  flow.preflow_push,
10: (4)                 flow.shortest_augmenting_path,
11: (0)             ]
12: (0)             def is_path(G, path):
13: (4)                 return all(v in G[u] for u, v in pairwise(path))
14: (0)             def are_edge_disjoint_paths(G, paths):
15: (4)                 if not paths:
16: (8)                     return False
17: (4)                 for path in paths:
18: (8)                     assert is_path(G, path)
19: (4)                 paths_edges = [list(pairwise(p)) for p in paths]
20: (4)                 num_of_edges = sum(len(e) for e in paths_edges)
21: (4)                 num_unique_edges = len(set.union(*[set(es) for es in paths_edges]))
22: (4)                 if num_of_edges == num_unique_edges:
23: (8)                     return True
24: (4)                 return False
25: (0)             def are_node_disjoint_paths(G, paths):
26: (4)                 if not paths:
27: (8)                     return False
28: (4)                 for path in paths:
29: (8)                     assert is_path(G, path)
30: (4)                 st = {paths[0][0], paths[0][-1]}
31: (4)                 num_of_nodes = len([n for path in paths for n in path if n not in st])
32: (4)                 num_unique_nodes = len({n for path in paths for n in path if n not in st})
33: (4)                 if num_of_nodes == num_unique_nodes:
34: (8)                     return True
35: (4)                 return False
36: (0)             def test_graph_from_pr_2053():
37: (4)                 G = nx.Graph()
38: (4)                 G.add_edges_from(
39: (8)                     [
40: (12)                        ("A", "B"),
41: (12)                        ("A", "D"),
42: (12)                        ("A", "F"),
43: (12)                        ("A", "G"),
44: (12)                        ("B", "C"),
45: (12)                        ("B", "D"),
46: (12)                        ("B", "G"),
47: (12)                        ("C", "D"),
48: (12)                        ("C", "E"),
49: (12)                        ("C", "Z"),
50: (12)                        ("D", "E"),
51: (12)                        ("D", "F"),
52: (12)                        ("E", "F"),
53: (12)                        ("E", "Z"),
54: (12)                        ("F", "Z"),
55: (12)                        ("G", "Z"),
56: (8)                     ]
57: (4)                 )
58: (4)                 for flow_func in flow_funcs:
59: (8)                     kwargs = {"flow_func": flow_func}
60: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
61: (8)                     edge_paths = list(nx.edge_disjoint_paths(G, "A", "Z", **kwargs))
62: (8)                     assert are_edge_disjoint_paths(G, edge_paths), errmsg
63: (8)                     assert nx.edge_connectivity(G, "A", "Z") == len(edge_paths), errmsg
64: (8)                     node_paths = list(nx.node_disjoint_paths(G, "A", "Z", **kwargs))
```

```
65: (8)                     assert are_node_disjoint_paths(G, node_paths), errmsg
66: (8)                     assert nx.node_connectivity(G, "A", "Z") == len(node_paths), errmsg
67: (0)             def test_florentine_families():
68: (4)                 G = nx.florentine_families_graph()
69: (4)                 for flow_func in flow_funcs:
70: (8)                     kwargs = {"flow_func": flow_func}
71: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
72: (8)                     edge_dpaths = list(nx.edge_disjoint_paths(G, "Medici", "Strozzi",
**kwargs))
73: (8)                     assert are_edge_disjoint_paths(G, edge_dpaths), errmsg
74: (8)                     assert nx.edge_connectivity(G, "Medici", "Strozzi") ==
len(edge_dpaths), errmsg
75: (8)                     node_dpaths = list(nx.node_disjoint_paths(G, "Medici", "Strozzi",
**kwargs))
76: (8)                     assert are_node_disjoint_paths(G, node_dpaths), errmsg
77: (8)                     assert nx.node_connectivity(G, "Medici", "Strozzi") ==
len(node_dpaths), errmsg
78: (0)             def test_karate():
79: (4)                 G = nx.karate_club_graph()
80: (4)                 for flow_func in flow_funcs:
81: (8)                     kwargs = {"flow_func": flow_func}
82: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
83: (8)                     edge_dpaths = list(nx.edge_disjoint_paths(G, 0, 33, **kwargs))
84: (8)                     assert are_edge_disjoint_paths(G, edge_dpaths), errmsg
85: (8)                     assert nx.edge_connectivity(G, 0, 33) == len(edge_dpaths), errmsg
86: (8)                     node_dpaths = list(nx.node_disjoint_paths(G, 0, 33, **kwargs))
87: (8)                     assert are_node_disjoint_paths(G, node_dpaths), errmsg
88: (8)                     assert nx.node_connectivity(G, 0, 33) == len(node_dpaths), errmsg
89: (0)             def test_petersen_disjoint_paths():
90: (4)                 G = nx.petersen_graph()
91: (4)                 for flow_func in flow_funcs:
92: (8)                     kwargs = {"flow_func": flow_func}
93: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
94: (8)                     edge_dpaths = list(nx.edge_disjoint_paths(G, 0, 6, **kwargs))
95: (8)                     assert are_edge_disjoint_paths(G, edge_dpaths), errmsg
96: (8)                     assert 3 == len(edge_dpaths), errmsg
97: (8)                     node_dpaths = list(nx.node_disjoint_paths(G, 0, 6, **kwargs))
98: (8)                     assert are_node_disjoint_paths(G, node_dpaths), errmsg
99: (8)                     assert 3 == len(node_dpaths), errmsg
100: (0)            def test_octahedral_disjoint_paths():
101: (4)                G = nx.octahedral_graph()
102: (4)                for flow_func in flow_funcs:
103: (8)                    kwargs = {"flow_func": flow_func}
104: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
105: (8)                    edge_dpaths = list(nx.edge_disjoint_paths(G, 0, 5, **kwargs))
106: (8)                    assert are_edge_disjoint_paths(G, edge_dpaths), errmsg
107: (8)                    assert 4 == len(edge_dpaths), errmsg
108: (8)                    node_dpaths = list(nx.node_disjoint_paths(G, 0, 5, **kwargs))
109: (8)                    assert are_node_disjoint_paths(G, node_dpaths), errmsg
110: (8)                    assert 4 == len(node_dpaths), errmsg
111: (0)            def test_icosahedral_disjoint_paths():
112: (4)                G = nx.icosahedral_graph()
113: (4)                for flow_func in flow_funcs:
114: (8)                    kwargs = {"flow_func": flow_func}
115: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
116: (8)                    edge_dpaths = list(nx.edge_disjoint_paths(G, 0, 6, **kwargs))
117: (8)                    assert are_edge_disjoint_paths(G, edge_dpaths), errmsg
118: (8)                    assert 5 == len(edge_dpaths), errmsg
119: (8)                    node_dpaths = list(nx.node_disjoint_paths(G, 0, 6, **kwargs))
120: (8)                    assert are_node_disjoint_paths(G, node_dpaths), errmsg
121: (8)                    assert 5 == len(node_dpaths), errmsg
122: (0)            def test_cutoff_disjoint_paths():
123: (4)                G = nx.icosahedral_graph()
124: (4)                for flow_func in flow_funcs:
125: (8)                    kwargs = {"flow_func": flow_func}
126: (8)                    errmsg = f"Assertion failed in function: {flow_func.__name__}"
127: (8)                    for cutoff in [2, 4]:
128: (12)                       kwargs["cutoff"] = cutoff
129: (12)                       edge_dpaths = list(nx.edge_disjoint_paths(G, 0, 6, **kwargs))
```

```
130: (12)                        assert are_edge_disjoint_paths(G, edge_dpaths), errmsg
131: (12)                        assert cutoff == len(edge_dpaths), errmsg
132: (12)                        node_dpaths = list(nx.node_disjoint_paths(G, 0, 6, **kwargs))
133: (12)                        assert are_node_disjoint_paths(G, node_dpaths), errmsg
134: (12)                        assert cutoff == len(node_dpaths), errmsg
135: (0)            def test_missing_source_edge_paths():
136: (4)                with pytest.raises(nx.NetworkXError):
137: (8)                    G = nx.path_graph(4)
138: (8)                    list(nx.edge_disjoint_paths(G, 10, 1))
139: (0)            def test_missing_source_node_paths():
140: (4)                with pytest.raises(nx.NetworkXError):
141: (8)                    G = nx.path_graph(4)
142: (8)                    list(nx.node_disjoint_paths(G, 10, 1))
143: (0)            def test_missing_target_edge_paths():
144: (4)                with pytest.raises(nx.NetworkXError):
145: (8)                    G = nx.path_graph(4)
146: (8)                    list(nx.edge_disjoint_paths(G, 1, 10))
147: (0)            def test_missing_target_node_paths():
148: (4)                with pytest.raises(nx.NetworkXError):
149: (8)                    G = nx.path_graph(4)
150: (8)                    list(nx.node_disjoint_paths(G, 1, 10))
151: (0)            def test_not_weakly_connected_edges():
152: (4)                with pytest.raises(nx.NetworkXNoPath):
153: (8)                    G = nx.DiGraph()
154: (8)                    nx.add_path(G, [1, 2, 3])
155: (8)                    nx.add_path(G, [4, 5])
156: (8)                    list(nx.edge_disjoint_paths(G, 1, 5))
157: (0)            def test_not_weakly_connected_nodes():
158: (4)                with pytest.raises(nx.NetworkXNoPath):
159: (8)                    G = nx.DiGraph()
160: (8)                    nx.add_path(G, [1, 2, 3])
161: (8)                    nx.add_path(G, [4, 5])
162: (8)                    list(nx.node_disjoint_paths(G, 1, 5))
163: (0)            def test_not_connected_edges():
164: (4)                with pytest.raises(nx.NetworkXNoPath):
165: (8)                    G = nx.Graph()
166: (8)                    nx.add_path(G, [1, 2, 3])
167: (8)                    nx.add_path(G, [4, 5])
168: (8)                    list(nx.edge_disjoint_paths(G, 1, 5))
169: (0)            def test_not_connected_nodes():
170: (4)                with pytest.raises(nx.NetworkXNoPath):
171: (8)                    G = nx.Graph()
172: (8)                    nx.add_path(G, [1, 2, 3])
173: (8)                    nx.add_path(G, [4, 5])
174: (8)                    list(nx.node_disjoint_paths(G, 1, 5))
175: (0)            def test_isolated_edges():
176: (4)                with pytest.raises(nx.NetworkXNoPath):
177: (8)                    G = nx.Graph()
178: (8)                    G.add_node(1)
179: (8)                    nx.add_path(G, [4, 5])
180: (8)                    list(nx.edge_disjoint_paths(G, 1, 5))
181: (0)            def test_isolated_nodes():
182: (4)                with pytest.raises(nx.NetworkXNoPath):
183: (8)                    G = nx.Graph()
184: (8)                    G.add_node(1)
185: (8)                    nx.add_path(G, [4, 5])
186: (8)                    list(nx.node_disjoint_paths(G, 1, 5))
187: (0)            def test_invalid_auxiliary():
188: (4)                with pytest.raises(nx.NetworkXError):
189: (8)                    G = nx.complete_graph(5)
190: (8)                    list(nx.node_disjoint_paths(G, 0, 3, auxiliary=G))

----------------------------------------

File 231 - test_edge_augmentation.py:

1: (0)              import itertools as it
2: (0)              import random
3: (0)              import pytest
```

```
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.connectivity import k_edge_augmentation
6: (0)              from networkx.algorithms.connectivity.edge_augmentation import (
7: (4)                  _unpack_available_edges,
8: (4)                  collapse,
9: (4)                  complement_edges,
10: (4)                 is_k_edge_connected,
11: (4)                 is_locally_k_edge_connected,
12: (0)             )
13: (0)             from networkx.utils import pairwise
14: (0)             MAX_EFFICIENT_K = 2
15: (0)             def tarjan_bridge_graph():
16: (4)                 ccs = [
17: (8)                     (1, 2, 4, 3, 1, 4),
18: (8)                     (5, 6, 7, 5),
19: (8)                     (8, 9, 10, 8),
20: (8)                     (17, 18, 16, 15, 17),
21: (8)                     (11, 12, 14, 13, 11, 14),
22: (4)                 ]
23: (4)                 bridges = [(4, 8), (3, 5), (3, 17)]
24: (4)                 G = nx.Graph(it.chain(*(pairwise(path) for path in ccs + bridges)))
25: (4)                 return G
26: (0)             def test_weight_key():
27: (4)                 G = nx.Graph()
28: (4)                 G.add_nodes_from([1, 2, 3, 4, 5, 6, 7, 8, 9])
29: (4)                 G.add_edges_from([(3, 8), (1, 2), (2, 3)])
30: (4)                 impossible = {(3, 6), (3, 9)}
31: (4)                 rng = random.Random(0)
32: (4)                 avail_uv = list(set(complement_edges(G)) - impossible)
33: (4)                 avail = [(u, v, {"cost": rng.random()}) for u, v in avail_uv]
34: (4)                 _augment_and_check(G, k=1)
35: (4)                 _augment_and_check(G, k=1, avail=avail_uv)
36: (4)                 _augment_and_check(G, k=1, avail=avail, weight="cost")
37: (4)                 _check_augmentations(G, avail, weight="cost")
38: (0)             def test_is_locally_k_edge_connected_exceptions():
39: (4)                 pytest.raises(nx.NetworkXNotImplemented, is_k_edge_connected,
nx.DiGraph(), k=0)
40: (4)                 pytest.raises(nx.NetworkXNotImplemented, is_k_edge_connected,
nx.MultiGraph(), k=0)
41: (4)                 pytest.raises(ValueError, is_k_edge_connected, nx.Graph(), k=0)
42: (0)             def test_is_k_edge_connected():
43: (4)                 G = nx.barbell_graph(10, 0)
44: (4)                 assert is_k_edge_connected(G, k=1)
45: (4)                 assert not is_k_edge_connected(G, k=2)
46: (4)                 G = nx.Graph()
47: (4)                 G.add_nodes_from([5, 15])
48: (4)                 assert not is_k_edge_connected(G, k=1)
49: (4)                 assert not is_k_edge_connected(G, k=2)
50: (4)                 G = nx.complete_graph(5)
51: (4)                 assert is_k_edge_connected(G, k=1)
52: (4)                 assert is_k_edge_connected(G, k=2)
53: (4)                 assert is_k_edge_connected(G, k=3)
54: (4)                 assert is_k_edge_connected(G, k=4)
55: (4)                 G = nx.compose(nx.complete_graph([0, 1, 2]), nx.complete_graph([3, 4, 5]))
56: (4)                 assert not is_k_edge_connected(G, k=1)
57: (4)                 assert not is_k_edge_connected(G, k=2)
58: (4)                 assert not is_k_edge_connected(G, k=3)
59: (0)             def test_is_k_edge_connected_exceptions():
60: (4)                 pytest.raises(
61: (8)                     nx.NetworkXNotImplemented, is_locally_k_edge_connected, nx.DiGraph(),
1, 2, k=0
62: (4)                 )
63: (4)                 pytest.raises(
64: (8)                     nx.NetworkXNotImplemented,
65: (8)                     is_locally_k_edge_connected,
66: (8)                     nx.MultiGraph(),
67: (8)                     1,
68: (8)                     2,
69: (8)                     k=0,
```

```
70: (4)                 )
71: (4)                 pytest.raises(ValueError, is_locally_k_edge_connected, nx.Graph(), 1, 2,
k=0)
72: (0)             def test_is_locally_k_edge_connected():
73: (4)                 G = nx.barbell_graph(10, 0)
74: (4)                 assert is_locally_k_edge_connected(G, 5, 15, k=1)
75: (4)                 assert not is_locally_k_edge_connected(G, 5, 15, k=2)
76: (4)                 G = nx.Graph()
77: (4)                 G.add_nodes_from([5, 15])
78: (4)                 assert not is_locally_k_edge_connected(G, 5, 15, k=2)
79: (0)             def test_null_graph():
80: (4)                 G = nx.Graph()
81: (4)                 _check_augmentations(G, max_k=MAX_EFFICIENT_K + 2)
82: (0)             def test_cliques():
83: (4)                 for n in range(1, 10):
84: (8)                     G = nx.complete_graph(n)
85: (8)                     _check_augmentations(G, max_k=MAX_EFFICIENT_K + 2)
86: (0)             def test_clique_and_node():
87: (4)                 for n in range(1, 10):
88: (8)                     G = nx.complete_graph(n)
89: (8)                     G.add_node(n + 1)
90: (8)                     _check_augmentations(G, max_k=MAX_EFFICIENT_K + 2)
91: (0)             def test_point_graph():
92: (4)                 G = nx.Graph()
93: (4)                 G.add_node(1)
94: (4)                 _check_augmentations(G, max_k=MAX_EFFICIENT_K + 2)
95: (0)             def test_edgeless_graph():
96: (4)                 G = nx.Graph()
97: (4)                 G.add_nodes_from([1, 2, 3, 4])
98: (4)                 _check_augmentations(G)
99: (0)             def test_invalid_k():
100: (4)                G = nx.Graph()
101: (4)                pytest.raises(ValueError, list, k_edge_augmentation(G, k=-1))
102: (4)                pytest.raises(ValueError, list, k_edge_augmentation(G, k=0))
103: (0)            def test_unfeasible():
104: (4)                G = tarjan_bridge_graph()
105: (4)                pytest.raises(nx.NetworkXUnfeasible, list, k_edge_augmentation(G, k=1,
avail=[]))
106: (4)                pytest.raises(nx.NetworkXUnfeasible, list, k_edge_augmentation(G, k=2,
avail=[]))
107: (4)                pytest.raises(
108: (8)                    nx.NetworkXUnfeasible, list, k_edge_augmentation(G, k=2, avail=[(7,
9)])
109: (4)                )
110: (4)                aug_edges = list(k_edge_augmentation(G, k=2, avail=[(7, 9)],
partial=True))
111: (4)                assert aug_edges == [(7, 9)]
112: (4)                _check_augmentations(G, avail=[], max_k=MAX_EFFICIENT_K + 2)
113: (4)                _check_augmentations(G, avail=[(7, 9)], max_k=MAX_EFFICIENT_K + 2)
114: (0)            def test_tarjan():
115: (4)                G = tarjan_bridge_graph()
116: (4)                aug_edges = set(_augment_and_check(G, k=2)[0])
117: (4)                print(f"aug_edges = {aug_edges!r}")
118: (4)                assert len(aug_edges) == 3
119: (4)                avail = [
120: (8)                    (9, 7),
121: (8)                    (8, 5),
122: (8)                    (2, 10),
123: (8)                    (6, 13),
124: (8)                    (11, 18),
125: (8)                    (1, 17),
126: (8)                    (2, 3),
127: (8)                    (16, 17),
128: (8)                    (18, 14),
129: (8)                    (15, 14),
130: (4)                ]
131: (4)                aug_edges = set(_augment_and_check(G, avail=avail, k=2)[0])
132: (4)                assert len(aug_edges) <= 3 * 2
133: (4)                _check_augmentations(G, avail)
```

```
134: (0)            def test_configuration():
135: (4)                seeds = [1001, 1002, 1003, 1004]
136: (4)                for seed in seeds:
137: (8)                    deg_seq = nx.random_powerlaw_tree_sequence(20, seed=seed, tries=5000)
138: (8)                    G = nx.Graph(nx.configuration_model(deg_seq, seed=seed))
139: (8)                    G.remove_edges_from(nx.selfloop_edges(G))
140: (8)                    _check_augmentations(G)
141: (0)            def test_shell():
142: (4)                seeds = [18]
143: (4)                for seed in seeds:
144: (8)                    constructor = [(12, 70, 0.8), (15, 40, 0.6)]
145: (8)                    G = nx.random_shell_graph(constructor, seed=seed)
146: (8)                    _check_augmentations(G)
147: (0)            def test_karate():
148: (4)                G = nx.karate_club_graph()
149: (4)                _check_augmentations(G)
150: (0)            def test_star():
151: (4)                G = nx.star_graph(3)
152: (4)                _check_augmentations(G)
153: (4)                G = nx.star_graph(5)
154: (4)                _check_augmentations(G)
155: (4)                G = nx.star_graph(10)
156: (4)                _check_augmentations(G)
157: (0)            def test_barbell():
158: (4)                G = nx.barbell_graph(5, 0)
159: (4)                _check_augmentations(G)
160: (4)                G = nx.barbell_graph(5, 2)
161: (4)                _check_augmentations(G)
162: (4)                G = nx.barbell_graph(5, 3)
163: (4)                _check_augmentations(G)
164: (4)                G = nx.barbell_graph(5, 4)
165: (4)                _check_augmentations(G)
166: (0)            def test_bridge():
167: (4)                G = nx.Graph([(2393, 2257), (2393, 2685), (2685, 2257), (1758, 2257)])
168: (4)                _check_augmentations(G)
169: (0)            def test_gnp_augmentation():
170: (4)                rng = random.Random(0)
171: (4)                G = nx.gnp_random_graph(30, 0.005, seed=0)
172: (4)                avail = {
173: (8)                    (u, v): 1 + rng.random() for u, v in complement_edges(G) if
rng.random() < 0.25
174: (4)                }
175: (4)                _check_augmentations(G, avail)
176: (0)            def _assert_solution_properties(G, aug_edges, avail_dict=None):
177: (4)                """Checks that aug_edges are consistently formatted"""
178: (4)                if avail_dict is not None:
179: (8)                    assert all(
180: (12)                       e in avail_dict for e in aug_edges
181: (8)                    ), "when avail is specified aug-edges should be in avail"
182: (4)                unique_aug = set(map(tuple, map(sorted, aug_edges)))
183: (4)                unique_aug = list(map(tuple, map(sorted, aug_edges)))
184: (4)                assert len(aug_edges) == len(unique_aug), "edges should be unique"
185: (4)                assert not any(u == v for u, v in unique_aug), "should be no self-edges"
186: (4)                assert not any(
187: (8)                    G.has_edge(u, v) for u, v in unique_aug
188: (4)                ), "aug edges and G.edges should be disjoint"
189: (0)            def _augment_and_check(
190: (4)                G, k, avail=None, weight=None, verbose=False, orig_k=None, max_aug_k=None
191: (0)            ):
192: (4)                """
193: (4)                Does one specific augmentation and checks for properties of the result
194: (4)                """
195: (4)                if orig_k is None:
196: (8)                    try:
197: (12)                       orig_k = nx.edge_connectivity(G)
198: (8)                    except nx.NetworkXPointlessConcept:
199: (12)                       orig_k = 0
200: (4)                info = {}
201: (4)                try:
```

```
(8)                     if avail is not None:
(12)                        avail_dict = dict(zip(*_unpack_available_edges(avail,
weight=weight)))
(8)                     else:
(12)                        avail_dict = None
(8)                     try:
(12)                        generator = nx.k_edge_augmentation(G, k=k, weight=weight,
avail=avail)
(12)                        assert not isinstance(generator, list), "should always return an
iter"
(12)                        aug_edges = []
(12)                        for edge in generator:
(16)                            aug_edges.append(edge)
(8)                     except nx.NetworkXUnfeasible:
(12)                        infeasible = True
(12)                        info["infeasible"] = True
(12)                        assert len(aug_edges) == 0, "should not generate anything if
unfeasible"
(12)                        if avail is None:
(16)                            n_nodes = G.number_of_nodes()
(16)                            assert n_nodes <= k, (
(20)                                "unconstrained cases are only unfeasible if |V| <= k. "
(20)                                f"Got |V|={n_nodes} and k={k}"
(16)                            )
(12)                        else:
(16)                            if max_aug_k is None:
(20)                                G_aug_all = G.copy()
(20)                                G_aug_all.add_edges_from(avail_dict.keys())
(20)                                try:
(24)                                    max_aug_k = nx.edge_connectivity(G_aug_all)
(20)                                except nx.NetworkXPointlessConcept:
(24)                                    max_aug_k = 0
(16)                            assert max_aug_k < k, (
(20)                                "avail should only be unfeasible if using all edges "
(20)                                "does not achieve k-edge-connectivity"
(16)                            )
(12)                        partial_edges = list(
(16)                            nx.k_edge_augmentation(G, k=k, weight=weight, partial=True,
avail=avail)
(12)                        )
(12)                        info["n_partial_edges"] = len(partial_edges)
(12)                        if avail_dict is None:
(16)                            assert set(partial_edges) == set(
(20)                                complement_edges(G)
(16)                            ), "unweighted partial solutions should be the complement"
(12)                        elif len(avail_dict) > 0:
(16)                            H = G.copy()
(16)                            H.add_edges_from(partial_edges)
(16)                            partial_conn = nx.edge_connectivity(H)
(16)                            H.add_edges_from(set(avail_dict.keys()))
(16)                            full_conn = nx.edge_connectivity(H)
(16)                            assert (
(20)                                partial_conn == full_conn
(16)                            ), "adding more edges should not increase k-conn"
(12)                        aug_edges = partial_edges
(8)                     else:
(12)                        infeasible = False
(8)                     num_edges = len(aug_edges)
(8)                     if avail is not None:
(12)                        total_weight = sum(avail_dict[e] for e in aug_edges)
(8)                     else:
(12)                        total_weight = num_edges
(8)                     info["total_weight"] = total_weight
(8)                     info["num_edges"] = num_edges
(8)                     G_aug = G.copy()
(8)                     G_aug.add_edges_from(aug_edges)
(8)                     try:
(12)                        aug_k = nx.edge_connectivity(G_aug)
(8)                     except nx.NetworkXPointlessConcept:
```

```
266: (12)                        aug_k = 0
267: (8)                    info["aug_k"] = aug_k
268: (8)                    if not infeasible and orig_k < k:
269: (12)                        assert info["aug_k"] >= k, f"connectivity should increase to k={k}
or more"
270: (8)                    assert info["aug_k"] >= orig_k, "augmenting should never reduce
connectivity"
271: (8)                    _assert_solution_properties(G, aug_edges, avail_dict)
272: (4)                except Exception:
273: (8)                    info["failed"] = True
274: (8)                    print(f"edges = {list(G.edges())}")
275: (8)                    print(f"nodes = {list(G.nodes())}")
276: (8)                    print(f"aug_edges = {list(aug_edges)}")
277: (8)                    print(f"info  = {info}")
278: (8)                    raise
279: (4)                else:
280: (8)                    if verbose:
281: (12)                       print(f"info  = {info}")
282: (4)                if infeasible:
283: (8)                    aug_edges = None
284: (4)                return aug_edges, info
285: (0)            def _check_augmentations(G, avail=None, max_k=None, weight=None,
verbose=False):
286: (4)                """Helper to check weighted/unweighted cases with multiple values of k"""
287: (4)                try:
288: (8)                    orig_k = nx.edge_connectivity(G)
289: (4)                except nx.NetworkXPointlessConcept:
290: (8)                    orig_k = 0
291: (4)                if avail is not None:
292: (8)                    all_aug_edges = _unpack_available_edges(avail, weight=weight)[0]
293: (8)                    G_aug_all = G.copy()
294: (8)                    G_aug_all.add_edges_from(all_aug_edges)
295: (8)                    try:
296: (12)                       max_aug_k = nx.edge_connectivity(G_aug_all)
297: (8)                    except nx.NetworkXPointlessConcept:
298: (12)                       max_aug_k = 0
299: (4)                else:
300: (8)                    max_aug_k = G.number_of_nodes() - 1
301: (4)                if max_k is None:
302: (8)                    max_k = min(4, max_aug_k)
303: (4)                avail_uniform = {e: 1 for e in complement_edges(G)}
304: (4)                if verbose:
305: (8)                    print("\n=== CHECK_AUGMENTATION ===")
306: (8)                    print(f"G.number_of_nodes = {G.number_of_nodes()!r}")
307: (8)                    print(f"G.number_of_edges = {G.number_of_edges()!r}")
308: (8)                    print(f"max_k = {max_k!r}")
309: (8)                    print(f"max_aug_k = {max_aug_k!r}")
310: (8)                    print(f"orig_k = {orig_k!r}")
311: (4)                for k in range(1, max_k + 1):
312: (8)                    if verbose:
313: (12)                       print("---------------")
314: (12)                       print(f"Checking k = {k}")
315: (8)                    if verbose:
316: (12)                       print("unweighted case")
317: (8)                    aug_edges1, info1 = _augment_and_check(G, k=k, verbose=verbose,
orig_k=orig_k)
318: (8)                    if verbose:
319: (12)                       print("weighted uniform case")
320: (8)                    aug_edges2, info2 = _augment_and_check(
321: (12)                       G,
322: (12)                       k=k,
323: (12)                       avail=avail_uniform,
324: (12)                       verbose=verbose,
325: (12)                       orig_k=orig_k,
326: (12)                       max_aug_k=G.number_of_nodes() - 1,
327: (8)                    )
328: (8)                    if avail is not None:
329: (12)                       if verbose:
330: (16)                           print("weighted case")
```

```
331: (12)                        aug_edges3, info3 = _augment_and_check(
332: (16)                            G,
333: (16)                            k=k,
334: (16)                            avail=avail,
335: (16)                            weight=weight,
336: (16)                            verbose=verbose,
337: (16)                            max_aug_k=max_aug_k,
338: (16)                            orig_k=orig_k,
339: (12)                        )
340: (8)                    if aug_edges1 is not None:
341: (12)                        if k == 1:
342: (16)                            assert info2["total_weight"] == info1["total_weight"]
343: (12)                        if k == 2:
344: (16)                            if orig_k == 0:
345: (20)                                assert info2["total_weight"] <= info1["total_weight"] * 3
346: (16)                            else:
347: (20)                                assert info2["total_weight"] <= info1["total_weight"] * 2
348: (16)                            _check_unconstrained_bridge_property(G, info1)
349: (0)            def _check_unconstrained_bridge_property(G, info1):
350: (4)                import math
351: (4)                bridge_ccs = list(nx.connectivity.bridge_components(G))
352: (4)                C = collapse(G, bridge_ccs)
353: (4)                p = len([n for n, d in C.degree() if d == 1])  # leafs
354: (4)                q = len([n for n, d in C.degree() if d == 0])  # isolated
355: (4)                if p + q > 1:
356: (8)                    size_target = math.ceil(p / 2) + q
357: (8)                    size_aug = info1["num_edges"]
358: (8)                    assert (
359: (12)                       size_aug == size_target
360: (8)                    ), "augmentation size is different from what theory predicts"
```

File 232 - test_edge_kcomponents.py:

```
1: (0)              import itertools as it
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.connectivity import EdgeComponentAuxGraph,
bridge_components
5: (0)              from networkx.algorithms.connectivity.edge_kcomponents import
general_k_edge_subgraphs
6: (0)              from networkx.utils import pairwise
7: (0)              def fset(list_of_sets):
8: (4)                  """allows == to be used for list of sets"""
9: (4)                  return set(map(frozenset, list_of_sets))
10: (0)             def _assert_subgraph_edge_connectivity(G, ccs_subgraph, k):
11: (4)                 """
12: (4)                 tests properties of k-edge-connected subgraphs
13: (4)                 the actual edge connectivity should be no less than k unless the cc is a
14: (4)                 single node.
15: (4)                 """
16: (4)                 for cc in ccs_subgraph:
17: (8)                     C = G.subgraph(cc)
18: (8)                     if len(cc) > 1:
19: (12)                        connectivity = nx.edge_connectivity(C)
20: (12)                        assert connectivity >= k
21: (0)             def _memo_connectivity(G, u, v, memo):
22: (4)                 edge = (u, v)
23: (4)                 if edge in memo:
24: (8)                     return memo[edge]
25: (4)                 if not G.is_directed():
26: (8)                     redge = (v, u)
27: (8)                     if redge in memo:
28: (12)                        return memo[redge]
29: (4)                 memo[edge] = nx.edge_connectivity(G, *edge)
30: (4)                 return memo[edge]
31: (0)             def _all_pairs_connectivity(G, cc, k, memo):
32: (4)                 for u, v in it.combinations(cc, 2):
```

```
 33: (8)                    connectivity = _memo_connectivity(G, u, v, memo)
 34: (8)                    if G.is_directed():
 35: (12)                       connectivity = min(connectivity, _memo_connectivity(G, v, u,
memo))
 36: (8)                    assert connectivity >= k
 37: (0)            def _assert_local_cc_edge_connectivity(G, ccs_local, k, memo):
 38: (4)                """
 39: (4)                tests properties of k-edge-connected components
 40: (4)                the local edge connectivity between each pair of nodes in the original
 41: (4)                graph should be no less than k unless the cc is a single node.
 42: (4)                """
 43: (4)                for cc in ccs_local:
 44: (8)                    if len(cc) > 1:
 45: (12)                       C = G.subgraph(cc)
 46: (12)                       connectivity = nx.edge_connectivity(C)
 47: (12)                       if connectivity < k:
 48: (16)                           _all_pairs_connectivity(G, cc, k, memo)
 49: (0)            def _check_edge_connectivity(G):
 50: (4)                """
 51: (4)                Helper - generates all k-edge-components using the aux graph.  Checks the
 52: (4)                both local and subgraph edge connectivity of each cc. Also checks that
 53: (4)                alternate methods of computing the k-edge-ccs generate the same result.
 54: (4)                """
 55: (4)                aux_graph = EdgeComponentAuxGraph.construct(G)
 56: (4)                memo = {}
 57: (4)                for k in it.count(1):
 58: (8)                    ccs_local = fset(aux_graph.k_edge_components(k))
 59: (8)                    ccs_subgraph = fset(aux_graph.k_edge_subgraphs(k))
 60: (8)                    _assert_local_cc_edge_connectivity(G, ccs_local, k, memo)
 61: (8)                    _assert_subgraph_edge_connectivity(G, ccs_subgraph, k)
 62: (8)                    if k == 1 or k == 2 and not G.is_directed():
 63: (12)                       assert (
 64: (16)                           ccs_local == ccs_subgraph
 65: (12)                       ), "Subgraphs and components should be the same when k == 1 or (k
== 2 and not G.directed())"
 66: (8)                    if G.is_directed():
 67: (12)                       if k == 1:
 68: (16)                           alt_sccs = fset(nx.strongly_connected_components(G))
 69: (16)                           assert alt_sccs == ccs_local, "k=1 failed alt"
 70: (16)                           assert alt_sccs == ccs_subgraph, "k=1 failed alt"
 71: (8)                    else:
 72: (12)                       if k == 1:
 73: (16)                           alt_ccs = fset(nx.connected_components(G))
 74: (16)                           assert alt_ccs == ccs_local, "k=1 failed alt"
 75: (16)                           assert alt_ccs == ccs_subgraph, "k=1 failed alt"
 76: (12)                       elif k == 2:
 77: (16)                           alt_bridge_ccs = fset(bridge_components(G))
 78: (16)                           assert alt_bridge_ccs == ccs_local, "k=2 failed alt"
 79: (16)                           assert alt_bridge_ccs == ccs_subgraph, "k=2 failed alt"
 80: (8)                    alt_subgraph_ccs = fset(
 81: (12)                       [set(C.nodes()) for C in general_k_edge_subgraphs(G, k=k)]
 82: (8)                    )
 83: (8)                    assert alt_subgraph_ccs == ccs_subgraph, "alt subgraph method failed"
 84: (8)                    if k > 2 and all(len(cc) == 1 for cc in ccs_local):
 85: (12)                       break
 86: (0)            def test_zero_k_exception():
 87: (4)                G = nx.Graph()
 88: (4)                pytest.raises(ValueError, nx.k_edge_components, G, k=0)
 89: (4)                pytest.raises(ValueError, nx.k_edge_subgraphs, G, k=0)
 90: (4)                aux_graph = EdgeComponentAuxGraph.construct(G)
 91: (4)                pytest.raises(ValueError, list, aux_graph.k_edge_components(k=0))
 92: (4)                pytest.raises(ValueError, list, aux_graph.k_edge_subgraphs(k=0))
 93: (4)                pytest.raises(ValueError, list, general_k_edge_subgraphs(G, k=0))
 94: (0)            def test_empty_input():
 95: (4)                G = nx.Graph()
 96: (4)                assert [] == list(nx.k_edge_components(G, k=5))
 97: (4)                assert [] == list(nx.k_edge_subgraphs(G, k=5))
 98: (4)                G = nx.DiGraph()
 99: (4)                assert [] == list(nx.k_edge_components(G, k=5))
```

```
(4)                 assert [] == list(nx.k_edge_subgraphs(G, k=5))
(0)             def test_not_implemented():
(4)                 G = nx.MultiGraph()
(4)                 pytest.raises(nx.NetworkXNotImplemented, EdgeComponentAuxGraph.construct, G)
(4)                 pytest.raises(nx.NetworkXNotImplemented, nx.k_edge_components, G, k=2)
(4)                 pytest.raises(nx.NetworkXNotImplemented, nx.k_edge_subgraphs, G, k=2)
(4)                 with pytest.raises(nx.NetworkXNotImplemented):
(8)                     next(bridge_components(G))
(4)                 with pytest.raises(nx.NetworkXNotImplemented):
(8)                     next(bridge_components(nx.DiGraph()))
(0)             def test_general_k_edge_subgraph_quick_return():
(4)                 G = nx.Graph()
(4)                 G.add_node(0)
(4)                 subgraphs = list(general_k_edge_subgraphs(G, k=1))
(4)                 assert len(subgraphs) == 1
(4)                 for subgraph in subgraphs:
(8)                     assert subgraph.number_of_nodes() == 1
(4)                 G.add_node(1)
(4)                 subgraphs = list(general_k_edge_subgraphs(G, k=1))
(4)                 assert len(subgraphs) == 2
(4)                 for subgraph in subgraphs:
(8)                     assert subgraph.number_of_nodes() == 1
(0)             def test_random_gnp():
(4)                 seeds = [12, 13]
(4)                 for seed in seeds:
(8)                     G = nx.gnp_random_graph(20, 0.2, seed=seed)
(8)                     _check_edge_connectivity(G)
(0)             def test_configuration():
(4)                 seeds = [14, 15]
(4)                 for seed in seeds:
(8)                     deg_seq = nx.random_powerlaw_tree_sequence(20, seed=seed, tries=5000)
(8)                     G = nx.Graph(nx.configuration_model(deg_seq, seed=seed))
(8)                     G.remove_edges_from(nx.selfloop_edges(G))
(8)                     _check_edge_connectivity(G)
(0)             def test_shell():
(4)                 seeds = [20]
(4)                 for seed in seeds:
(8)                     constructor = [(12, 70, 0.8), (15, 40, 0.6)]
(8)                     G = nx.random_shell_graph(constructor, seed=seed)
(8)                     _check_edge_connectivity(G)
(0)             def test_karate():
(4)                 G = nx.karate_club_graph()
(4)                 _check_edge_connectivity(G)
(0)             def test_tarjan_bridge():
(4)                 ccs = [
(8)                     (1, 2, 4, 3, 1, 4),
(8)                     (5, 6, 7, 5),
(8)                     (8, 9, 10, 8),
(8)                     (17, 18, 16, 15, 17),
(8)                     (11, 12, 14, 13, 11, 14),
(4)                 ]
(4)                 bridges = [(4, 8), (3, 5), (3, 17)]
(4)                 G = nx.Graph(it.chain(*(pairwise(path) for path in ccs + bridges)))
(4)                 _check_edge_connectivity(G)
(0)             def test_bridge_cc():
(4)                 cc2 = [(1, 2, 4, 3, 1, 4), (8, 9, 10, 8), (11, 12, 13, 11)]
(4)                 bridges = [(4, 8), (3, 5), (20, 21), (22, 23, 24)]
(4)                 G = nx.Graph(it.chain(*(pairwise(path) for path in cc2 + bridges)))
(4)                 bridge_ccs = fset(bridge_components(G))
(4)                 target_ccs = fset(
(8)                     [{1, 2, 3, 4}, {5}, {8, 9, 10}, {11, 12, 13}, {20}, {21}, {22}, {23}, {24}]
(4)                 )
(4)                 assert bridge_ccs == target_ccs
(4)                 _check_edge_connectivity(G)
(0)             def test_undirected_aux_graph():
(4)                 a, b, c, d, e, f, g, h, i = "abcdefghi"
(4)                 paths = [
```

```
167: (8)                    (a, d, b, f, c),
168: (8)                    (a, e, b),
169: (8)                    (a, e, b, c, g, b, a),
170: (8)                    (c, b),
171: (8)                    (f, g, f),
172: (8)                    (h, i),
173: (4)                ]
174: (4)                G = nx.Graph(it.chain(*[pairwise(path) for path in paths]))
175: (4)                aux_graph = EdgeComponentAuxGraph.construct(G)
176: (4)                components_1 = fset(aux_graph.k_edge_subgraphs(k=1))
177: (4)                target_1 = fset([{a, b, c, d, e, f, g}, {h, i}])
178: (4)                assert target_1 == components_1
179: (4)                alt_1 = fset(nx.k_edge_subgraphs(G, k=1))
180: (4)                assert alt_1 == components_1
181: (4)                components_2 = fset(aux_graph.k_edge_subgraphs(k=2))
182: (4)                target_2 = fset([{a, b, c, d, e, f, g}, {h}, {i}])
183: (4)                assert target_2 == components_2
184: (4)                alt_2 = fset(nx.k_edge_subgraphs(G, k=2))
185: (4)                assert alt_2 == components_2
186: (4)                components_3 = fset(aux_graph.k_edge_subgraphs(k=3))
187: (4)                target_3 = fset([{a}, {b, c, f, g}, {d}, {e}, {h}, {i}])
188: (4)                assert target_3 == components_3
189: (4)                components_4 = fset(aux_graph.k_edge_subgraphs(k=4))
190: (4)                target_4 = fset([{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}])
191: (4)                assert target_4 == components_4
192: (4)                _check_edge_connectivity(G)
193: (0)            def test_local_subgraph_difference():
194: (4)                paths = [
195: (8)                    (11, 12, 13, 14, 11, 13, 14, 12),  # first 4-clique
196: (8)                    (21, 22, 23, 24, 21, 23, 24, 22),  # second 4-clique
197: (8)                    (11, 101, 21),
198: (8)                    (12, 102, 22),
199: (8)                    (13, 103, 23),
200: (8)                    (14, 104, 24),
201: (4)                ]
202: (4)                G = nx.Graph(it.chain(*[pairwise(path) for path in paths]))
203: (4)                aux_graph = EdgeComponentAuxGraph.construct(G)
204: (4)                subgraph_ccs = fset(aux_graph.k_edge_subgraphs(3))
205: (4)                subgraph_target = fset(
206: (8)                    [{101}, {102}, {103}, {104}, {21, 22, 23, 24}, {11, 12, 13, 14}]
207: (4)                )
208: (4)                assert subgraph_ccs == subgraph_target
209: (4)                local_ccs = fset(aux_graph.k_edge_components(3))
210: (4)                local_target = fset([{101}, {102}, {103}, {104}, {11, 12, 13, 14, 21, 22,
23, 24}])
211: (4)                assert local_ccs == local_target
212: (0)            def test_local_subgraph_difference_directed():
213: (4)                dipaths = [(1, 2, 3, 4, 1), (1, 3, 1)]
214: (4)                G = nx.DiGraph(it.chain(*[pairwise(path) for path in dipaths]))
215: (4)                assert fset(nx.k_edge_components(G, k=1)) == fset(nx.k_edge_subgraphs(G,
k=1))
216: (4)                assert fset(nx.k_edge_components(G, k=2)) != fset(nx.k_edge_subgraphs(G,
k=2))
217: (4)                assert fset(nx.k_edge_components(G, k=3)) == fset(nx.k_edge_subgraphs(G,
k=3))
218: (4)                _check_edge_connectivity(G)
219: (0)            def test_triangles():
220: (4)                paths = [
221: (8)                    (11, 12, 13, 11),  # first 3-clique
222: (8)                    (21, 22, 23, 21),  # second 3-clique
223: (8)                    (11, 21),  # connected by an edge
224: (4)                ]
225: (4)                G = nx.Graph(it.chain(*[pairwise(path) for path in paths]))
226: (4)                assert fset(nx.k_edge_components(G, k=1)) == fset(nx.k_edge_subgraphs(G,
k=1))
227: (4)                assert fset(nx.k_edge_components(G, k=2)) == fset(nx.k_edge_subgraphs(G,
k=2))
228: (4)                assert fset(nx.k_edge_components(G, k=3)) == fset(nx.k_edge_subgraphs(G,
k=3))
```

```
(4)                 _check_edge_connectivity(G)
(0)             def test_four_clique():
(4)                 paths = [
(8)                     (11, 12, 13, 14, 11, 13, 14, 12),  # first 4-clique
(8)                     (21, 22, 23, 24, 21, 23, 24, 22),  # second 4-clique
(8)                     (100, 13),
(8)                     (12, 100, 22),
(8)                     (13, 200, 23),
(8)                     (14, 300, 24),
(4)                 ]
(4)                 G = nx.Graph(it.chain(*[pairwise(path) for path in paths]))
(4)                 local_ccs = fset(nx.k_edge_components(G, k=3))
(4)                 subgraphs = fset(nx.k_edge_subgraphs(G, k=3))
(4)                 assert local_ccs != subgraphs
(4)                 clique1 = frozenset(paths[0])
(4)                 clique2 = frozenset(paths[1])
(4)                 assert clique1.union(clique2).union({100}) in local_ccs
(4)                 assert clique1 in subgraphs
(4)                 assert clique2 in subgraphs
(4)                 assert G.degree(100) == 3
(4)                 _check_edge_connectivity(G)
(0)             def test_five_clique():
(4)                 G = nx.disjoint_union(nx.complete_graph(5), nx.complete_graph(5))
(4)                 paths = [
(8)                     (1, 100, 6),
(8)                     (2, 100, 7),
(8)                     (3, 200, 8),
(8)                     (4, 200, 100),
(4)                 ]
(4)                 G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
(4)                 assert min(dict(nx.degree(G)).values()) == 4
(4)                 assert fset(nx.k_edge_components(G, k=3)) == fset(nx.k_edge_subgraphs(G,
k=3))
(4)                 assert fset(nx.k_edge_components(G, k=4)) != fset(nx.k_edge_subgraphs(G,
k=4))
(4)                 assert fset(nx.k_edge_components(G, k=5)) != fset(nx.k_edge_subgraphs(G,
k=5))
(4)                 assert fset(nx.k_edge_components(G, k=6)) == fset(nx.k_edge_subgraphs(G,
k=6))
(4)                 _check_edge_connectivity(G)
(0)             def test_directed_aux_graph():
(4)                 a, b, c, d, e, f, g, h, i = "abcdefghi"
(4)                 dipaths = [
(8)                     (a, d, b, f, c),
(8)                     (a, e, b),
(8)                     (a, e, b, c, g, b, a),
(8)                     (c, b),
(8)                     (f, g, f),
(8)                     (h, i),
(4)                 ]
(4)                 G = nx.DiGraph(it.chain(*[pairwise(path) for path in dipaths]))
(4)                 aux_graph = EdgeComponentAuxGraph.construct(G)
(4)                 components_1 = fset(aux_graph.k_edge_subgraphs(k=1))
(4)                 target_1 = fset([{a, b, c, d, e, f, g}, {h}, {i}])
(4)                 assert target_1 == components_1
(4)                 alt_1 = fset(nx.strongly_connected_components(G))
(4)                 assert alt_1 == components_1
(4)                 components_2 = fset(aux_graph.k_edge_subgraphs(k=2))
(4)                 target_2 = fset([{i}, {e}, {d}, {b, c, f, g}, {h}, {a}])
(4)                 assert target_2 == components_2
(4)                 components_3 = fset(aux_graph.k_edge_subgraphs(k=3))
(4)                 target_3 = fset([{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}])
(4)                 assert target_3 == components_3
(0)             def test_random_gnp_directed():
(4)                 seeds = [21]
(4)                 for seed in seeds:
(8)                     G = nx.gnp_random_graph(20, 0.2, directed=True, seed=seed)
(8)                     _check_edge_connectivity(G)
(0)             def test_configuration_directed():
```

```
294: (4)                seeds = [67]
295: (4)                for seed in seeds:
296: (8)                    deg_seq = nx.random_powerlaw_tree_sequence(20, seed=seed, tries=5000)
297: (8)                    G = nx.DiGraph(nx.configuration_model(deg_seq, seed=seed))
298: (8)                    G.remove_edges_from(nx.selfloop_edges(G))
299: (8)                    _check_edge_connectivity(G)
300: (0)            def test_shell_directed():
301: (4)                seeds = [31]
302: (4)                for seed in seeds:
303: (8)                    constructor = [(12, 70, 0.8), (15, 40, 0.6)]
304: (8)                    G = nx.random_shell_graph(constructor, seed=seed).to_directed()
305: (8)                    _check_edge_connectivity(G)
306: (0)            def test_karate_directed():
307: (4)                G = nx.karate_club_graph().to_directed()
308: (4)                _check_edge_connectivity(G)
```

File 233 - test_kcomponents.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.connectivity.kcomponents import (
4: (4)                  _consolidate,
5: (4)                  build_k_number_dict,
6: (0)              )
7: (0)              def torrents_and_ferraro_graph():
8: (4)                  G = nx.convert_node_labels_to_integers(
9: (8)                      nx.grid_graph([5, 5]), label_attribute="labels"
10: (4)                 )
11: (4)                 rlabels = nx.get_node_attributes(G, "labels")
12: (4)                 labels = {v: k for k, v in rlabels.items()}
13: (4)                 for nodes in [(labels[(0, 4)], labels[(1, 4)]), (labels[(3, 4)],
labels[(4, 4)])]:
14: (8)                     new_node = G.order() + 1
15: (8)                     P = nx.petersen_graph()
16: (8)                     G = nx.disjoint_union(G, P)
17: (8)                     G.add_edge(new_node + 1, nodes[0])
18: (8)                     G.add_edge(new_node, nodes[1])
19: (8)                     K = nx.complete_graph(5)
20: (8)                     G = nx.disjoint_union(G, K)
21: (8)                     G.add_edge(new_node + 2, new_node + 11)
22: (8)                     G.add_edge(new_node + 3, new_node + 12)
23: (8)                     G.add_edge(new_node + 4, new_node + 13)
24: (8)                     G = nx.disjoint_union(G, K)
25: (8)                     nbrs = G[new_node + 10]
26: (8)                     G.remove_node(new_node + 10)
27: (8)                     for nbr in nbrs:
28: (12)                        G.add_edge(new_node + 17, nbr)
29: (8)                     G.add_edge(new_node + 16, new_node + 8)
30: (4)                 for nodes in [(labels[(0, 0)], labels[(1, 0)]), (labels[(3, 0)],
labels[(4, 0)])]:
31: (8)                     new_node = G.order() + 1
32: (8)                     P = nx.petersen_graph()
33: (8)                     G = nx.disjoint_union(G, P)
34: (8)                     G.add_edge(new_node + 1, nodes[0])
35: (8)                     G.add_edge(new_node, nodes[1])
36: (8)                     K = nx.complete_graph(5)
37: (8)                     G = nx.disjoint_union(G, K)
38: (8)                     G.add_edge(new_node + 2, new_node + 11)
39: (8)                     G.add_edge(new_node + 3, new_node + 12)
40: (8)                     G.add_edge(new_node + 4, new_node + 13)
41: (8)                     G = nx.disjoint_union(G, K)
42: (8)                     nbrs = G[new_node + 10]
43: (8)                     G.remove_node(new_node + 10)
44: (8)                     for nbr in nbrs:
45: (12)                        G.add_edge(new_node + 17, nbr)
46: (8)                     nbrs2 = G[new_node + 9]
47: (8)                     G.remove_node(new_node + 9)
```

```
48: (8)                     for nbr in nbrs2:
49: (12)                        G.add_edge(new_node + 18, nbr)
50: (4)                 return G
51: (0)             def test_directed():
52: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
53: (8)                     G = nx.gnp_random_graph(10, 0.2, directed=True, seed=42)
54: (8)                     nx.k_components(G)
55: (0)             def _check_connectivity(G, k_components):
56: (4)                 for k, components in k_components.items():
57: (8)                     if k < 3:
58: (12)                        continue
59: (8)                     for component in components:
60: (12)                        C = G.subgraph(component)
61: (12)                        K = nx.node_connectivity(C)
62: (12)                        assert K >= k
63: (0)             @pytest.mark.slow
64: (0)             def test_torrents_and_ferraro_graph():
65: (4)                 G = torrents_and_ferraro_graph()
66: (4)                 result = nx.k_components(G)
67: (4)                 _check_connectivity(G, result)
68: (4)                 assert len(result[3]) == 8
69: (4)                 assert len([c for c in result[3] if len(c) == 15]) == 4
70: (4)                 assert len([c for c in result[3] if len(c) == 5]) == 4
71: (4)                 assert len(result[4]) == 8
72: (4)                 assert all(len(c) == 5 for c in result[4])
73: (0)             @pytest.mark.slow
74: (0)             def test_random_gnp():
75: (4)                 G = nx.gnp_random_graph(50, 0.2, seed=42)
76: (4)                 result = nx.k_components(G)
77: (4)                 _check_connectivity(G, result)
78: (0)             @pytest.mark.slow
79: (0)             def test_shell():
80: (4)                 constructor = [(20, 80, 0.8), (80, 180, 0.6)]
81: (4)                 G = nx.random_shell_graph(constructor, seed=42)
82: (4)                 result = nx.k_components(G)
83: (4)                 _check_connectivity(G, result)
84: (0)             def test_configuration():
85: (4)                 deg_seq = nx.random_powerlaw_tree_sequence(100, tries=5, seed=72)
86: (4)                 G = nx.Graph(nx.configuration_model(deg_seq))
87: (4)                 G.remove_edges_from(nx.selfloop_edges(G))
88: (4)                 result = nx.k_components(G)
89: (4)                 _check_connectivity(G, result)
90: (0)             def test_karate():
91: (4)                 G = nx.karate_club_graph()
92: (4)                 result = nx.k_components(G)
93: (4)                 _check_connectivity(G, result)
94: (0)             def test_karate_component_number():
95: (4)                 karate_k_num = {
96: (8)                     0: 4,
97: (8)                     1: 4,
98: (8)                     2: 4,
99: (8)                     3: 4,
100: (8)                    4: 3,
101: (8)                    5: 3,
102: (8)                    6: 3,
103: (8)                    7: 4,
104: (8)                    8: 4,
105: (8)                    9: 2,
106: (8)                    10: 3,
107: (8)                    11: 1,
108: (8)                    12: 2,
109: (8)                    13: 4,
110: (8)                    14: 2,
111: (8)                    15: 2,
112: (8)                    16: 2,
113: (8)                    17: 2,
114: (8)                    18: 2,
115: (8)                    19: 3,
116: (8)                    20: 2,
```

```
(8)                     21: 2,
(8)                     22: 2,
(8)                     23: 3,
(8)                     24: 3,
(8)                     25: 3,
(8)                     26: 2,
(8)                     27: 3,
(8)                     28: 3,
(8)                     29: 3,
(8)                     30: 4,
(8)                     31: 3,
(8)                     32: 4,
(8)                     33: 4,
(4)                 }
(4)                 G = nx.karate_club_graph()
(4)                 k_components = nx.k_components(G)
(4)                 k_num = build_k_number_dict(k_components)
(4)                 assert karate_k_num == k_num
(0)             def test_davis_southern_women():
(4)                 G = nx.davis_southern_women_graph()
(4)                 result = nx.k_components(G)
(4)                 _check_connectivity(G, result)
(0)             def test_davis_southern_women_detail_3_and_4():
(4)                 solution = {
(8)                     3: [
(12)                        {
(16)                            "Nora Fayette",
(16)                            "E10",
(16)                            "Myra Liddel",
(16)                            "E12",
(16)                            "E14",
(16)                            "Frances Anderson",
(16)                            "Evelyn Jefferson",
(16)                            "Ruth DeSand",
(16)                            "Helen Lloyd",
(16)                            "Eleanor Nye",
(16)                            "E9",
(16)                            "E8",
(16)                            "E5",
(16)                            "E4",
(16)                            "E7",
(16)                            "E6",
(16)                            "E1",
(16)                            "Verne Sanderson",
(16)                            "E3",
(16)                            "E2",
(16)                            "Theresa Anderson",
(16)                            "Pearl Oglethorpe",
(16)                            "Katherina Rogers",
(16)                            "Brenda Rogers",
(16)                            "E13",
(16)                            "Charlotte McDowd",
(16)                            "Sylvia Avondale",
(16)                            "Laura Mandeville",
(12)                        }
(8)                     ],
(8)                     4: [
(12)                        {
(16)                            "Nora Fayette",
(16)                            "E10",
(16)                            "Verne Sanderson",
(16)                            "E12",
(16)                            "Frances Anderson",
(16)                            "Evelyn Jefferson",
(16)                            "Ruth DeSand",
(16)                            "Helen Lloyd",
(16)                            "Eleanor Nye",
(16)                            "E9",
(16)                            "E8",
```

```
186: (16)                            "E5",
187: (16)                            "E4",
188: (16)                            "E7",
189: (16)                            "E6",
190: (16)                            "Myra Liddel",
191: (16)                            "E3",
192: (16)                            "Theresa Anderson",
193: (16)                            "Katherina Rogers",
194: (16)                            "Brenda Rogers",
195: (16)                            "Charlotte McDowd",
196: (16)                            "Sylvia Avondale",
197: (16)                            "Laura Mandeville",
198: (12)                        }
199: (8)                    ],
200: (4)                }
201: (4)                G = nx.davis_southern_women_graph()
202: (4)                result = nx.k_components(G)
203: (4)                for k, components in result.items():
204: (8)                    if k < 3:
205: (12)                       continue
206: (8)                    assert len(components) == len(solution[k])
207: (8)                    for component in components:
208: (12)                       assert component in solution[k]
209: (0)            def test_set_consolidation_rosettacode():
210: (4)                def list_of_sets_equal(result, solution):
211: (8)                    assert {frozenset(s) for s in result} == {frozenset(s) for s in
solution}
212: (4)                question = [{"A", "B"}, {"C", "D"}]
213: (4)                solution = [{"A", "B"}, {"C", "D"}]
214: (4)                list_of_sets_equal(_consolidate(question, 1), solution)
215: (4)                question = [{"A", "B"}, {"B", "C"}]
216: (4)                solution = [{"A", "B", "C"}]
217: (4)                list_of_sets_equal(_consolidate(question, 1), solution)
218: (4)                question = [{"A", "B"}, {"C", "D"}, {"D", "B"}]
219: (4)                solution = [{"A", "C", "B", "D"}]
220: (4)                list_of_sets_equal(_consolidate(question, 1), solution)
221: (4)                question = [{"H", "I", "K"}, {"A", "B"}, {"C", "D"}, {"D", "B"}, {"F",
"G", "H"}]
222: (4)                solution = [{"A", "C", "B", "D"}, {"G", "F", "I", "H", "K"}]
223: (4)                list_of_sets_equal(_consolidate(question, 1), solution)
224: (4)                question = [
225: (8)                    {"A", "H"},
226: (8)                    {"H", "I", "K"},
227: (8)                    {"A", "B"},
228: (8)                    {"C", "D"},
229: (8)                    {"D", "B"},
230: (8)                    {"F", "G", "H"},
231: (4)                ]
232: (4)                solution = [{"A", "C", "B", "D", "G", "F", "I", "H", "K"}]
233: (4)                list_of_sets_equal(_consolidate(question, 1), solution)
234: (4)                question = [
235: (8)                    {"H", "I", "K"},
236: (8)                    {"A", "B"},
237: (8)                    {"C", "D"},
238: (8)                    {"D", "B"},
239: (8)                    {"F", "G", "H"},
240: (8)                    {"A", "H"},
241: (4)                ]
242: (4)                solution = [{"A", "C", "B", "D", "G", "F", "I", "H", "K"}]
243: (4)                list_of_sets_equal(_consolidate(question, 1), solution)

----------------------------------------

File 234 - test_kcutsets.py:

1: (0)              import itertools
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms import flow
```

```
5: (0)              from networkx.algorithms.connectivity.kcutsets import _is_separating_set
6: (0)              MAX_CUTSETS_TO_TEST = 4  # originally 100. cut to decrease testing time
7: (0)              flow_funcs = [
8: (4)                  flow.boykov_kolmogorov,
9: (4)                  flow.dinitz,
10: (4)                 flow.edmonds_karp,
11: (4)                 flow.preflow_push,
12: (4)                 flow.shortest_augmenting_path,
13: (0)             ]
14: (0)             def graph_example_1():
15: (4)                 G = nx.convert_node_labels_to_integers(
16: (8)                     nx.grid_graph([5, 5]), label_attribute="labels"
17: (4)                 )
18: (4)                 rlabels = nx.get_node_attributes(G, "labels")
19: (4)                 labels = {v: k for k, v in rlabels.items()}
20: (4)                 for nodes in [
21: (8)                     (labels[(0, 0)], labels[(1, 0)]),
22: (8)                     (labels[(0, 4)], labels[(1, 4)]),
23: (8)                     (labels[(3, 0)], labels[(4, 0)]),
24: (8)                     (labels[(3, 4)], labels[(4, 4)]),
25: (4)                 ]:
26: (8)                     new_node = G.order() + 1
27: (8)                     P = nx.petersen_graph()
28: (8)                     G = nx.disjoint_union(G, P)
29: (8)                     G.add_edge(new_node + 1, nodes[0])
30: (8)                     G.add_edge(new_node, nodes[1])
31: (8)                     K = nx.complete_graph(5)
32: (8)                     G = nx.disjoint_union(G, K)
33: (8)                     G.add_edge(new_node + 2, new_node + 11)
34: (8)                     G.add_edge(new_node + 3, new_node + 12)
35: (8)                     G.add_edge(new_node + 4, new_node + 13)
36: (8)                     G = nx.disjoint_union(G, K)
37: (8)                     nbrs = G[new_node + 10]
38: (8)                     G.remove_node(new_node + 10)
39: (8)                     for nbr in nbrs:
40: (12)                        G.add_edge(new_node + 17, nbr)
41: (8)                     G.add_edge(new_node + 16, new_node + 5)
42: (4)                 return G
43: (0)             def torrents_and_ferraro_graph():
44: (4)                 G = nx.convert_node_labels_to_integers(
45: (8)                     nx.grid_graph([5, 5]), label_attribute="labels"
46: (4)                 )
47: (4)                 rlabels = nx.get_node_attributes(G, "labels")
48: (4)                 labels = {v: k for k, v in rlabels.items()}
49: (4)                 for nodes in [(labels[(0, 4)], labels[(1, 4)]), (labels[(3, 4)],
labels[(4, 4)])]:
50: (8)                     new_node = G.order() + 1
51: (8)                     P = nx.petersen_graph()
52: (8)                     G = nx.disjoint_union(G, P)
53: (8)                     G.add_edge(new_node + 1, nodes[0])
54: (8)                     G.add_edge(new_node, nodes[1])
55: (8)                     K = nx.complete_graph(5)
56: (8)                     G = nx.disjoint_union(G, K)
57: (8)                     G.add_edge(new_node + 2, new_node + 11)
58: (8)                     G.add_edge(new_node + 3, new_node + 12)
59: (8)                     G.add_edge(new_node + 4, new_node + 13)
60: (8)                     G = nx.disjoint_union(G, K)
61: (8)                     nbrs = G[new_node + 10]
62: (8)                     G.remove_node(new_node + 10)
63: (8)                     for nbr in nbrs:
64: (12)                        G.add_edge(new_node + 17, nbr)
65: (8)                     G.add_edge(new_node + 16, new_node + 8)
66: (4)                 for nodes in [(labels[(0, 0)], labels[(1, 0)]), (labels[(3, 0)],
labels[(4, 0)])]:
67: (8)                     new_node = G.order() + 1
68: (8)                     P = nx.petersen_graph()
69: (8)                     G = nx.disjoint_union(G, P)
70: (8)                     G.add_edge(new_node + 1, nodes[0])
71: (8)                     G.add_edge(new_node, nodes[1])
```

```
(8)                     K = nx.complete_graph(5)
(8)                     G = nx.disjoint_union(G, K)
(8)                     G.add_edge(new_node + 2, new_node + 11)
(8)                     G.add_edge(new_node + 3, new_node + 12)
(8)                     G.add_edge(new_node + 4, new_node + 13)
(8)                     G = nx.disjoint_union(G, K)
(8)                     nbrs = G[new_node + 10]
(8)                     G.remove_node(new_node + 10)
(8)                     for nbr in nbrs:
(12)                        G.add_edge(new_node + 17, nbr)
(8)                     nbrs2 = G[new_node + 9]
(8)                     G.remove_node(new_node + 9)
(8)                     for nbr in nbrs2:
(12)                        G.add_edge(new_node + 18, nbr)
(4)                 return G
(0)             def _check_separating_sets(G):
(4)                 for cc in nx.connected_components(G):
(8)                     if len(cc) < 3:
(12)                        continue
(8)                     Gc = G.subgraph(cc)
(8)                     node_conn = nx.node_connectivity(Gc)
(8)                     all_cuts = nx.all_node_cuts(Gc)
(8)                     for cut in itertools.islice(all_cuts, MAX_CUTSETS_TO_TEST):
(12)                        assert node_conn == len(cut)
(12)                        assert not nx.is_connected(nx.restricted_view(G, cut, []))
(0)             @pytest.mark.slow
(0)             def test_torrents_and_ferraro_graph():
(4)                 G = torrents_and_ferraro_graph()
(4)                 _check_separating_sets(G)
(0)             def test_example_1():
(4)                 G = graph_example_1()
(4)                 _check_separating_sets(G)
(0)             def test_random_gnp():
(4)                 G = nx.gnp_random_graph(100, 0.1, seed=42)
(4)                 _check_separating_sets(G)
(0)             def test_shell():
(4)                 constructor = [(20, 80, 0.8), (80, 180, 0.6)]
(4)                 G = nx.random_shell_graph(constructor, seed=42)
(4)                 _check_separating_sets(G)
(0)             def test_configuration():
(4)                 deg_seq = nx.random_powerlaw_tree_sequence(100, tries=5, seed=72)
(4)                 G = nx.Graph(nx.configuration_model(deg_seq))
(4)                 G.remove_edges_from(nx.selfloop_edges(G))
(4)                 _check_separating_sets(G)
(0)             def test_karate():
(4)                 G = nx.karate_club_graph()
(4)                 _check_separating_sets(G)
(0)             def _generate_no_biconnected(max_attempts=50):
(4)                 attempts = 0
(4)                 while True:
(8)                     G = nx.fast_gnp_random_graph(100, 0.0575, seed=42)
(8)                     if nx.is_connected(G) and not nx.is_biconnected(G):
(12)                        attempts = 0
(12)                        yield G
(8)                     else:
(12)                        if attempts >= max_attempts:
(16)                            msg = f"Tried {attempts} times: no suitable Graph."
(16)                            raise Exception(msg)
(12)                        else:
(16)                            attempts += 1
(0)             def test_articulation_points():
(4)                 Ggen = _generate_no_biconnected()
(4)                 for i in range(1):  # change 1 to 3 or more for more realizations.
(8)                     G = next(Ggen)
(8)                     articulation_points = [{a} for a in nx.articulation_points(G)]
(8)                     for cut in nx.all_node_cuts(G):
(12)                        assert cut in articulation_points
(0)             def test_grid_2d_graph():
(4)                 G = nx.grid_2d_graph(5, 5)
```

```
141: (4)                solution = [{(0, 1), (1, 0)}, {(3, 0), (4, 1)}, {(3, 4), (4, 3)}, {(0, 3),
(1, 4)}]
142: (4)                for cut in nx.all_node_cuts(G):
143: (8)                    assert cut in solution
144: (0)            def test_disconnected_graph():
145: (4)                G = nx.fast_gnp_random_graph(100, 0.01, seed=42)
146: (4)                cuts = nx.all_node_cuts(G)
147: (4)                pytest.raises(nx.NetworkXError, next, cuts)
148: (0)            @pytest.mark.slow
149: (0)            def test_alternative_flow_functions():
150: (4)                graphs = [nx.grid_2d_graph(4, 4), nx.cycle_graph(5)]
151: (4)                for G in graphs:
152: (8)                    node_conn = nx.node_connectivity(G)
153: (8)                    for flow_func in flow_funcs:
154: (12)                       all_cuts = nx.all_node_cuts(G, flow_func=flow_func)
155: (12)                       for cut in itertools.islice(all_cuts, MAX_CUTSETS_TO_TEST):
156: (16)                           assert node_conn == len(cut)
157: (16)                           assert not nx.is_connected(nx.restricted_view(G, cut, []))
158: (0)            def test_is_separating_set_complete_graph():
159: (4)                G = nx.complete_graph(5)
160: (4)                assert _is_separating_set(G, {0, 1, 2, 3})
161: (0)            def test_is_separating_set():
162: (4)                for i in [5, 10, 15]:
163: (8)                    G = nx.star_graph(i)
164: (8)                    max_degree_node = max(G, key=G.degree)
165: (8)                    assert _is_separating_set(G, {max_degree_node})
166: (0)            def test_non_repeated_cuts():
167: (4)                K = nx.karate_club_graph()
168: (4)                bcc = max(list(nx.biconnected_components(K)), key=len)
169: (4)                G = K.subgraph(bcc)
170: (4)                solution = [{32, 33}, {2, 33}, {0, 3}, {0, 1}, {29, 33}]
171: (4)                cuts = list(nx.all_node_cuts(G))
172: (4)                if len(solution) != len(cuts):
173: (8)                    print(f"Solution: {solution}")
174: (8)                    print(f"Result: {cuts}")
175: (4)                assert len(solution) == len(cuts)
176: (4)                for cut in cuts:
177: (8)                    assert cut in solution
178: (0)            def test_cycle_graph():
179: (4)                G = nx.cycle_graph(5)
180: (4)                solution = [{0, 2}, {0, 3}, {1, 3}, {1, 4}, {2, 4}]
181: (4)                cuts = list(nx.all_node_cuts(G))
182: (4)                assert len(solution) == len(cuts)
183: (4)                for cut in cuts:
184: (8)                    assert cut in solution
185: (0)            def test_complete_graph():
186: (4)                G = nx.complete_graph(5)
187: (4)                assert nx.node_connectivity(G) == 4
188: (4)                assert list(nx.all_node_cuts(G)) == []
189: (0)            def test_all_node_cuts_simple_case():
190: (4)                G = nx.complete_graph(5)
191: (4)                G.remove_edges_from([(0, 1), (3, 4)])
192: (4)                expected = [{0, 1, 2}, {2, 3, 4}]
193: (4)                actual = list(nx.all_node_cuts(G))
194: (4)                assert len(actual) == len(expected)
195: (4)                for cut in actual:
196: (8)                    assert cut in expected
```

File 235 - test_stoer_wagner.py:

```
1: (0)              from itertools import chain
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              def _check_partition(G, cut_value, partition, weight):
5: (4)                  assert isinstance(partition, tuple)
6: (4)                  assert len(partition) == 2
7: (4)                  assert isinstance(partition[0], list)
```

```
8: (4)                  assert isinstance(partition[1], list)
9: (4)                  assert len(partition[0]) > 0
10: (4)                 assert len(partition[1]) > 0
11: (4)                 assert sum(map(len, partition)) == len(G)
12: (4)                 assert set(chain.from_iterable(partition)) == set(G)
13: (4)                 partition = tuple(map(set, partition))
14: (4)                 w = 0
15: (4)                 for u, v, e in G.edges(data=True):
16: (8)                     if (u in partition[0]) == (v in partition[1]):
17: (12)                        w += e.get(weight, 1)
18: (4)                 assert w == cut_value
19: (0)             def _test_stoer_wagner(G, answer, weight="weight"):
20: (4)                 cut_value, partition = nx.stoer_wagner(G, weight,
heap=nx.utils.PairingHeap)
21: (4)                 assert cut_value == answer
22: (4)                 _check_partition(G, cut_value, partition, weight)
23: (4)                 cut_value, partition = nx.stoer_wagner(G, weight,
heap=nx.utils.BinaryHeap)
24: (4)                 assert cut_value == answer
25: (4)                 _check_partition(G, cut_value, partition, weight)
26: (0)             def test_graph1():
27: (4)                 G = nx.Graph()
28: (4)                 G.add_edge("x", "a", weight=3)
29: (4)                 G.add_edge("x", "b", weight=1)
30: (4)                 G.add_edge("a", "c", weight=3)
31: (4)                 G.add_edge("b", "c", weight=5)
32: (4)                 G.add_edge("b", "d", weight=4)
33: (4)                 G.add_edge("d", "e", weight=2)
34: (4)                 G.add_edge("c", "y", weight=2)
35: (4)                 G.add_edge("e", "y", weight=3)
36: (4)                 _test_stoer_wagner(G, 4)
37: (0)             def test_graph2():
38: (4)                 G = nx.Graph()
39: (4)                 G.add_edge("x", "a")
40: (4)                 G.add_edge("x", "b")
41: (4)                 G.add_edge("a", "c")
42: (4)                 G.add_edge("b", "c")
43: (4)                 G.add_edge("b", "d")
44: (4)                 G.add_edge("d", "e")
45: (4)                 G.add_edge("c", "y")
46: (4)                 G.add_edge("e", "y")
47: (4)                 _test_stoer_wagner(G, 2)
48: (0)             def test_graph3():
49: (4)                 G = nx.Graph()
50: (4)                 G.add_edge(1, 2, weight=2)
51: (4)                 G.add_edge(1, 5, weight=3)
52: (4)                 G.add_edge(2, 3, weight=3)
53: (4)                 G.add_edge(2, 5, weight=2)
54: (4)                 G.add_edge(2, 6, weight=2)
55: (4)                 G.add_edge(3, 4, weight=4)
56: (4)                 G.add_edge(3, 7, weight=2)
57: (4)                 G.add_edge(4, 7, weight=2)
58: (4)                 G.add_edge(4, 8, weight=2)
59: (4)                 G.add_edge(5, 6, weight=3)
60: (4)                 G.add_edge(6, 7, weight=1)
61: (4)                 G.add_edge(7, 8, weight=3)
62: (4)                 _test_stoer_wagner(G, 4)
63: (0)             def test_weight_name():
64: (4)                 G = nx.Graph()
65: (4)                 G.add_edge(1, 2, weight=1, cost=8)
66: (4)                 G.add_edge(1, 3, cost=2)
67: (4)                 G.add_edge(2, 3, cost=4)
68: (4)                 _test_stoer_wagner(G, 6, weight="cost")
69: (0)             def test_exceptions():
70: (4)                 G = nx.Graph()
71: (4)                 pytest.raises(nx.NetworkXError, nx.stoer_wagner, G)
72: (4)                 G.add_node(1)
73: (4)                 pytest.raises(nx.NetworkXError, nx.stoer_wagner, G)
74: (4)                 G.add_node(2)
```

```
75: (4)                 pytest.raises(nx.NetworkXError, nx.stoer_wagner, G)
76: (4)                 G.add_edge(1, 2, weight=-2)
77: (4)                 pytest.raises(nx.NetworkXError, nx.stoer_wagner, G)
78: (4)                 G = nx.DiGraph()
79: (4)                 pytest.raises(nx.NetworkXNotImplemented, nx.stoer_wagner, G)
80: (4)                 G = nx.MultiGraph()
81: (4)                 pytest.raises(nx.NetworkXNotImplemented, nx.stoer_wagner, G)
82: (4)                 G = nx.MultiDiGraph()
83: (4)                 pytest.raises(nx.NetworkXNotImplemented, nx.stoer_wagner, G)
```

File 236 - __init__.py:

```
1: (0)              from .maxflow import *
2: (0)              from .mincost import *
3: (0)              from .boykovkolmogorov import *
4: (0)              from .dinitz_alg import *
5: (0)              from .edmondskarp import *
6: (0)              from .gomory_hu import *
7: (0)              from .preflowpush import *
8: (0)              from .shortestaugmentingpath import *
9: (0)              from .capacityscaling import *
10: (0)             from .networksimplex import *
11: (0)             from .utils import build_flow_dict, build_residual_network
```

File 237 - capacityscaling.py:

```
1: (0)              """
2: (0)              Capacity scaling minimum cost flow algorithm.
3: (0)              """
4: (0)              __all__ = ["capacity_scaling"]
5: (0)              from itertools import chain
6: (0)              from math import log
7: (0)              import networkx as nx
8: (0)              from ...utils import BinaryHeap, arbitrary_element, not_implemented_for
9: (0)              def _detect_unboundedness(R):
10: (4)                 """Detect infinite-capacity negative cycles."""
11: (4)                 G = nx.DiGraph()
12: (4)                 G.add_nodes_from(R)
13: (4)                 inf = R.graph["inf"]
14: (4)                 f_inf = float("inf")
15: (4)                 for u in R:
16: (8)                     for v, e in R[u].items():
17: (12)                        w = f_inf
18: (12)                        for k, e in e.items():
19: (16)                            if e["capacity"] == inf:
20: (20)                                w = min(w, e["weight"])
21: (12)                        if w != f_inf:
22: (16)                            G.add_edge(u, v, weight=w)
23: (4)                 if nx.negative_edge_cycle(G):
24: (8)                     raise nx.NetworkXUnbounded(
25: (12)                        "Negative cost cycle of infinite capacity found. "
26: (12)                        "Min cost flow may be unbounded below."
27: (8)                     )
28: (0)             @not_implemented_for("undirected")
29: (0)             def _build_residual_network(G, demand, capacity, weight):
30: (4)                 """Build a residual network and initialize a zero flow."""
31: (4)                 if sum(G.nodes[u].get(demand, 0) for u in G) != 0:
32: (8)                     raise nx.NetworkXUnfeasible("Sum of the demands should be 0.")
33: (4)                 R = nx.MultiDiGraph()
34: (4)                 R.add_nodes_from(
35: (8)                     (u, {"excess": -G.nodes[u].get(demand, 0), "potential": 0}) for u in G
36: (4)                 )
37: (4)                 inf = float("inf")
38: (4)                 for u, v, e in nx.selfloop_edges(G, data=True):
39: (8)                     if e.get(weight, 0) < 0 and e.get(capacity, inf) == inf:
```

```
40: (12)                        raise nx.NetworkXUnbounded(
41: (16)                            "Negative cost cycle of infinite capacity found. "
42: (16)                            "Min cost flow may be unbounded below."
43: (12)                        )
44: (4)                 if G.is_multigraph():
45: (8)                     edge_list = [
46: (12)                        (u, v, k, e)
47: (12)                        for u, v, k, e in G.edges(data=True, keys=True)
48: (12)                        if u != v and e.get(capacity, inf) > 0
49: (8)                     ]
50: (4)                 else:
51: (8)                     edge_list = [
52: (12)                        (u, v, 0, e)
53: (12)                        for u, v, e in G.edges(data=True)
54: (12)                        if u != v and e.get(capacity, inf) > 0
55: (8)                     ]
56: (4)                 inf = (
57: (8)                     max(
58: (12)                        sum(abs(R.nodes[u]["excess"]) for u in R),
59: (12)                        2
60: (12)                        * sum(
61: (16)                            e[capacity]
62: (16)                            for u, v, k, e in edge_list
63: (16)                            if capacity in e and e[capacity] != inf
64: (12)                        ),
65: (8)                     )
66: (8)                     or 1
67: (4)                 )
68: (4)                 for u, v, k, e in edge_list:
69: (8)                     r = min(e.get(capacity, inf), inf)
70: (8)                     w = e.get(weight, 0)
71: (8)                     R.add_edge(u, v, key=(k, True), capacity=r, weight=w, flow=0)
72: (8)                     R.add_edge(v, u, key=(k, False), capacity=0, weight=-w, flow=0)
73: (4)                 R.graph["inf"] = inf
74: (4)                 _detect_unboundedness(R)
75: (4)                 return R
76: (0)             def _build_flow_dict(G, R, capacity, weight):
77: (4)                 """Build a flow dictionary from a residual network."""
78: (4)                 inf = float("inf")
79: (4)                 flow_dict = {}
80: (4)                 if G.is_multigraph():
81: (8)                     for u in G:
82: (12)                        flow_dict[u] = {}
83: (12)                        for v, es in G[u].items():
84: (16)                            flow_dict[u][v] = {
85: (20)                                k: (
86: (24)                                    0
87: (24)                                    if (
88: (28)                                        u != v or e.get(capacity, inf) <= 0 or
e.get(weight, 0) >= 0
89: (24)                                    )
90: (24)                                    else e[capacity]
91: (20)                                )
92: (20)                                for k, e in es.items()
93: (16)                            }
94: (12)                        for v, es in R[u].items():
95: (16)                            if v in flow_dict[u]:
96: (20)                                flow_dict[u][v].update(
97: (24)                                    (k[0], e["flow"]) for k, e in es.items() if e["flow"]
> 0
98: (20)                                )
99: (4)                 else:
100: (8)                    for u in G:
101: (12)                       flow_dict[u] = {
102: (16)                           v: (
103: (20)                               0
104: (20)                               if (u != v or e.get(capacity, inf) <= 0 or e.get(weight,
0) >= 0)
105: (20)                               else e[capacity]
```

```
106: (16)                            )
107: (16)                            for v, e in G[u].items()
108: (12)                        }
109: (12)                        flow_dict[u].update(
110: (16)                            (v, e["flow"])
111: (16)                            for v, es in R[u].items()
112: (16)                            for e in es.values()
113: (16)                            if e["flow"] > 0
114: (12)                        )
115: (4)                    return flow_dict
116: (0)                @nx._dispatchable(
117: (4)                    node_attrs="demand", edge_attrs={"capacity": float("inf"), "weight": 0}
118: (0)                )
119: (0)                def capacity_scaling(
120: (4)                    G, demand="demand", capacity="capacity", weight="weight", heap=BinaryHeap
121: (0)                ):
122: (4)                    r"""Find a minimum cost flow satisfying all demands in digraph G.
123: (4)                    This is a capacity scaling successive shortest augmenting path algorithm.
124: (4)                    G is a digraph with edge costs and capacities and in which nodes
125: (4)                    have demand, i.e., they want to send or receive some amount of
126: (4)                    flow. A negative demand means that the node wants to send flow, a
127: (4)                    positive demand means that the node want to receive flow. A flow on
128: (4)                    the digraph G satisfies all demand if the net flow into each node
129: (4)                    is equal to the demand of that node.
130: (4)                    Parameters
131: (4)                    ----------
132: (4)                    G : NetworkX graph
133: (8)                        DiGraph or MultiDiGraph on which a minimum cost flow satisfying all
134: (8)                        demands is to be found.
135: (4)                    demand : string
136: (8)                        Nodes of the graph G are expected to have an attribute demand
137: (8)                        that indicates how much flow a node wants to send (negative
138: (8)                        demand) or receive (positive demand). Note that the sum of the
139: (8)                        demands should be 0 otherwise the problem in not feasible. If
140: (8)                        this attribute is not present, a node is considered to have 0
141: (8)                        demand. Default value: 'demand'.
142: (4)                    capacity : string
143: (8)                        Edges of the graph G are expected to have an attribute capacity
144: (8)                        that indicates how much flow the edge can support. If this
145: (8)                        attribute is not present, the edge is considered to have
146: (8)                        infinite capacity. Default value: 'capacity'.
147: (4)                    weight : string
148: (8)                        Edges of the graph G are expected to have an attribute weight
149: (8)                        that indicates the cost incurred by sending one unit of flow on
150: (8)                        that edge. If not present, the weight is considered to be 0.
151: (8)                        Default value: 'weight'.
152: (4)                    heap : class
153: (8)                        Type of heap to be used in the algorithm. It should be a subclass of
154: (8)                        :class:`MinHeap` or implement a compatible interface.
155: (8)                        If a stock heap implementation is to be used, :class:`BinaryHeap` is
156: (8)                        recommended over :class:`PairingHeap` for Python implementations
without
157: (8)                        optimized attribute accesses (e.g., CPython) despite a slower
158: (8)                        asymptotic running time. For Python implementations with optimized
159: (8)                        attribute accesses (e.g., PyPy), :class:`PairingHeap` provides better
160: (8)                        performance. Default value: :class:`BinaryHeap`.
161: (4)                    Returns
162: (4)                    -------
163: (4)                    flowCost : integer
164: (8)                        Cost of a minimum cost flow satisfying all demands.
165: (4)                    flowDict : dictionary
166: (8)                        If G is a digraph, a dict-of-dicts keyed by nodes such that
167: (8)                        flowDict[u][v] is the flow on edge (u, v).
168: (8)                        If G is a MultiDiGraph, a dict-of-dicts-of-dicts keyed by nodes
169: (8)                        so that flowDict[u][v][key] is the flow on edge (u, v, key).
170: (4)                    Raises
171: (4)                    ------
172: (4)                    NetworkXError
173: (8)                        This exception is raised if the input graph is not directed,
```

```
(8)                 not connected.
(4)             NetworkXUnfeasible
(8)                 This exception is raised in the following situations:
(12)                    * The sum of the demands is not zero. Then, there is no
(14)                      flow satisfying all demands.
(12)                    * There is no flow satisfying all demand.
(4)             NetworkXUnbounded
(8)                 This exception is raised if the digraph G has a cycle of
(8)                 negative cost and infinite capacity. Then, the cost of a flow
(8)                 satisfying all demands is unbounded below.
(4)             Notes
(4)             -----
(4)             This algorithm does not work if edge weights are floating-point numbers.
(4)             See also
(4)             --------
(4)             :meth:`network_simplex`
(4)             Examples
(4)             --------
(4)             A simple example of a min cost flow problem.
(4)             >>> G = nx.DiGraph()
(4)             >>> G.add_node("a", demand=-5)
(4)             >>> G.add_node("d", demand=5)
(4)             >>> G.add_edge("a", "b", weight=3, capacity=4)
(4)             >>> G.add_edge("a", "c", weight=6, capacity=10)
(4)             >>> G.add_edge("b", "d", weight=1, capacity=9)
(4)             >>> G.add_edge("c", "d", weight=2, capacity=5)
(4)             >>> flowCost, flowDict = nx.capacity_scaling(G)
(4)             >>> flowCost
(4)             24
(4)             >>> flowDict
(4)             {'a': {'b': 4, 'c': 1}, 'd': {}, 'b': {'d': 4}, 'c': {'d': 1}}
(4)             It is possible to change the name of the attributes used for the
(4)             algorithm.
(4)             >>> G = nx.DiGraph()
(4)             >>> G.add_node("p", spam=-4)
(4)             >>> G.add_node("q", spam=2)
(4)             >>> G.add_node("a", spam=-2)
(4)             >>> G.add_node("d", spam=-1)
(4)             >>> G.add_node("t", spam=2)
(4)             >>> G.add_node("w", spam=3)
(4)             >>> G.add_edge("p", "q", cost=7, vacancies=5)
(4)             >>> G.add_edge("p", "a", cost=1, vacancies=4)
(4)             >>> G.add_edge("q", "d", cost=2, vacancies=3)
(4)             >>> G.add_edge("t", "q", cost=1, vacancies=2)
(4)             >>> G.add_edge("a", "t", cost=2, vacancies=4)
(4)             >>> G.add_edge("d", "w", cost=3, vacancies=4)
(4)             >>> G.add_edge("t", "w", cost=4, vacancies=1)
(4)             >>> flowCost, flowDict = nx.capacity_scaling(
(4)             ...     G, demand="spam", capacity="vacancies", weight="cost"
(4)             ... )
(4)             >>> flowCost
(4)             37
(4)             >>> flowDict
(4)             {'p': {'q': 2, 'a': 2}, 'q': {'d': 1}, 'a': {'t': 4}, 'd': {'w': 2}, 't':
{'q': 1, 'w': 1}, 'w': {}}
(4)             """
(4)             R = _build_residual_network(G, demand, capacity, weight)
(4)             inf = float("inf")
(4)             flow_cost = sum(
(8)                 0
(8)                 if e.get(capacity, inf) <= 0 or e.get(weight, 0) >= 0
(8)                 else e[capacity] * e[weight]
(8)                 for u, v, e in nx.selfloop_edges(G, data=True)
(4)             )
(4)             wmax = max(chain([-inf], (e["capacity"] for u, v, e in
R.edges(data=True))))
(4)             if wmax == -inf:
(8)                 return flow_cost, _build_flow_dict(G, R, capacity, weight)
(4)             R_nodes = R.nodes
```

```
(4)                 R_succ = R.succ
(4)                 delta = 2 ** int(log(wmax, 2))
(4)                 while delta >= 1:
(8)                     for u in R:
(12)                        p_u = R_nodes[u]["potential"]
(12)                        for v, es in R_succ[u].items():
(16)                            for k, e in es.items():
(20)                                flow = e["capacity"] - e["flow"]
(20)                                if e["weight"] - p_u + R_nodes[v]["potential"] < 0:
(24)                                    flow = e["capacity"] - e["flow"]
(24)                                    if flow >= delta:
(28)                                        e["flow"] += flow
(28)                                        R_succ[v][u][(k[0], not k[1])]["flow"] -= flow
(28)                                        R_nodes[u]["excess"] -= flow
(28)                                        R_nodes[v]["excess"] += flow
(8)                     S = set()
(8)                     T = set()
(8)                     S_add = S.add
(8)                     S_remove = S.remove
(8)                     T_add = T.add
(8)                     T_remove = T.remove
(8)                     for u in R:
(12)                        excess = R_nodes[u]["excess"]
(12)                        if excess >= delta:
(16)                            S_add(u)
(12)                        elif excess <= -delta:
(16)                            T_add(u)
(8)                     while S and T:
(12)                        s = arbitrary_element(S)
(12)                        t = None
(12)                        d = {}
(12)                        pred = {s: None}
(12)                        h = heap()
(12)                        h_insert = h.insert
(12)                        h_get = h.get
(12)                        h_insert(s, 0)
(12)                        while h:
(16)                            u, d_u = h.pop()
(16)                            d[u] = d_u
(16)                            if u in T:
(20)                                t = u
(20)                                break
(16)                            p_u = R_nodes[u]["potential"]
(16)                            for v, es in R_succ[u].items():
(20)                                if v in d:
(24)                                    continue
(20)                                wmin = inf
(20)                                for k, e in es.items():
(24)                                    if e["capacity"] - e["flow"] >= delta:
(28)                                        w = e["weight"]
(28)                                        if w < wmin:
(32)                                            wmin = w
(32)                                            kmin = k
(32)                                            emin = e
(20)                                if wmin == inf:
(24)                                    continue
(20)                                d_v = d_u + wmin - p_u + R_nodes[v]["potential"]
(20)                                if h_insert(v, d_v):
(24)                                    pred[v] = (u, kmin, emin)
(12)                        if t is not None:
(16)                            while u != s:
(20)                                v = u
(20)                                u, k, e = pred[v]
(20)                                e["flow"] += delta
(20)                                R_succ[v][u][(k[0], not k[1])]["flow"] -= delta
(16)                            R_nodes[s]["excess"] -= delta
(16)                            R_nodes[t]["excess"] += delta
(16)                            if R_nodes[s]["excess"] < delta:
(20)                                S_remove(s)
```

```
310: (16)                            if R_nodes[t]["excess"] > -delta:
311: (20)                                T_remove(t)
312: (16)                            d_t = d[t]
313: (16)                            for u, d_u in d.items():
314: (20)                                R_nodes[u]["potential"] -= d_u - d_t
315: (12)                        else:
316: (16)                            S_remove(s)
317: (8)                    delta //= 2
318: (4)                if any(R.nodes[u]["excess"] != 0 for u in R):
319: (8)                    raise nx.NetworkXUnfeasible("No flow satisfying all demands.")
320: (4)                for u in R:
321: (8)                    for v, es in R_succ[u].items():
322: (12)                       for e in es.values():
323: (16)                           flow = e["flow"]
324: (16)                           if flow > 0:
325: (20)                               flow_cost += flow * e["weight"]
326: (4)                return flow_cost, _build_flow_dict(G, R, capacity, weight)
```

File 238 - boykovkolmogorov.py:

```
1: (0)              """
2: (0)              Boykov-Kolmogorov algorithm for maximum flow problems.
3: (0)              """
4: (0)              from collections import deque
5: (0)              from operator import itemgetter
6: (0)              import networkx as nx
7: (0)              from networkx.algorithms.flow.utils import build_residual_network
8: (0)              __all__ = ["boykov_kolmogorov"]
9: (0)              @nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
10: (0)             def boykov_kolmogorov(
11: (4)                 G, s, t, capacity="capacity", residual=None, value_only=False, cutoff=None
12: (0)             ):
13: (4)                 r"""Find a maximum single-commodity flow using Boykov-Kolmogorov
algorithm.
14: (4)                 This function returns the residual network resulting after computing
15: (4)                 the maximum flow. See below for details about the conventions
16: (4)                 NetworkX uses for defining residual networks.
17: (4)                 This algorithm has worse case complexity $O(n^2 m |C|)$ for $n$ nodes, $m$
18: (4)                 edges, and $|C|$ the cost of the minimum cut [1]_. This implementation
19: (4)                 uses the marking heuristic defined in [2]_ which improves its running
20: (4)                 time in many practical problems.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : NetworkX graph
24: (8)                     Edges of the graph are expected to have an attribute called
25: (8)                     'capacity'. If this attribute is not present, the edge is
26: (8)                     considered to have infinite capacity.
27: (4)                 s : node
28: (8)                     Source node for the flow.
29: (4)                 t : node
30: (8)                     Sink node for the flow.
31: (4)                 capacity : string
32: (8)                     Edges of the graph G are expected to have an attribute capacity
33: (8)                     that indicates how much flow the edge can support. If this
34: (8)                     attribute is not present, the edge is considered to have
35: (8)                     infinite capacity. Default value: 'capacity'.
36: (4)                 residual : NetworkX graph
37: (8)                     Residual network on which the algorithm is to be executed. If None, a
38: (8)                     new residual network is created. Default value: None.
39: (4)                 value_only : bool
40: (8)                     If True compute only the value of the maximum flow. This parameter
41: (8)                     will be ignored by this algorithm because it is not applicable.
42: (4)                 cutoff : integer, float
43: (8)                     If specified, the algorithm will terminate when the flow value reaches
44: (8)                     or exceeds the cutoff. In this case, it may be unable to immediately
45: (8)                     determine a minimum cut. Default value: None.
46: (4)                 Returns
```

```
(4)                 -------
(4)                 R : NetworkX DiGraph
(8)                     Residual network after computing the maximum flow.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     The algorithm does not support MultiGraph and MultiDiGraph. If
(8)                     the input graph is an instance of one of these two classes, a
(8)                     NetworkXError is raised.
(4)                 NetworkXUnbounded
(8)                     If the graph has a path of infinite capacity, the value of a
(8)                     feasible flow on the graph is unbounded above and the function
(8)                     raises a NetworkXUnbounded.
(4)                 See also
(4)                 --------
(4)                 :meth:`maximum_flow`
(4)                 :meth:`minimum_cut`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 Notes
(4)                 -----
(4)                 The residual network :samp:`R` from an input graph :samp:`G` has the
(4)                 same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
(4)                 of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
(4)                 self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
(4)                 in :samp:`G`.
(4)                 For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
(4)                 is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
(4)                 in :samp:`G` or zero otherwise. If the capacity is infinite,
(4)                 :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
(4)                 that does not affect the solution of the problem. This value is stored in
(4)                 :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
(4)                 :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
(4)                 satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
(4)                 The flow value, defined as the total flow into :samp:`t`, the sink, is
(4)                 stored in :samp:`R.graph['flow_value']`. If :samp:`cutoff` is not
(4)                 specified, reachability to :samp:`t` using only edges :samp:`(u, v)` such
(4)                 that :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
(4)                 :samp:`s`-:samp:`t` cut.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms.flow import boykov_kolmogorov
(4)                 The functions that implement flow algorithms and output a residual
(4)                 network, such as this one, are not imported to the base NetworkX
(4)                 namespace, so you have to explicitly import them from the flow package.
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_edge("x", "a", capacity=3.0)
(4)                 >>> G.add_edge("x", "b", capacity=1.0)
(4)                 >>> G.add_edge("a", "c", capacity=3.0)
(4)                 >>> G.add_edge("b", "c", capacity=5.0)
(4)                 >>> G.add_edge("b", "d", capacity=4.0)
(4)                 >>> G.add_edge("d", "e", capacity=2.0)
(4)                 >>> G.add_edge("c", "y", capacity=2.0)
(4)                 >>> G.add_edge("e", "y", capacity=3.0)
(4)                 >>> R = boykov_kolmogorov(G, "x", "y")
(4)                 >>> flow_value = nx.maximum_flow_value(G, "x", "y")
(4)                 >>> flow_value
(4)                 3.0
(4)                 >>> flow_value == R.graph["flow_value"]
(4)                 True
(4)                 A nice feature of the Boykov-Kolmogorov algorithm is that a partition
(4)                 of the nodes that defines a minimum cut can be easily computed based
(4)                 on the search trees used during the algorithm. These trees are stored
(4)                 in the graph attribute `trees` of the residual network.
(4)                 >>> source_tree, target_tree = R.graph["trees"]
(4)                 >>> partition = (set(source_tree), set(G) - set(source_tree))
(4)                 Or equivalently:
(4)                 >>> partition = (set(G) - set(target_tree), set(target_tree))
(4)                 References
```

```
(4)                 ----------
(4)                 .. [1] Boykov, Y., & Kolmogorov, V. (2004). An experimental comparison
(11)                       of min-cut/max-flow algorithms for energy minimization in vision.
(11)                       Pattern Analysis and Machine Intelligence, IEEE Transactions on,
(11)                       26(9), 1124-1137.
(11)                       https://doi.org/10.1109/TPAMI.2004.60
(4)                 .. [2] Vladimir Kolmogorov. Graph-based Algorithms for Multi-camera
(11)                       Reconstruction Problem. PhD thesis, Cornell University, CS Department,
(11)                       2003. pp. 109-114.
(11)                https://web.archive.org/web/20170809091249/https://pub.ist.ac.at/~vnk/papers/thesis.pdf
(4)                 """
(4)                 R = boykov_kolmogorov_impl(G, s, t, capacity, residual, cutoff)
(4)                 R.graph["algorithm"] = "boykov_kolmogorov"
(4)                 nx._clear_cache(R)
(4)                 return R
(0)             def boykov_kolmogorov_impl(G, s, t, capacity, residual, cutoff):
(4)                 if s not in G:
(8)                     raise nx.NetworkXError(f"node {str(s)} not in graph")
(4)                 if t not in G:
(8)                     raise nx.NetworkXError(f"node {str(t)} not in graph")
(4)                 if s == t:
(8)                     raise nx.NetworkXError("source and sink are the same node")
(4)                 if residual is None:
(8)                     R = build_residual_network(G, capacity)
(4)                 else:
(8)                     R = residual
(4)                 for u in R:
(8)                     for e in R[u].values():
(12)                        e["flow"] = 0
(4)                 INF = R.graph["inf"]
(4)                 if cutoff is None:
(8)                     cutoff = INF
(4)                 R_succ = R.succ
(4)                 R_pred = R.pred
(4)                 def grow():
(8)                     """Bidirectional breadth-first search for the growth stage.
(8)                     Returns a connecting edge, that is and edge that connects
(8)                     a node from the source search tree with a node from the
(8)                     target search tree.
(8)                     The first node in the connecting edge is always from the
(8)                     source tree and the last node from the target tree.
(8)                     """
(8)                     while active:
(12)                        u = active[0]
(12)                        if u in source_tree:
(16)                            this_tree = source_tree
(16)                            other_tree = target_tree
(16)                            neighbors = R_succ
(12)                        else:
(16)                            this_tree = target_tree
(16)                            other_tree = source_tree
(16)                            neighbors = R_pred
(12)                        for v, attr in neighbors[u].items():
(16)                            if attr["capacity"] - attr["flow"] > 0:
(20)                                if v not in this_tree:
(24)                                    if v in other_tree:
(28)                                        return (u, v) if this_tree is source_tree else (v, u)
(24)                                    this_tree[v] = u
(24)                                    dist[v] = dist[u] + 1
(24)                                    timestamp[v] = timestamp[u]
(24)                                    active.append(v)
(20)                                elif v in this_tree and _is_closer(u, v):
(24)                                    this_tree[v] = u
(24)                                    dist[v] = dist[u] + 1
(24)                                    timestamp[v] = timestamp[u]
(12)                        _ = active.popleft()
```

```
(8)                 return None, None
(4)             def augment(u, v):
(8)                 """Augmentation stage.
(8)                 Reconstruct path and determine its residual capacity.
(8)                 We start from a connecting edge, which links a node
(8)                 from the source tree to a node from the target tree.
(8)                 The connecting edge is the output of the grow function
(8)                 and the input of this function.
(8)                 """
(8)                 attr = R_succ[u][v]
(8)                 flow = min(INF, attr["capacity"] - attr["flow"])
(8)                 path = [u]
(8)                 w = u
(8)                 while w != s:
(12)                    n = w
(12)                    w = source_tree[n]
(12)                    attr = R_pred[n][w]
(12)                    flow = min(flow, attr["capacity"] - attr["flow"])
(12)                    path.append(w)
(8)                 path.reverse()
(8)                 path.append(v)
(8)                 w = v
(8)                 while w != t:
(12)                    n = w
(12)                    w = target_tree[n]
(12)                    attr = R_succ[n][w]
(12)                    flow = min(flow, attr["capacity"] - attr["flow"])
(12)                    path.append(w)
(8)                 it = iter(path)
(8)                 u = next(it)
(8)                 these_orphans = []
(8)                 for v in it:
(12)                    R_succ[u][v]["flow"] += flow
(12)                    R_succ[v][u]["flow"] -= flow
(12)                    if R_succ[u][v]["flow"] == R_succ[u][v]["capacity"]:
(16)                        if v in source_tree:
(20)                            source_tree[v] = None
(20)                            these_orphans.append(v)
(16)                        if u in target_tree:
(20)                            target_tree[u] = None
(20)                            these_orphans.append(u)
(12)                    u = v
(8)                 orphans.extend(sorted(these_orphans, key=dist.get))
(8)                 return flow
(4)             def adopt():
(8)                 """Adoption stage.
(8)                 Reconstruct search trees by adopting or discarding orphans.
(8)                 During augmentation stage some edges got saturated and thus
(8)                 the source and target search trees broke down to forests, with
(8)                 orphans as roots of some of its trees. We have to reconstruct
(8)                 the search trees rooted to source and target before we can grow
(8)                 them again.
(8)                 """
(8)                 while orphans:
(12)                    u = orphans.popleft()
(12)                    if u in source_tree:
(16)                        tree = source_tree
(16)                        neighbors = R_pred
(12)                    else:
(16)                        tree = target_tree
(16)                        neighbors = R_succ
(12)                    nbrs = ((n, attr, dist[n]) for n, attr in neighbors[u].items() if
n in tree)
(12)                    for v, attr, d in sorted(nbrs, key=itemgetter(2)):
(16)                        if attr["capacity"] - attr["flow"] > 0:
(20)                            if _has_valid_root(v, tree):
(24)                                tree[u] = v
(24)                                dist[u] = dist[v] + 1
(24)                                timestamp[u] = time
```

```
250: (24)                                   break
251: (12)                       else:
252: (16)                           nbrs = (
253: (20)                               (n, attr, dist[n]) for n, attr in neighbors[u].items() if
n in tree
254: (16)                           )
255: (16)                           for v, attr, d in sorted(nbrs, key=itemgetter(2)):
256: (20)                               if attr["capacity"] - attr["flow"] > 0:
257: (24)                                   if v not in active:
258: (28)                                       active.append(v)
259: (20)                               if tree[v] == u:
260: (24)                                   tree[v] = None
261: (24)                                   orphans.appendleft(v)
262: (16)                           if u in active:
263: (20)                               active.remove(u)
264: (16)                           del tree[u]
265: (4)                def _has_valid_root(n, tree):
266: (8)                    path = []
267: (8)                    v = n
268: (8)                    while v is not None:
269: (12)                       path.append(v)
270: (12)                       if v in (s, t):
271: (16)                           base_dist = 0
272: (16)                           break
273: (12)                       elif timestamp[v] == time:
274: (16)                           base_dist = dist[v]
275: (16)                           break
276: (12)                       v = tree[v]
277: (8)                    else:
278: (12)                       return False
279: (8)                    length = len(path)
280: (8)                    for i, u in enumerate(path, 1):
281: (12)                       dist[u] = base_dist + length - i
282: (12)                       timestamp[u] = time
283: (8)                    return True
284: (4)                def _is_closer(u, v):
285: (8)                    return timestamp[v] <= timestamp[u] and dist[v] > dist[u] + 1
286: (4)                source_tree = {s: None}
287: (4)                target_tree = {t: None}
288: (4)                active = deque([s, t])
289: (4)                orphans = deque()
290: (4)                flow_value = 0
291: (4)                time = 1
292: (4)                timestamp = {s: time, t: time}
293: (4)                dist = {s: 0, t: 0}
294: (4)                while flow_value < cutoff:
295: (8)                    u, v = grow()
296: (8)                    if u is None:
297: (12)                       break
298: (8)                    time += 1
299: (8)                    flow_value += augment(u, v)
300: (8)                    adopt()
301: (4)                if flow_value * 2 > INF:
302: (8)                    raise nx.NetworkXUnbounded("Infinite capacity path, flow unbounded
above.")
303: (4)                R.graph["trees"] = (source_tree, target_tree)
304: (4)                R.graph["flow_value"] = flow_value
305: (4)                return R
```

File 239 - dinitz_alg.py:

```
1: (0)              """
2: (0)              Dinitz' algorithm for maximum flow problems.
3: (0)              """
4: (0)              from collections import deque
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.flow.utils import build_residual_network
```

```
from networkx.utils import pairwise
__all__ = ["dinitz"]
@nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
def dinitz(G, s, t, capacity="capacity", residual=None, value_only=False, cutoff=None):
    """Find a maximum single-commodity flow using Dinitz' algorithm.
    This function returns the residual network resulting after computing
    the maximum flow. See below for details about the conventions
    NetworkX uses for defining residual networks.
    This algorithm has a running time of $O(n^2 m)$ for $n$ nodes and $m$
    edges [1]_.
    Parameters
    ----------
    G : NetworkX graph
        Edges of the graph are expected to have an attribute called
        'capacity'. If this attribute is not present, the edge is
        considered to have infinite capacity.
    s : node
        Source node for the flow.
    t : node
        Sink node for the flow.
    capacity : string
        Edges of the graph G are expected to have an attribute capacity
        that indicates how much flow the edge can support. If this
        attribute is not present, the edge is considered to have
        infinite capacity. Default value: 'capacity'.
    residual : NetworkX graph
        Residual network on which the algorithm is to be executed. If None, a
        new residual network is created. Default value: None.
    value_only : bool
        If True compute only the value of the maximum flow. This parameter
        will be ignored by this algorithm because it is not applicable.
    cutoff : integer, float
        If specified, the algorithm will terminate when the flow value reaches
        or exceeds the cutoff. In this case, it may be unable to immediately
        determine a minimum cut. Default value: None.
    Returns
    -------
    R : NetworkX DiGraph
        Residual network after computing the maximum flow.
    Raises
    ------
    NetworkXError
        The algorithm does not support MultiGraph and MultiDiGraph. If
        the input graph is an instance of one of these two classes, a
        NetworkXError is raised.
    NetworkXUnbounded
        If the graph has a path of infinite capacity, the value of a
        feasible flow on the graph is unbounded above and the function
        raises a NetworkXUnbounded.
    See also
    --------
    :meth:`maximum_flow`
    :meth:`minimum_cut`
    :meth:`preflow_push`
    :meth:`shortest_augmenting_path`
    Notes
    -----
    The residual network :samp:`R` from an input graph :samp:`G` has the
    same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
    of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
    self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
    in :samp:`G`.
    For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
    is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
    in :samp:`G` or zero otherwise. If the capacity is infinite,
    :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
    that does not affect the solution of the problem. This value is stored in
    :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
```

```
(4)                 :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
(4)                 satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
(4)                 The flow value, defined as the total flow into :samp:`t`, the sink, is
(4)                 stored in :samp:`R.graph['flow_value']`. If :samp:`cutoff` is not
(4)                 specified, reachability to :samp:`t` using only edges :samp:`(u, v)` such
(4)                 that :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
(4)                 :samp:`s`-:samp:`t` cut.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms.flow import dinitz
(4)                 The functions that implement flow algorithms and output a residual
(4)                 network, such as this one, are not imported to the base NetworkX
(4)                 namespace, so you have to explicitly import them from the flow package.
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_edge("x", "a", capacity=3.0)
(4)                 >>> G.add_edge("x", "b", capacity=1.0)
(4)                 >>> G.add_edge("a", "c", capacity=3.0)
(4)                 >>> G.add_edge("b", "c", capacity=5.0)
(4)                 >>> G.add_edge("b", "d", capacity=4.0)
(4)                 >>> G.add_edge("d", "e", capacity=2.0)
(4)                 >>> G.add_edge("c", "y", capacity=2.0)
(4)                 >>> G.add_edge("e", "y", capacity=3.0)
(4)                 >>> R = dinitz(G, "x", "y")
(4)                 >>> flow_value = nx.maximum_flow_value(G, "x", "y")
(4)                 >>> flow_value
(4)                 3.0
(4)                 >>> flow_value == R.graph["flow_value"]
(4)                 True
(4)                 References
(4)                 ----------
(4)                 .. [1] Dinitz' Algorithm: The Original Version and Even's Version.
(11)                       2006. Yefim Dinitz. In Theoretical Computer Science. Lecture
(11)                       Notes in Computer Science. Volume 3895. pp 218-240.
(11)                       https://doi.org/10.1007/11685654_10
(4)                 """
(4)                 R = dinitz_impl(G, s, t, capacity, residual, cutoff)
(4)                 R.graph["algorithm"] = "dinitz"
(4)                 nx._clear_cache(R)
(4)                 return R
(0)             def dinitz_impl(G, s, t, capacity, residual, cutoff):
(4)                 if s not in G:
(8)                     raise nx.NetworkXError(f"node {str(s)} not in graph")
(4)                 if t not in G:
(8)                     raise nx.NetworkXError(f"node {str(t)} not in graph")
(4)                 if s == t:
(8)                     raise nx.NetworkXError("source and sink are the same node")
(4)                 if residual is None:
(8)                     R = build_residual_network(G, capacity)
(4)                 else:
(8)                     R = residual
(4)                 for u in R:
(8)                     for e in R[u].values():
(12)                        e["flow"] = 0
(4)                 INF = R.graph["inf"]
(4)                 if cutoff is None:
(8)                     cutoff = INF
(4)                 R_succ = R.succ
(4)                 R_pred = R.pred
(4)                 def breath_first_search():
(8)                     parents = {}
(8)                     vertex_dist = {s: 0}
(8)                     queue = deque([(s, 0)])
(8)                     while queue:
(12)                        if t in parents:
(16)                            break
(12)                        u, dist = queue.popleft()
(12)                        for v, attr in R_succ[u].items():
(16)                            if attr["capacity"] - attr["flow"] > 0:
(20)                                if v in parents:
```

```
144: (24)                                   if vertex_dist[v] == dist + 1:
145: (28)                                       parents[v].append(u)
146: (20)                               else:
147: (24)                                   parents[v] = deque([u])
148: (24)                                   vertex_dist[v] = dist + 1
149: (24)                                   queue.append((v, dist + 1))
150: (8)                    return parents
151: (4)                def depth_first_search(parents):
152: (8)                    """Build a path using DFS starting from the sink"""
153: (8)                    total_flow = 0
154: (8)                    u = t
155: (8)                    path = [u]
156: (8)                    while True:
157: (12)                       if len(parents[u]) > 0:
158: (16)                           v = parents[u][0]
159: (16)                           path.append(v)
160: (12)                       else:
161: (16)                           path.pop()
162: (16)                           if len(path) == 0:
163: (20)                               break
164: (16)                           v = path[-1]
165: (16)                           parents[v].popleft()
166: (12)                       if v == s:
167: (16)                           flow = INF
168: (16)                           for u, v in pairwise(path):
169: (20)                               flow = min(flow, R_pred[u][v]["capacity"] - R_pred[u][v]
["flow"])
170: (16)                           for u, v in pairwise(reversed(path)):
171: (20)                               R_pred[v][u]["flow"] += flow
172: (20)                               R_pred[u][v]["flow"] -= flow
173: (20)                               if R_pred[v][u]["capacity"] - R_pred[v][u]["flow"] == 0:
174: (24)                                   parents[v].popleft()
175: (24)                                   while path[-1] != v:
176: (28)                                       path.pop()
177: (16)                           total_flow += flow
178: (16)                           v = path[-1]
179: (12)                       u = v
180: (8)                    return total_flow
181: (4)                flow_value = 0
182: (4)                while flow_value < cutoff:
183: (8)                    parents = breath_first_search()
184: (8)                    if t not in parents:
185: (12)                       break
186: (8)                    this_flow = depth_first_search(parents)
187: (8)                    if this_flow * 2 > INF:
188: (12)                       raise nx.NetworkXUnbounded("Infinite capacity path, flow unbounded
above.")
189: (8)                    flow_value += this_flow
190: (4)                R.graph["flow_value"] = flow_value
191: (4)                return R
```

File 240 - edmondskarp.py:

```
1: (0)              """
2: (0)              Edmonds-Karp algorithm for maximum flow problems.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.flow.utils import build_residual_network
6: (0)              __all__ = ["edmonds_karp"]
7: (0)              def edmonds_karp_core(R, s, t, cutoff):
8: (4)                  """Implementation of the Edmonds-Karp algorithm."""
9: (4)                  R_nodes = R.nodes
10: (4)                 R_pred = R.pred
11: (4)                 R_succ = R.succ
12: (4)                 inf = R.graph["inf"]
13: (4)                 def augment(path):
14: (8)                     """Augment flow along a path from s to t."""
```

```
 15: (8)                    flow = inf
 16: (8)                    it = iter(path)
 17: (8)                    u = next(it)
 18: (8)                    for v in it:
 19: (12)                       attr = R_succ[u][v]
 20: (12)                       flow = min(flow, attr["capacity"] - attr["flow"])
 21: (12)                       u = v
 22: (8)                    if flow * 2 > inf:
 23: (12)                       raise nx.NetworkXUnbounded("Infinite capacity path, flow unbounded
 above.")
 24: (8)                    it = iter(path)
 25: (8)                    u = next(it)
 26: (8)                    for v in it:
 27: (12)                       R_succ[u][v]["flow"] += flow
 28: (12)                       R_succ[v][u]["flow"] -= flow
 29: (12)                       u = v
 30: (8)                    return flow
 31: (4)                def bidirectional_bfs():
 32: (8)                    """Bidirectional breadth-first search for an augmenting path."""
 33: (8)                    pred = {s: None}
 34: (8)                    q_s = [s]
 35: (8)                    succ = {t: None}
 36: (8)                    q_t = [t]
 37: (8)                    while True:
 38: (12)                       q = []
 39: (12)                       if len(q_s) <= len(q_t):
 40: (16)                           for u in q_s:
 41: (20)                               for v, attr in R_succ[u].items():
 42: (24)                                   if v not in pred and attr["flow"] < attr["capacity"]:
 43: (28)                                       pred[v] = u
 44: (28)                                       if v in succ:
 45: (32)                                           return v, pred, succ
 46: (28)                                       q.append(v)
 47: (16)                           if not q:
 48: (20)                               return None, None, None
 49: (16)                           q_s = q
 50: (12)                       else:
 51: (16)                           for u in q_t:
 52: (20)                               for v, attr in R_pred[u].items():
 53: (24)                                   if v not in succ and attr["flow"] < attr["capacity"]:
 54: (28)                                       succ[v] = u
 55: (28)                                       if v in pred:
 56: (32)                                           return v, pred, succ
 57: (28)                                       q.append(v)
 58: (16)                           if not q:
 59: (20)                               return None, None, None
 60: (16)                           q_t = q
 61: (4)                flow_value = 0
 62: (4)                while flow_value < cutoff:
 63: (8)                    v, pred, succ = bidirectional_bfs()
 64: (8)                    if pred is None:
 65: (12)                       break
 66: (8)                    path = [v]
 67: (8)                    u = v
 68: (8)                    while u != s:
 69: (12)                       u = pred[u]
 70: (12)                       path.append(u)
 71: (8)                    path.reverse()
 72: (8)                    u = v
 73: (8)                    while u != t:
 74: (12)                       u = succ[u]
 75: (12)                       path.append(u)
 76: (8)                    flow_value += augment(path)
 77: (4)                return flow_value
 78: (0)            def edmonds_karp_impl(G, s, t, capacity, residual, cutoff):
 79: (4)                """Implementation of the Edmonds-Karp algorithm."""
 80: (4)                if s not in G:
 81: (8)                    raise nx.NetworkXError(f"node {str(s)} not in graph")
 82: (4)                if t not in G:
```

```
 83: (8)                    raise nx.NetworkXError(f"node {str(t)} not in graph")
 84: (4)                if s == t:
 85: (8)                    raise nx.NetworkXError("source and sink are the same node")
 86: (4)                if residual is None:
 87: (8)                    R = build_residual_network(G, capacity)
 88: (4)                else:
 89: (8)                    R = residual
 90: (4)                for u in R:
 91: (8)                    for e in R[u].values():
 92: (12)                       e["flow"] = 0
 93: (4)                if cutoff is None:
 94: (8)                    cutoff = float("inf")
 95: (4)                R.graph["flow_value"] = edmonds_karp_core(R, s, t, cutoff)
 96: (4)                return R
 97: (0)            @nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
 98: (0)            def edmonds_karp(
 99: (4)                G, s, t, capacity="capacity", residual=None, value_only=False, cutoff=None
100: (0)            ):
101: (4)                """Find a maximum single-commodity flow using the Edmonds-Karp algorithm.
102: (4)                This function returns the residual network resulting after computing
103: (4)                the maximum flow. See below for details about the conventions
104: (4)                NetworkX uses for defining residual networks.
105: (4)                This algorithm has a running time of $O(n m^2)$ for $n$ nodes and $m$
106: (4)                edges.
107: (4)                Parameters
108: (4)                ----------
109: (4)                G : NetworkX graph
110: (8)                    Edges of the graph are expected to have an attribute called
111: (8)                    'capacity'. If this attribute is not present, the edge is
112: (8)                    considered to have infinite capacity.
113: (4)                s : node
114: (8)                    Source node for the flow.
115: (4)                t : node
116: (8)                    Sink node for the flow.
117: (4)                capacity : string
118: (8)                    Edges of the graph G are expected to have an attribute capacity
119: (8)                    that indicates how much flow the edge can support. If this
120: (8)                    attribute is not present, the edge is considered to have
121: (8)                    infinite capacity. Default value: 'capacity'.
122: (4)                residual : NetworkX graph
123: (8)                    Residual network on which the algorithm is to be executed. If None, a
124: (8)                    new residual network is created. Default value: None.
125: (4)                value_only : bool
126: (8)                    If True compute only the value of the maximum flow. This parameter
127: (8)                    will be ignored by this algorithm because it is not applicable.
128: (4)                cutoff : integer, float
129: (8)                    If specified, the algorithm will terminate when the flow value reaches
130: (8)                    or exceeds the cutoff. In this case, it may be unable to immediately
131: (8)                    determine a minimum cut. Default value: None.
132: (4)                Returns
133: (4)                -------
134: (4)                R : NetworkX DiGraph
135: (8)                    Residual network after computing the maximum flow.
136: (4)                Raises
137: (4)                ------
138: (4)                NetworkXError
139: (8)                    The algorithm does not support MultiGraph and MultiDiGraph. If
140: (8)                    the input graph is an instance of one of these two classes, a
141: (8)                    NetworkXError is raised.
142: (4)                NetworkXUnbounded
143: (8)                    If the graph has a path of infinite capacity, the value of a
144: (8)                    feasible flow on the graph is unbounded above and the function
145: (8)                    raises a NetworkXUnbounded.
146: (4)                See also
147: (4)                --------
148: (4)                :meth:`maximum_flow`
149: (4)                :meth:`minimum_cut`
150: (4)                :meth:`preflow_push`
151: (4)                :meth:`shortest_augmenting_path`
```

```
152: (4)                Notes
153: (4)                -----
154: (4)                The residual network :samp:`R` from an input graph :samp:`G` has the
155: (4)                same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
156: (4)                of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
157: (4)                self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
158: (4)                in :samp:`G`.
159: (4)                For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
160: (4)                is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
161: (4)                in :samp:`G` or zero otherwise. If the capacity is infinite,
162: (4)                :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
163: (4)                that does not affect the solution of the problem. This value is stored in
164: (4)                :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
165: (4)                :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
166: (4)                satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
167: (4)                The flow value, defined as the total flow into :samp:`t`, the sink, is
168: (4)                stored in :samp:`R.graph['flow_value']`. If :samp:`cutoff` is not
169: (4)                specified, reachability to :samp:`t` using only edges :samp:`(u, v)` such
170: (4)                that :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
171: (4)                :samp:`s`-:samp:`t` cut.
172: (4)                Examples
173: (4)                --------
174: (4)                >>> from networkx.algorithms.flow import edmonds_karp
175: (4)                The functions that implement flow algorithms and output a residual
176: (4)                network, such as this one, are not imported to the base NetworkX
177: (4)                namespace, so you have to explicitly import them from the flow package.
178: (4)                >>> G = nx.DiGraph()
179: (4)                >>> G.add_edge("x", "a", capacity=3.0)
180: (4)                >>> G.add_edge("x", "b", capacity=1.0)
181: (4)                >>> G.add_edge("a", "c", capacity=3.0)
182: (4)                >>> G.add_edge("b", "c", capacity=5.0)
183: (4)                >>> G.add_edge("b", "d", capacity=4.0)
184: (4)                >>> G.add_edge("d", "e", capacity=2.0)
185: (4)                >>> G.add_edge("c", "y", capacity=2.0)
186: (4)                >>> G.add_edge("e", "y", capacity=3.0)
187: (4)                >>> R = edmonds_karp(G, "x", "y")
188: (4)                >>> flow_value = nx.maximum_flow_value(G, "x", "y")
189: (4)                >>> flow_value
190: (4)                3.0
191: (4)                >>> flow_value == R.graph["flow_value"]
192: (4)                True
193: (4)                """
194: (4)                R = edmonds_karp_impl(G, s, t, capacity, residual, cutoff)
195: (4)                R.graph["algorithm"] = "edmonds_karp"
196: (4)                nx._clear_cache(R)
197: (4)                return R

----------------------------------------

File 241 - maxflow.py:

1: (0)              """
2: (0)              Maximum flow (and minimum cut) algorithms on capacitated graphs.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from .boykovkolmogorov import boykov_kolmogorov
6: (0)              from .dinitz_alg import dinitz
7: (0)              from .edmondskarp import edmonds_karp
8: (0)              from .preflowpush import preflow_push
9: (0)              from .shortestaugmentingpath import shortest_augmenting_path
10: (0)             from .utils import build_flow_dict
11: (0)             default_flow_func = preflow_push
12: (0)             __all__ = ["maximum_flow", "maximum_flow_value", "minimum_cut",
"minimum_cut_value"]
13: (0)             @nx._dispatchable(graphs="flowG", edge_attrs={"capacity": float("inf")})
14: (0)             def maximum_flow(flowG, _s, _t, capacity="capacity", flow_func=None,
**kwargs):
15: (4)                 """Find a maximum single-commodity flow.
16: (4)                 Parameters
```

```
(4)                 ----------
(4)                 flowG : NetworkX graph
(8)                     Edges of the graph are expected to have an attribute called
(8)                     'capacity'. If this attribute is not present, the edge is
(8)                     considered to have infinite capacity.
(4)                 _s : node
(8)                     Source node for the flow.
(4)                 _t : node
(8)                     Sink node for the flow.
(4)                 capacity : string
(8)                     Edges of the graph G are expected to have an attribute capacity
(8)                     that indicates how much flow the edge can support. If this
(8)                     attribute is not present, the edge is considered to have
(8)                     infinite capacity. Default value: 'capacity'.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes
(8)                     in a capacitated graph. The function has to accept at least three
(8)                     parameters: a Graph or Digraph, a source node, and a target node.
(8)                     And return a residual network that follows NetworkX conventions
(8)                     (see Notes). If flow_func is None, the default maximum
(8)                     flow function (:meth:`preflow_push`) is used. See below for
(8)                     alternative algorithms. The choice of the default function may change
(8)                     from version to version and should not be relied on. Default value:
(8)                     None.
(4)                 kwargs : Any other keyword parameter is passed to the function that
(8)                     computes the maximum flow.
(4)                 Returns
(4)                 -------
(4)                 flow_value : integer, float
(8)                     Value of the maximum flow, i.e., net outflow from the source.
(4)                 flow_dict : dict
(8)                     A dictionary containing the value of the flow that went through
(8)                     each edge.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     The algorithm does not support MultiGraph and MultiDiGraph. If
(8)                     the input graph is an instance of one of these two classes, a
(8)                     NetworkXError is raised.
(4)                 NetworkXUnbounded
(8)                     If the graph has a path of infinite capacity, the value of a
(8)                     feasible flow on the graph is unbounded above and the function
(8)                     raises a NetworkXUnbounded.
(4)                 See also
(4)                 --------
(4)                 :meth:`maximum_flow_value`
(4)                 :meth:`minimum_cut`
(4)                 :meth:`minimum_cut_value`
(4)                 :meth:`edmonds_karp`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 Notes
(4)                 -----
(4)                 The function used in the flow_func parameter has to return a residual
(4)                 network that follows NetworkX conventions:
(4)                 The residual network :samp:`R` from an input graph :samp:`G` has the
(4)                 same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
(4)                 of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
(4)                 self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
(4)                 in :samp:`G`.
(4)                 For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
(4)                 is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
(4)                 in :samp:`G` or zero otherwise. If the capacity is infinite,
(4)                 :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
(4)                 that does not affect the solution of the problem. This value is stored in
(4)                 :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
(4)                 :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
(4)                 satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
(4)                 The flow value, defined as the total flow into :samp:`t`, the sink, is
```

```
 86: (4)                 stored in :samp:`R.graph['flow_value']`. Reachability to :samp:`t` using
 87: (4)                 only edges :samp:`(u, v)` such that
 88: (4)                 :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
 89: (4)                 :samp:`s`-:samp:`t` cut.
 90: (4)                 Specific algorithms may store extra data in :samp:`R`.
 91: (4)                 The function should supports an optional boolean parameter value_only.
 When
 92: (4)                 True, it can optionally terminate the algorithm as soon as the maximum
 flow
 93: (4)                 value and the minimum cut can be determined.
 94: (4)                 Examples
 95: (4)                 --------
 96: (4)                 >>> G = nx.DiGraph()
 97: (4)                 >>> G.add_edge("x", "a", capacity=3.0)
 98: (4)                 >>> G.add_edge("x", "b", capacity=1.0)
 99: (4)                 >>> G.add_edge("a", "c", capacity=3.0)
 100: (4)                >>> G.add_edge("b", "c", capacity=5.0)
 101: (4)                >>> G.add_edge("b", "d", capacity=4.0)
 102: (4)                >>> G.add_edge("d", "e", capacity=2.0)
 103: (4)                >>> G.add_edge("c", "y", capacity=2.0)
 104: (4)                >>> G.add_edge("e", "y", capacity=3.0)
 105: (4)                maximum_flow returns both the value of the maximum flow and a
 106: (4)                dictionary with all flows.
 107: (4)                >>> flow_value, flow_dict = nx.maximum_flow(G, "x", "y")
 108: (4)                >>> flow_value
 109: (4)                3.0
 110: (4)                >>> print(flow_dict["x"]["b"])
 111: (4)                1.0
 112: (4)                You can also use alternative algorithms for computing the
 113: (4)                maximum flow by using the flow_func parameter.
 114: (4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
 115: (4)                >>> flow_value == nx.maximum_flow(G, "x", "y",
 flow_func=shortest_augmenting_path)[0]
 116: (4)                True
 117: (4)                """
 118: (4)                if flow_func is None:
 119: (8)                    if kwargs:
 120: (12)                       raise nx.NetworkXError(
 121: (16)                           "You have to explicitly set a flow_func if"
 122: (16)                           " you need to pass parameters via kwargs."
 123: (12)                       )
 124: (8)                    flow_func = default_flow_func
 125: (4)                if not callable(flow_func):
 126: (8)                    raise nx.NetworkXError("flow_func has to be callable.")
 127: (4)                R = flow_func(flowG, _s, _t, capacity=capacity, value_only=False,
 **kwargs)
 128: (4)                flow_dict = build_flow_dict(flowG, R)
 129: (4)                return (R.graph["flow_value"], flow_dict)
 130: (0)            @nx._dispatchable(graphs="flowG", edge_attrs={"capacity": float("inf")})
 131: (0)            def maximum_flow_value(flowG, _s, _t, capacity="capacity", flow_func=None,
 **kwargs):
 132: (4)                """Find the value of maximum single-commodity flow.
 133: (4)                Parameters
 134: (4)                ----------
 135: (4)                flowG : NetworkX graph
 136: (8)                    Edges of the graph are expected to have an attribute called
 137: (8)                    'capacity'. If this attribute is not present, the edge is
 138: (8)                    considered to have infinite capacity.
 139: (4)                _s : node
 140: (8)                    Source node for the flow.
 141: (4)                _t : node
 142: (8)                    Sink node for the flow.
 143: (4)                capacity : string
 144: (8)                    Edges of the graph G are expected to have an attribute capacity
 145: (8)                    that indicates how much flow the edge can support. If this
 146: (8)                    attribute is not present, the edge is considered to have
 147: (8)                    infinite capacity. Default value: 'capacity'.
 148: (4)                flow_func : function
 149: (8)                    A function for computing the maximum flow among a pair of nodes
```

```
(8)                 in a capacitated graph. The function has to accept at least three
(8)                 parameters: a Graph or Digraph, a source node, and a target node.
(8)                 And return a residual network that follows NetworkX conventions
(8)                 (see Notes). If flow_func is None, the default maximum
(8)                 flow function (:meth:`preflow_push`) is used. See below for
(8)                 alternative algorithms. The choice of the default function may change
(8)                 from version to version and should not be relied on. Default value:
(8)                 None.
(4)             kwargs : Any other keyword parameter is passed to the function that
(8)                 computes the maximum flow.
(4)             Returns
(4)             -------
(4)             flow_value : integer, float
(8)                 Value of the maximum flow, i.e., net outflow from the source.
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 The algorithm does not support MultiGraph and MultiDiGraph. If
(8)                 the input graph is an instance of one of these two classes, a
(8)                 NetworkXError is raised.
(4)             NetworkXUnbounded
(8)                 If the graph has a path of infinite capacity, the value of a
(8)                 feasible flow on the graph is unbounded above and the function
(8)                 raises a NetworkXUnbounded.
(4)             See also
(4)             --------
(4)             :meth:`maximum_flow`
(4)             :meth:`minimum_cut`
(4)             :meth:`minimum_cut_value`
(4)             :meth:`edmonds_karp`
(4)             :meth:`preflow_push`
(4)             :meth:`shortest_augmenting_path`
(4)             Notes
(4)             -----
(4)             The function used in the flow_func parameter has to return a residual
(4)             network that follows NetworkX conventions:
(4)             The residual network :samp:`R` from an input graph :samp:`G` has the
(4)             same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
(4)             of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
(4)             self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
(4)             in :samp:`G`.
(4)             For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
(4)             is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
(4)             in :samp:`G` or zero otherwise. If the capacity is infinite,
(4)             :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
(4)             that does not affect the solution of the problem. This value is stored in
(4)             :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
(4)             :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
(4)             satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
(4)             The flow value, defined as the total flow into :samp:`t`, the sink, is
(4)             stored in :samp:`R.graph['flow_value']`. Reachability to :samp:`t` using
(4)             only edges :samp:`(u, v)` such that
(4)             :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
(4)             :samp:`s`-:samp:`t` cut.
(4)             Specific algorithms may store extra data in :samp:`R`.
(4)             The function should supports an optional boolean parameter value_only.
When
(4)             True, it can optionally terminate the algorithm as soon as the maximum
flow
(4)             value and the minimum cut can be determined.
(4)             Examples
(4)             --------
(4)             >>> G = nx.DiGraph()
(4)             >>> G.add_edge("x", "a", capacity=3.0)
(4)             >>> G.add_edge("x", "b", capacity=1.0)
(4)             >>> G.add_edge("a", "c", capacity=3.0)
(4)             >>> G.add_edge("b", "c", capacity=5.0)
(4)             >>> G.add_edge("b", "d", capacity=4.0)
(4)             >>> G.add_edge("d", "e", capacity=2.0)
```

```
(4)                 >>> G.add_edge("c", "y", capacity=2.0)
(4)                 >>> G.add_edge("e", "y", capacity=3.0)
(4)                 maximum_flow_value computes only the value of the
(4)                 maximum flow:
(4)                 >>> flow_value = nx.maximum_flow_value(G, "x", "y")
(4)                 >>> flow_value
(4)                 3.0
(4)                 You can also use alternative algorithms for computing the
(4)                 maximum flow by using the flow_func parameter.
(4)                 >>> from networkx.algorithms.flow import shortest_augmenting_path
(4)                 >>> flow_value == nx.maximum_flow_value(G, "x", "y",
flow_func=shortest_augmenting_path)
(4)                 True
(4)                 """
(4)                 if flow_func is None:
(8)                     if kwargs:
(12)                        raise nx.NetworkXError(
(16)                            "You have to explicitly set a flow_func if"
(16)                            " you need to pass parameters via kwargs."
(12)                        )
(8)                     flow_func = default_flow_func
(4)                 if not callable(flow_func):
(8)                     raise nx.NetworkXError("flow_func has to be callable.")
(4)                 R = flow_func(flowG, _s, _t, capacity=capacity, value_only=True, **kwargs)
(4)                 return R.graph["flow_value"]
(0)             @nx._dispatchable(graphs="flowG", edge_attrs={"capacity": float("inf")})
(0)             def minimum_cut(flowG, _s, _t, capacity="capacity", flow_func=None, **kwargs):
(4)                 """Compute the value and the node partition of a minimum (s, t)-cut.
(4)                 Use the max-flow min-cut theorem, i.e., the capacity of a minimum
(4)                 capacity cut is equal to the flow value of a maximum flow.
(4)                 Parameters
(4)                 ----------
(4)                 flowG : NetworkX graph
(8)                     Edges of the graph are expected to have an attribute called
(8)                     'capacity'. If this attribute is not present, the edge is
(8)                     considered to have infinite capacity.
(4)                 _s : node
(8)                     Source node for the flow.
(4)                 _t : node
(8)                     Sink node for the flow.
(4)                 capacity : string
(8)                     Edges of the graph G are expected to have an attribute capacity
(8)                     that indicates how much flow the edge can support. If this
(8)                     attribute is not present, the edge is considered to have
(8)                     infinite capacity. Default value: 'capacity'.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes
(8)                     in a capacitated graph. The function has to accept at least three
(8)                     parameters: a Graph or Digraph, a source node, and a target node.
(8)                     And return a residual network that follows NetworkX conventions
(8)                     (see Notes). If flow_func is None, the default maximum
(8)                     flow function (:meth:`preflow_push`) is used. See below for
(8)                     alternative algorithms. The choice of the default function may change
(8)                     from version to version and should not be relied on. Default value:
(8)                     None.
(4)                 kwargs : Any other keyword parameter is passed to the function that
(8)                     computes the maximum flow.
(4)                 Returns
(4)                 -------
(4)                 cut_value : integer, float
(8)                     Value of the minimum cut.
(4)                 partition : pair of node sets
(8)                     A partitioning of the nodes that defines a minimum cut.
(4)                 Raises
(4)                 ------
(4)                 NetworkXUnbounded
(8)                     If the graph has a path of infinite capacity, all cuts have
(8)                     infinite capacity and the function raises a NetworkXError.
(4)                 See also
```

```
285: (4)                --------
286: (4)                :meth:`maximum_flow`
287: (4)                :meth:`maximum_flow_value`
288: (4)                :meth:`minimum_cut_value`
289: (4)                :meth:`edmonds_karp`
290: (4)                :meth:`preflow_push`
291: (4)                :meth:`shortest_augmenting_path`
292: (4)                Notes
293: (4)                -----
294: (4)                The function used in the flow_func parameter has to return a residual
295: (4)                network that follows NetworkX conventions:
296: (4)                The residual network :samp:`R` from an input graph :samp:`G` has the
297: (4)                same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
298: (4)                of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
299: (4)                self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
300: (4)                in :samp:`G`.
301: (4)                For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
302: (4)                is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
303: (4)                in :samp:`G` or zero otherwise. If the capacity is infinite,
304: (4)                :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
305: (4)                that does not affect the solution of the problem. This value is stored in
306: (4)                :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
307: (4)                :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
308: (4)                satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
309: (4)                The flow value, defined as the total flow into :samp:`t`, the sink, is
310: (4)                stored in :samp:`R.graph['flow_value']`. Reachability to :samp:`t` using
311: (4)                only edges :samp:`(u, v)` such that
312: (4)                :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
313: (4)                :samp:`s`-:samp:`t` cut.
314: (4)                Specific algorithms may store extra data in :samp:`R`.
315: (4)                The function should supports an optional boolean parameter value_only.
When
316: (4)                True, it can optionally terminate the algorithm as soon as the maximum
flow
317: (4)                value and the minimum cut can be determined.
318: (4)                Examples
319: (4)                --------
320: (4)                >>> G = nx.DiGraph()
321: (4)                >>> G.add_edge("x", "a", capacity=3.0)
322: (4)                >>> G.add_edge("x", "b", capacity=1.0)
323: (4)                >>> G.add_edge("a", "c", capacity=3.0)
324: (4)                >>> G.add_edge("b", "c", capacity=5.0)
325: (4)                >>> G.add_edge("b", "d", capacity=4.0)
326: (4)                >>> G.add_edge("d", "e", capacity=2.0)
327: (4)                >>> G.add_edge("c", "y", capacity=2.0)
328: (4)                >>> G.add_edge("e", "y", capacity=3.0)
329: (4)                minimum_cut computes both the value of the
330: (4)                minimum cut and the node partition:
331: (4)                >>> cut_value, partition = nx.minimum_cut(G, "x", "y")
332: (4)                >>> reachable, non_reachable = partition
333: (4)                'partition' here is a tuple with the two sets of nodes that define
334: (4)                the minimum cut. You can compute the cut set of edges that induce
335: (4)                the minimum cut as follows:
336: (4)                >>> cutset = set()
337: (4)                >>> for u, nbrs in ((n, G[n]) for n in reachable):
338: (4)                ...     cutset.update((u, v) for v in nbrs if v in non_reachable)
339: (4)                >>> print(sorted(cutset))
340: (4)                [('c', 'y'), ('x', 'b')]
341: (4)                >>> cut_value == sum(G.edges[u, v]["capacity"] for (u, v) in cutset)
342: (4)                True
343: (4)                You can also use alternative algorithms for computing the
344: (4)                minimum cut by using the flow_func parameter.
345: (4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
346: (4)                >>> cut_value == nx.minimum_cut(G, "x", "y",
flow_func=shortest_augmenting_path)[0]
347: (4)                True
348: (4)                """
349: (4)                if flow_func is None:
350: (8)                    if kwargs:
```

```
(12)                        raise nx.NetworkXError(
(16)                            "You have to explicitly set a flow_func if"
(16)                            " you need to pass parameters via kwargs."
(12)                        )
(8)                     flow_func = default_flow_func
(4)                 if not callable(flow_func):
(8)                     raise nx.NetworkXError("flow_func has to be callable.")
(4)                 if kwargs.get("cutoff") is not None and flow_func is preflow_push:
(8)                     raise nx.NetworkXError("cutoff should not be specified.")
(4)                 R = flow_func(flowG, _s, _t, capacity=capacity, value_only=True, **kwargs)
(4)                 cutset = [(u, v, d) for u, v, d in R.edges(data=True) if d["flow"] ==
d["capacity"]]
(4)                 R.remove_edges_from(cutset)
(4)                 non_reachable = set(dict(nx.shortest_path_length(R, target=_t)))
(4)                 partition = (set(flowG) - non_reachable, non_reachable)
(4)                 if cutset is not None:
(8)                     R.add_edges_from(cutset)
(4)                 return (R.graph["flow_value"], partition)
(0)             @nx._dispatchable(graphs="flowG", edge_attrs={"capacity": float("inf")})
(0)             def minimum_cut_value(flowG, _s, _t, capacity="capacity", flow_func=None,
**kwargs):
(4)                 """Compute the value of a minimum (s, t)-cut.
(4)                 Use the max-flow min-cut theorem, i.e., the capacity of a minimum
(4)                 capacity cut is equal to the flow value of a maximum flow.
(4)                 Parameters
(4)                 ----------
(4)                 flowG : NetworkX graph
(8)                     Edges of the graph are expected to have an attribute called
(8)                     'capacity'. If this attribute is not present, the edge is
(8)                     considered to have infinite capacity.
(4)                 _s : node
(8)                     Source node for the flow.
(4)                 _t : node
(8)                     Sink node for the flow.
(4)                 capacity : string
(8)                     Edges of the graph G are expected to have an attribute capacity
(8)                     that indicates how much flow the edge can support. If this
(8)                     attribute is not present, the edge is considered to have
(8)                     infinite capacity. Default value: 'capacity'.
(4)                 flow_func : function
(8)                     A function for computing the maximum flow among a pair of nodes
(8)                     in a capacitated graph. The function has to accept at least three
(8)                     parameters: a Graph or Digraph, a source node, and a target node.
(8)                     And return a residual network that follows NetworkX conventions
(8)                     (see Notes). If flow_func is None, the default maximum
(8)                     flow function (:meth:`preflow_push`) is used. See below for
(8)                     alternative algorithms. The choice of the default function may change
(8)                     from version to version and should not be relied on. Default value:
(8)                     None.
(4)                 kwargs : Any other keyword parameter is passed to the function that
(8)                     computes the maximum flow.
(4)                 Returns
(4)                 -------
(4)                 cut_value : integer, float
(8)                     Value of the minimum cut.
(4)                 Raises
(4)                 ------
(4)                 NetworkXUnbounded
(8)                     If the graph has a path of infinite capacity, all cuts have
(8)                     infinite capacity and the function raises a NetworkXError.
(4)                 See also
(4)                 --------
(4)                 :meth:`maximum_flow`
(4)                 :meth:`maximum_flow_value`
(4)                 :meth:`minimum_cut`
(4)                 :meth:`edmonds_karp`
(4)                 :meth:`preflow_push`
(4)                 :meth:`shortest_augmenting_path`
(4)                 Notes
```

```
418: (4)                -----
419: (4)                The function used in the flow_func parameter has to return a residual
420: (4)                network that follows NetworkX conventions:
421: (4)                The residual network :samp:`R` from an input graph :samp:`G` has the
422: (4)                same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
423: (4)                of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
424: (4)                self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
425: (4)                in :samp:`G`.
426: (4)                For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
427: (4)                is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
428: (4)                in :samp:`G` or zero otherwise. If the capacity is infinite,
429: (4)                :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
430: (4)                that does not affect the solution of the problem. This value is stored in
431: (4)                :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
432: (4)                :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
433: (4)                satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
434: (4)                The flow value, defined as the total flow into :samp:`t`, the sink, is
435: (4)                stored in :samp:`R.graph['flow_value']`. Reachability to :samp:`t` using
436: (4)                only edges :samp:`(u, v)` such that
437: (4)                :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
438: (4)                :samp:`s`-:samp:`t` cut.
439: (4)                Specific algorithms may store extra data in :samp:`R`.
440: (4)                The function should supports an optional boolean parameter value_only.
When
441: (4)                True, it can optionally terminate the algorithm as soon as the maximum
flow
442: (4)                value and the minimum cut can be determined.
443: (4)                Examples
444: (4)                --------
445: (4)                >>> G = nx.DiGraph()
446: (4)                >>> G.add_edge("x", "a", capacity=3.0)
447: (4)                >>> G.add_edge("x", "b", capacity=1.0)
448: (4)                >>> G.add_edge("a", "c", capacity=3.0)
449: (4)                >>> G.add_edge("b", "c", capacity=5.0)
450: (4)                >>> G.add_edge("b", "d", capacity=4.0)
451: (4)                >>> G.add_edge("d", "e", capacity=2.0)
452: (4)                >>> G.add_edge("c", "y", capacity=2.0)
453: (4)                >>> G.add_edge("e", "y", capacity=3.0)
454: (4)                minimum_cut_value computes only the value of the
455: (4)                minimum cut:
456: (4)                >>> cut_value = nx.minimum_cut_value(G, "x", "y")
457: (4)                >>> cut_value
458: (4)                3.0
459: (4)                You can also use alternative algorithms for computing the
460: (4)                minimum cut by using the flow_func parameter.
461: (4)                >>> from networkx.algorithms.flow import shortest_augmenting_path
462: (4)                >>> cut_value == nx.minimum_cut_value(G, "x", "y",
flow_func=shortest_augmenting_path)
463: (4)                True
464: (4)                """
465: (4)                if flow_func is None:
466: (8)                    if kwargs:
467: (12)                       raise nx.NetworkXError(
468: (16)                           "You have to explicitly set a flow_func if"
469: (16)                           " you need to pass parameters via kwargs."
470: (12)                       )
471: (8)                    flow_func = default_flow_func
472: (4)                if not callable(flow_func):
473: (8)                    raise nx.NetworkXError("flow_func has to be callable.")
474: (4)                if kwargs.get("cutoff") is not None and flow_func is preflow_push:
475: (8)                    raise nx.NetworkXError("cutoff should not be specified.")
476: (4)                R = flow_func(flowG, _s, _t, capacity=capacity, value_only=True, **kwargs)
477: (4)                return R.graph["flow_value"]
```

File 242 - gomory_hu.py:

```
1: (0)              """
```

```
2: (0)              Gomory-Hu tree of undirected Graphs.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import not_implemented_for
6: (0)              from .edmondskarp import edmonds_karp
7: (0)              from .utils import build_residual_network
8: (0)              default_flow_func = edmonds_karp
9: (0)              __all__ = ["gomory_hu_tree"]
10: (0)             @not_implemented_for("directed")
11: (0)             @nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
12: (0)             def gomory_hu_tree(G, capacity="capacity", flow_func=None):
13: (4)                 r"""Returns the Gomory-Hu tree of an undirected graph G.
14: (4)                 A Gomory-Hu tree of an undirected graph with capacities is a
15: (4)                 weighted tree that represents the minimum s-t cuts for all s-t
16: (4)                 pairs in the graph.
17: (4)                 It only requires `n-1` minimum cut computations instead of the
18: (4)                 obvious `n(n-1)/2`. The tree represents all s-t cuts as the
19: (4)                 minimum cut value among any pair of nodes is the minimum edge
20: (4)                 weight in the shortest path between the two nodes in the
21: (4)                 Gomory-Hu tree.
22: (4)                 The Gomory-Hu tree also has the property that removing the
23: (4)                 edge with the minimum weight in the shortest path between
24: (4)                 any two nodes leaves two connected components that form
25: (4)                 a partition of the nodes in G that defines the minimum s-t
26: (4)                 cut.
27: (4)                 See Examples section below for details.
28: (4)                 Parameters
29: (4)                 ----------
30: (4)                 G : NetworkX graph
31: (8)                     Undirected graph
32: (4)                 capacity : string
33: (8)                     Edges of the graph G are expected to have an attribute capacity
34: (8)                     that indicates how much flow the edge can support. If this
35: (8)                     attribute is not present, the edge is considered to have
36: (8)                     infinite capacity. Default value: 'capacity'.
37: (4)                 flow_func : function
38: (8)                     Function to perform the underlying flow computations. Default value
39: (8)                     :func:`edmonds_karp`. This function performs better in sparse graphs
40: (8)                     with right tailed degree distributions.
41: (8)                     :func:`shortest_augmenting_path` will perform better in denser
42: (8)                     graphs.
43: (4)                 Returns
44: (4)                 -------
45: (4)                 Tree : NetworkX graph
46: (8)                     A NetworkX graph representing the Gomory-Hu tree of the input graph.
47: (4)                 Raises
48: (4)                 ------
49: (4)                 NetworkXNotImplemented
50: (8)                     Raised if the input graph is directed.
51: (4)                 NetworkXError
52: (8)                     Raised if the input graph is an empty Graph.
53: (4)                 Examples
54: (4)                 --------
55: (4)                 >>> G = nx.karate_club_graph()
56: (4)                 >>> nx.set_edge_attributes(G, 1, "capacity")
57: (4)                 >>> T = nx.gomory_hu_tree(G)
58: (4)                 >>> # The value of the minimum cut between any pair
59: (4)                 ... # of nodes in G is the minimum edge weight in the
60: (4)                 ... # shortest path between the two nodes in the
61: (4)                 ... # Gomory-Hu tree.
62: (4)                 ... def minimum_edge_weight_in_shortest_path(T, u, v):
63: (4)                 ...     path = nx.shortest_path(T, u, v, weight="weight")
64: (4)                 ...     return min((T[u][v]["weight"], (u, v)) for (u, v) in zip(path,
path[1:]))
65: (4)                 >>> u, v = 0, 33
66: (4)                 >>> cut_value, edge = minimum_edge_weight_in_shortest_path(T, u, v)
67: (4)                 >>> cut_value
68: (4)                 10
69: (4)                 >>> nx.minimum_cut_value(G, u, v)
```

```
(4)     10
(4)     >>> # The Gomory-Hu tree also has the property that removing the
(4)     ... # edge with the minimum weight in the shortest path between
(4)     ... # any two nodes leaves two connected components that form
(4)     ... # a partition of the nodes in G that defines the minimum s-t
(4)     ... # cut.
(4)     ... cut_value, edge = minimum_edge_weight_in_shortest_path(T, u, v)
(4)     >>> T.remove_edge(*edge)
(4)     >>> U, V = list(nx.connected_components(T))
(4)     >>> # Thus U and V form a partition that defines a minimum cut
(4)     ... # between u and v in G. You can compute the edge cut set,
(4)     ... # that is, the set of edges that if removed from G will
(4)     ... # disconnect u from v in G, with this information:
(4)     ... cutset = set()
(4)     >>> for x, nbrs in ((n, G[n]) for n in U):
(4)     ...     cutset.update((x, y) for y in nbrs if y in V)
(4)     >>> # Because we have set the capacities of all edges to 1
(4)     ... # the cutset contains ten edges
(4)     ... len(cutset)
(4)     10
(4)     >>> # You can use any maximum flow algorithm for the underlying
(4)     ... # flow computations using the argument flow_func
(4)     ... from networkx.algorithms import flow
(4)     >>> T = nx.gomory_hu_tree(G, flow_func=flow.boykov_kolmogorov)
(4)     >>> cut_value, edge = minimum_edge_weight_in_shortest_path(T, u, v)
(4)     >>> cut_value
(4)     10
(4)     >>> nx.minimum_cut_value(G, u, v, flow_func=flow.boykov_kolmogorov)
(4)     10
(4)     Notes
(4)     -----
(4)     This implementation is based on Gusfield approach [1]_ to compute
(4)     Gomory-Hu trees, which does not require node contractions and has
(4)     the same computational complexity than the original method.
(4)     See also
(4)     --------
(4)     :func:`minimum_cut`
(4)     :func:`maximum_flow`
(4)     References
(4)     ----------
(4)     .. [1] Gusfield D: Very simple methods for all pairs network flow
analysis.
(11)            SIAM J Comput 19(1):143-155, 1990.
(4)     """
(4)     if flow_func is None:
(8)         flow_func = default_flow_func
(4)     if len(G) == 0:  # empty graph
(8)         msg = "Empty Graph does not have a Gomory-Hu tree representation"
(8)         raise nx.NetworkXError(msg)
(4)     tree = {}
(4)     labels = {}
(4)     iter_nodes = iter(G)
(4)     root = next(iter_nodes)
(4)     for n in iter_nodes:
(8)         tree[n] = root
(4)     R = build_residual_network(G, capacity)
(4)     for source in tree:
(8)         target = tree[source]
(8)         cut_value, partition = nx.minimum_cut(
(12)            G, source, target, capacity=capacity, flow_func=flow_func,
residual=R
(8)         )
(8)         labels[(source, target)] = cut_value
(8)         for node in partition[0]:
(12)            if node != source and node in tree and tree[node] == target:
(16)                tree[node] = source
(16)                labels[node, source] = labels.get((node, target), cut_value)
(8)         if target != root and tree[target] in partition[0]:
(12)            labels[source, tree[target]] = labels[target, tree[target]]
```

```
137: (12)                        labels[target, source] = cut_value
138: (12)                        tree[source] = tree[target]
139: (12)                        tree[target] = source
140: (4)                T = nx.Graph()
141: (4)                T.add_nodes_from(G)
142: (4)                T.add_weighted_edges_from(((u, v, labels[u, v]) for u, v in tree.items()))
143: (4)                return T
```

File 243 - mincost.py:

```
1: (0)              """
2: (0)              Minimum cost flow algorithms on directed connected graphs.
3: (0)              """
4: (0)              __all__ = ["min_cost_flow_cost", "min_cost_flow", "cost_of_flow",
"max_flow_min_cost"]
5: (0)              import networkx as nx
6: (0)              @nx._dispatchable(
7: (4)                  node_attrs="demand", edge_attrs={"capacity": float("inf"), "weight": 0}
8: (0)              )
9: (0)              def min_cost_flow_cost(G, demand="demand", capacity="capacity",
weight="weight"):
10: (4)                 r"""Find the cost of a minimum cost flow satisfying all demands in digraph
G.
11: (4)                 G is a digraph with edge costs and capacities and in which nodes
12: (4)                 have demand, i.e., they want to send or receive some amount of
13: (4)                 flow. A negative demand means that the node wants to send flow, a
14: (4)                 positive demand means that the node want to receive flow. A flow on
15: (4)                 the digraph G satisfies all demand if the net flow into each node
16: (4)                 is equal to the demand of that node.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : NetworkX graph
20: (8)                     DiGraph on which a minimum cost flow satisfying all demands is
21: (8)                     to be found.
22: (4)                 demand : string
23: (8)                     Nodes of the graph G are expected to have an attribute demand
24: (8)                     that indicates how much flow a node wants to send (negative
25: (8)                     demand) or receive (positive demand). Note that the sum of the
26: (8)                     demands should be 0 otherwise the problem in not feasible. If
27: (8)                     this attribute is not present, a node is considered to have 0
28: (8)                     demand. Default value: 'demand'.
29: (4)                 capacity : string
30: (8)                     Edges of the graph G are expected to have an attribute capacity
31: (8)                     that indicates how much flow the edge can support. If this
32: (8)                     attribute is not present, the edge is considered to have
33: (8)                     infinite capacity. Default value: 'capacity'.
34: (4)                 weight : string
35: (8)                     Edges of the graph G are expected to have an attribute weight
36: (8)                     that indicates the cost incurred by sending one unit of flow on
37: (8)                     that edge. If not present, the weight is considered to be 0.
38: (8)                     Default value: 'weight'.
39: (4)                 Returns
40: (4)                 -------
41: (4)                 flowCost : integer, float
42: (8)                     Cost of a minimum cost flow satisfying all demands.
43: (4)                 Raises
44: (4)                 ------
45: (4)                 NetworkXError
46: (8)                     This exception is raised if the input graph is not directed or
47: (8)                     not connected.
48: (4)                 NetworkXUnfeasible
49: (8)                     This exception is raised in the following situations:
50: (12)                        * The sum of the demands is not zero. Then, there is no
51: (14)                          flow satisfying all demands.
52: (12)                        * There is no flow satisfying all demand.
53: (4)                 NetworkXUnbounded
54: (8)                     This exception is raised if the digraph G has a cycle of
```

```
 55: (8)                     negative cost and infinite capacity. Then, the cost of a flow
 56: (8)                     satisfying all demands is unbounded below.
 57: (4)                 See also
 58: (4)                 --------
 59: (4)                 cost_of_flow, max_flow_min_cost, min_cost_flow, network_simplex
 60: (4)                 Notes
 61: (4)                 -----
 62: (4)                 This algorithm is not guaranteed to work if edge weights or demands
 63: (4)                 are floating point numbers (overflows and roundoff errors can
 64: (4)                 cause problems). As a workaround you can use integer numbers by
 65: (4)                 multiplying the relevant edge attributes by a convenient
 66: (4)                 constant factor (eg 100).
 67: (4)                 Examples
 68: (4)                 --------
 69: (4)                 A simple example of a min cost flow problem.
 70: (4)                 >>> G = nx.DiGraph()
 71: (4)                 >>> G.add_node("a", demand=-5)
 72: (4)                 >>> G.add_node("d", demand=5)
 73: (4)                 >>> G.add_edge("a", "b", weight=3, capacity=4)
 74: (4)                 >>> G.add_edge("a", "c", weight=6, capacity=10)
 75: (4)                 >>> G.add_edge("b", "d", weight=1, capacity=9)
 76: (4)                 >>> G.add_edge("c", "d", weight=2, capacity=5)
 77: (4)                 >>> flowCost = nx.min_cost_flow_cost(G)
 78: (4)                 >>> flowCost
 79: (4)                 24
 80: (4)                 """
 81: (4)                 return nx.network_simplex(G, demand=demand, capacity=capacity,
weight=weight)[0]
 82: (0)             @nx._dispatchable(
 83: (4)                 node_attrs="demand", edge_attrs={"capacity": float("inf"), "weight": 0}
 84: (0)             )
 85: (0)             def min_cost_flow(G, demand="demand", capacity="capacity", weight="weight"):
 86: (4)                 r"""Returns a minimum cost flow satisfying all demands in digraph G.
 87: (4)                 G is a digraph with edge costs and capacities and in which nodes
 88: (4)                 have demand, i.e., they want to send or receive some amount of
 89: (4)                 flow. A negative demand means that the node wants to send flow, a
 90: (4)                 positive demand means that the node want to receive flow. A flow on
 91: (4)                 the digraph G satisfies all demand if the net flow into each node
 92: (4)                 is equal to the demand of that node.
 93: (4)                 Parameters
 94: (4)                 ----------
 95: (4)                 G : NetworkX graph
 96: (8)                     DiGraph on which a minimum cost flow satisfying all demands is
 97: (8)                     to be found.
 98: (4)                 demand : string
 99: (8)                     Nodes of the graph G are expected to have an attribute demand
100: (8)                     that indicates how much flow a node wants to send (negative
101: (8)                     demand) or receive (positive demand). Note that the sum of the
102: (8)                     demands should be 0 otherwise the problem in not feasible. If
103: (8)                     this attribute is not present, a node is considered to have 0
104: (8)                     demand. Default value: 'demand'.
105: (4)                 capacity : string
106: (8)                     Edges of the graph G are expected to have an attribute capacity
107: (8)                     that indicates how much flow the edge can support. If this
108: (8)                     attribute is not present, the edge is considered to have
109: (8)                     infinite capacity. Default value: 'capacity'.
110: (4)                 weight : string
111: (8)                     Edges of the graph G are expected to have an attribute weight
112: (8)                     that indicates the cost incurred by sending one unit of flow on
113: (8)                     that edge. If not present, the weight is considered to be 0.
114: (8)                     Default value: 'weight'.
115: (4)                 Returns
116: (4)                 -------
117: (4)                 flowDict : dictionary
118: (8)                     Dictionary of dictionaries keyed by nodes such that
119: (8)                     flowDict[u][v] is the flow edge (u, v).
120: (4)                 Raises
121: (4)                 ------
122: (4)                 NetworkXError
```

```
123: (8)                    This exception is raised if the input graph is not directed or
124: (8)                    not connected.
125: (4)                NetworkXUnfeasible
126: (8)                    This exception is raised in the following situations:
127: (12)                       * The sum of the demands is not zero. Then, there is no
128: (14)                         flow satisfying all demands.
129: (12)                       * There is no flow satisfying all demand.
130: (4)                NetworkXUnbounded
131: (8)                    This exception is raised if the digraph G has a cycle of
132: (8)                    negative cost and infinite capacity. Then, the cost of a flow
133: (8)                    satisfying all demands is unbounded below.
134: (4)                See also
135: (4)                --------
136: (4)                cost_of_flow, max_flow_min_cost, min_cost_flow_cost, network_simplex
137: (4)                Notes
138: (4)                -----
139: (4)                This algorithm is not guaranteed to work if edge weights or demands
140: (4)                are floating point numbers (overflows and roundoff errors can
141: (4)                cause problems). As a workaround you can use integer numbers by
142: (4)                multiplying the relevant edge attributes by a convenient
143: (4)                constant factor (eg 100).
144: (4)                Examples
145: (4)                --------
146: (4)                A simple example of a min cost flow problem.
147: (4)                >>> G = nx.DiGraph()
148: (4)                >>> G.add_node("a", demand=-5)
149: (4)                >>> G.add_node("d", demand=5)
150: (4)                >>> G.add_edge("a", "b", weight=3, capacity=4)
151: (4)                >>> G.add_edge("a", "c", weight=6, capacity=10)
152: (4)                >>> G.add_edge("b", "d", weight=1, capacity=9)
153: (4)                >>> G.add_edge("c", "d", weight=2, capacity=5)
154: (4)                >>> flowDict = nx.min_cost_flow(G)
155: (4)                >>> flowDict
156: (4)                {'a': {'b': 4, 'c': 1}, 'd': {}, 'b': {'d': 4}, 'c': {'d': 1}}
157: (4)                """
158: (4)                return nx.network_simplex(G, demand=demand, capacity=capacity,
weight=weight)[1]
159: (0)            @nx._dispatchable(edge_attrs={"weight": 0})
160: (0)            def cost_of_flow(G, flowDict, weight="weight"):
161: (4)                """Compute the cost of the flow given by flowDict on graph G.
162: (4)                Note that this function does not check for the validity of the
163: (4)                flow flowDict. This function will fail if the graph G and the
164: (4)                flow don't have the same edge set.
165: (4)                Parameters
166: (4)                ----------
167: (4)                G : NetworkX graph
168: (8)                    DiGraph on which a minimum cost flow satisfying all demands is
169: (8)                    to be found.
170: (4)                weight : string
171: (8)                    Edges of the graph G are expected to have an attribute weight
172: (8)                    that indicates the cost incurred by sending one unit of flow on
173: (8)                    that edge. If not present, the weight is considered to be 0.
174: (8)                    Default value: 'weight'.
175: (4)                flowDict : dictionary
176: (8)                    Dictionary of dictionaries keyed by nodes such that
177: (8)                    flowDict[u][v] is the flow edge (u, v).
178: (4)                Returns
179: (4)                -------
180: (4)                cost : Integer, float
181: (8)                    The total cost of the flow. This is given by the sum over all
182: (8)                    edges of the product of the edge's flow and the edge's weight.
183: (4)                See also
184: (4)                --------
185: (4)                max_flow_min_cost, min_cost_flow, min_cost_flow_cost, network_simplex
186: (4)                Notes
187: (4)                -----
188: (4)                This algorithm is not guaranteed to work if edge weights or demands
189: (4)                are floating point numbers (overflows and roundoff errors can
190: (4)                cause problems). As a workaround you can use integer numbers by
```

```
(4)                 multiplying the relevant edge attributes by a convenient
(4)                 constant factor (eg 100).
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_node("a", demand=-5)
(4)                 >>> G.add_node("d", demand=5)
(4)                 >>> G.add_edge("a", "b", weight=3, capacity=4)
(4)                 >>> G.add_edge("a", "c", weight=6, capacity=10)
(4)                 >>> G.add_edge("b", "d", weight=1, capacity=9)
(4)                 >>> G.add_edge("c", "d", weight=2, capacity=5)
(4)                 >>> flowDict = nx.min_cost_flow(G)
(4)                 >>> flowDict
(4)                 {'a': {'b': 4, 'c': 1}, 'd': {}, 'b': {'d': 4}, 'c': {'d': 1}}
(4)                 >>> nx.cost_of_flow(G, flowDict)
(4)                 24
(4)                 """
(4)                 return sum((flowDict[u][v] * d.get(weight, 0) for u, v, d in
G.edges(data=True)))
(0)             @nx._dispatchable(edge_attrs={"capacity": float("inf"), "weight": 0})
(0)             def max_flow_min_cost(G, s, t, capacity="capacity", weight="weight"):
(4)                 """Returns a maximum (s, t)-flow of minimum cost.
(4)                 G is a digraph with edge costs and capacities. There is a source
(4)                 node s and a sink node t. This function finds a maximum flow from
(4)                 s to t whose total cost is minimized.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     DiGraph on which a minimum cost flow satisfying all demands is
(8)                     to be found.
(4)                 s: node label
(8)                     Source of the flow.
(4)                 t: node label
(8)                     Destination of the flow.
(4)                 capacity: string
(8)                     Edges of the graph G are expected to have an attribute capacity
(8)                     that indicates how much flow the edge can support. If this
(8)                     attribute is not present, the edge is considered to have
(8)                     infinite capacity. Default value: 'capacity'.
(4)                 weight: string
(8)                     Edges of the graph G are expected to have an attribute weight
(8)                     that indicates the cost incurred by sending one unit of flow on
(8)                     that edge. If not present, the weight is considered to be 0.
(8)                     Default value: 'weight'.
(4)                 Returns
(4)                 -------
(4)                 flowDict: dictionary
(8)                     Dictionary of dictionaries keyed by nodes such that
(8)                     flowDict[u][v] is the flow edge (u, v).
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     This exception is raised if the input graph is not directed or
(8)                     not connected.
(4)                 NetworkXUnbounded
(8)                     This exception is raised if there is an infinite capacity path
(8)                     from s to t in G. In this case there is no maximum flow. This
(8)                     exception is also raised if the digraph G has a cycle of
(8)                     negative cost and infinite capacity. Then, the cost of a flow
(8)                     is unbounded below.
(4)                 See also
(4)                 --------
(4)                 cost_of_flow, min_cost_flow, min_cost_flow_cost, network_simplex
(4)                 Notes
(4)                 -----
(4)                 This algorithm is not guaranteed to work if edge weights or demands
(4)                 are floating point numbers (overflows and roundoff errors can
(4)                 cause problems). As a workaround you can use integer numbers by
(4)                 multiplying the relevant edge attributes by a convenient
```

```
259: (4)                constant factor (eg 100).
260: (4)                Examples
261: (4)                --------
262: (4)                >>> G = nx.DiGraph()
263: (4)                >>> G.add_edges_from(
264: (4)                ...     [
265: (4)                ...         (1, 2, {"capacity": 12, "weight": 4}),
266: (4)                ...         (1, 3, {"capacity": 20, "weight": 6}),
267: (4)                ...         (2, 3, {"capacity": 6, "weight": -3}),
268: (4)                ...         (2, 6, {"capacity": 14, "weight": 1}),
269: (4)                ...         (3, 4, {"weight": 9}),
270: (4)                ...         (3, 5, {"capacity": 10, "weight": 5}),
271: (4)                ...         (4, 2, {"capacity": 19, "weight": 13}),
272: (4)                ...         (4, 5, {"capacity": 4, "weight": 0}),
273: (4)                ...         (5, 7, {"capacity": 28, "weight": 2}),
274: (4)                ...         (6, 5, {"capacity": 11, "weight": 1}),
275: (4)                ...         (6, 7, {"weight": 8}),
276: (4)                ...         (7, 4, {"capacity": 6, "weight": 6}),
277: (4)                ...     ]
278: (4)                ... )
279: (4)                >>> mincostFlow = nx.max_flow_min_cost(G, 1, 7)
280: (4)                >>> mincost = nx.cost_of_flow(G, mincostFlow)
281: (4)                >>> mincost
282: (4)                373
283: (4)                >>> from networkx.algorithms.flow import maximum_flow
284: (4)                >>> maxFlow = maximum_flow(G, 1, 7)[1]
285: (4)                >>> nx.cost_of_flow(G, maxFlow) >= mincost
286: (4)                True
287: (4)                >>> mincostFlowValue = sum((mincostFlow[u][7] for u in G.predecessors(7)))
- sum(
288: (4)                ...     (mincostFlow[7][v] for v in G.successors(7))
289: (4)                ... )
290: (4)                >>> mincostFlowValue == nx.maximum_flow_value(G, 1, 7)
291: (4)                True
292: (4)                """
293: (4)                maxFlow = nx.maximum_flow_value(G, s, t, capacity=capacity)
294: (4)                H = nx.DiGraph(G)
295: (4)                H.add_node(s, demand=-maxFlow)
296: (4)                H.add_node(t, demand=maxFlow)
297: (4)                return min_cost_flow(H, capacity=capacity, weight=weight)

----------------------------------------

File 244 - networksimplex.py:

1: (0)              """
2: (0)              Minimum cost flow algorithms on directed connected graphs.
3: (0)              """
4: (0)              __all__ = ["network_simplex"]
5: (0)              from itertools import chain, islice, repeat
6: (0)              from math import ceil, sqrt
7: (0)              import networkx as nx
8: (0)              from networkx.utils import not_implemented_for
9: (0)              class _DataEssentialsAndFunctions:
10: (4)                 def __init__(
11: (8)                     self, G, multigraph, demand="demand", capacity="capacity",
weight="weight"
12: (4)                 ):
13: (8)                     self.node_list = list(G)  # nodes
14: (8)                     self.node_indices = {u: i for i, u in enumerate(self.node_list)}  #
node indices
15: (8)                     self.node_demands = [
16: (12)                        G.nodes[u].get(demand, 0) for u in self.node_list
17: (8)                     ]  # node demands
18: (8)                     self.edge_sources = []  # edge sources
19: (8)                     self.edge_targets = []  # edge targets
20: (8)                     if multigraph:
21: (12)                        self.edge_keys = []  # edge keys
22: (8)                     self.edge_indices = {}  # edge indices
```

```
 23: (8)                     self.edge_capacities = []  # edge capacities
 24: (8)                     self.edge_weights = []  # edge weights
 25: (8)                     if not multigraph:
 26: (12)                        edges = G.edges(data=True)
 27: (8)                     else:
 28: (12)                        edges = G.edges(data=True, keys=True)
 29: (8)                     inf = float("inf")
 30: (8)                     edges = (e for e in edges if e[0] != e[1] and e[-1].get(capacity, inf)
 != 0)
 31: (8)                     for i, e in enumerate(edges):
 32: (12)                        self.edge_sources.append(self.node_indices[e[0]])
 33: (12)                        self.edge_targets.append(self.node_indices[e[1]])
 34: (12)                        if multigraph:
 35: (16)                            self.edge_keys.append(e[2])
 36: (12)                        self.edge_indices[e[:-1]] = i
 37: (12)                        self.edge_capacities.append(e[-1].get(capacity, inf))
 38: (12)                        self.edge_weights.append(e[-1].get(weight, 0))
 39: (8)                     self.edge_count = None  # number of edges
 40: (8)                     self.edge_flow = None  # edge flows
 41: (8)                     self.node_potentials = None  # node potentials
 42: (8)                     self.parent = None  # parent nodes
 43: (8)                     self.parent_edge = None  # edges to parents
 44: (8)                     self.subtree_size = None  # subtree sizes
 45: (8)                     self.next_node_dft = None  # next nodes in depth-first thread
 46: (8)                     self.prev_node_dft = None  # previous nodes in depth-first thread
 47: (8)                     self.last_descendent_dft = None  # last descendants in depth-first
 thread
 48: (8)                     self._spanning_tree_initialized = (
 49: (12)                        False  # False until initialize_spanning_tree() is called
 50: (8)                     )
 51: (4)                 def initialize_spanning_tree(self, n, faux_inf):
 52: (8)                     self.edge_count = len(self.edge_indices)  # number of edges
 53: (8)                     self.edge_flow = list(
 54: (12)                        chain(repeat(0, self.edge_count), (abs(d) for d in
 self.node_demands))
 55: (8)                     )  # edge flows
 56: (8)                     self.node_potentials = [
 57: (12)                        faux_inf if d <= 0 else -faux_inf for d in self.node_demands
 58: (8)                     ]  # node potentials
 59: (8)                     self.parent = list(chain(repeat(-1, n), [None]))  # parent nodes
 60: (8)                     self.parent_edge = list(
 61: (12)                        range(self.edge_count, self.edge_count + n)
 62: (8)                     )  # edges to parents
 63: (8)                     self.subtree_size = list(chain(repeat(1, n), [n + 1]))  # subtree
 sizes
 64: (8)                     self.next_node_dft = list(
 65: (12)                        chain(range(1, n), [-1, 0])
 66: (8)                     )  # next nodes in depth-first thread
 67: (8)                     self.prev_node_dft = list(range(-1, n))  # previous nodes in depth-
 first thread
 68: (8)                     self.last_descendent_dft = list(
 69: (12)                        chain(range(n), [n - 1])
 70: (8)                     )  # last descendants in depth-first thread
 71: (8)                     self._spanning_tree_initialized = True  # True only if all the
 assignments pass
 72: (4)                 def find_apex(self, p, q):
 73: (8)                     """
 74: (8)                     Find the lowest common ancestor of nodes p and q in the spanning tree.
 75: (8)                     """
 76: (8)                     size_p = self.subtree_size[p]
 77: (8)                     size_q = self.subtree_size[q]
 78: (8)                     while True:
 79: (12)                        while size_p < size_q:
 80: (16)                            p = self.parent[p]
 81: (16)                            size_p = self.subtree_size[p]
 82: (12)                        while size_p > size_q:
 83: (16)                            q = self.parent[q]
 84: (16)                            size_q = self.subtree_size[q]
 85: (12)                        if size_p == size_q:
```

```
(16)                            if p != q:
(20)                                p = self.parent[p]
(20)                                size_p = self.subtree_size[p]
(20)                                q = self.parent[q]
(20)                                size_q = self.subtree_size[q]
(16)                            else:
(20)                                return p
(4)                 def trace_path(self, p, w):
(8)                     """
(8)                     Returns the nodes and edges on the path from node p to its ancestor w.
(8)                     """
(8)                     Wn = [p]
(8)                     We = []
(8)                     while p != w:
(12)                        We.append(self.parent_edge[p])
(12)                        p = self.parent[p]
(12)                        Wn.append(p)
(8)                     return Wn, We
(4)                 def find_cycle(self, i, p, q):
(8)                     """
(8)                     Returns the nodes and edges on the cycle containing edge i == (p, q)
(8)                     when the latter is added to the spanning tree.
(8)                     The cycle is oriented in the direction from p to q.
(8)                     """
(8)                     w = self.find_apex(p, q)
(8)                     Wn, We = self.trace_path(p, w)
(8)                     Wn.reverse()
(8)                     We.reverse()
(8)                     if We != [i]:
(12)                        We.append(i)
(8)                     WnR, WeR = self.trace_path(q, w)
(8)                     del WnR[-1]
(8)                     Wn += WnR
(8)                     We += WeR
(8)                     return Wn, We
(4)                 def augment_flow(self, Wn, We, f):
(8)                     """
(8)                     Augment f units of flow along a cycle represented by Wn and We.
(8)                     """
(8)                     for i, p in zip(We, Wn):
(12)                        if self.edge_sources[i] == p:
(16)                            self.edge_flow[i] += f
(12)                        else:
(16)                            self.edge_flow[i] -= f
(4)                 def trace_subtree(self, p):
(8)                     """
(8)                     Yield the nodes in the subtree rooted at a node p.
(8)                     """
(8)                     yield p
(8)                     l = self.last_descendent_dft[p]
(8)                     while p != l:
(12)                        p = self.next_node_dft[p]
(12)                        yield p
(4)                 def remove_edge(self, s, t):
(8)                     """
(8)                     Remove an edge (s, t) where parent[t] == s from the spanning tree.
(8)                     """
(8)                     size_t = self.subtree_size[t]
(8)                     prev_t = self.prev_node_dft[t]
(8)                     last_t = self.last_descendent_dft[t]
(8)                     next_last_t = self.next_node_dft[last_t]
(8)                     self.parent[t] = None
(8)                     self.parent_edge[t] = None
(8)                     self.next_node_dft[prev_t] = next_last_t
(8)                     self.prev_node_dft[next_last_t] = prev_t
(8)                     self.next_node_dft[last_t] = t
(8)                     self.prev_node_dft[t] = last_t
(8)                     while s is not None:
(12)                        self.subtree_size[s] -= size_t
```

```
(12)                        if self.last_descendent_dft[s] == last_t:
(16)                            self.last_descendent_dft[s] = prev_t
(12)                        s = self.parent[s]
(4)                 def make_root(self, q):
(8)                     """
(8)                     Make a node q the root of its containing subtree.
(8)                     """
(8)                     ancestors = []
(8)                     while q is not None:
(12)                        ancestors.append(q)
(12)                        q = self.parent[q]
(8)                     ancestors.reverse()
(8)                     for p, q in zip(ancestors, islice(ancestors, 1, None)):
(12)                        size_p = self.subtree_size[p]
(12)                        last_p = self.last_descendent_dft[p]
(12)                        prev_q = self.prev_node_dft[q]
(12)                        last_q = self.last_descendent_dft[q]
(12)                        next_last_q = self.next_node_dft[last_q]
(12)                        self.parent[p] = q
(12)                        self.parent[q] = None
(12)                        self.parent_edge[p] = self.parent_edge[q]
(12)                        self.parent_edge[q] = None
(12)                        self.subtree_size[p] = size_p - self.subtree_size[q]
(12)                        self.subtree_size[q] = size_p
(12)                        self.next_node_dft[prev_q] = next_last_q
(12)                        self.prev_node_dft[next_last_q] = prev_q
(12)                        self.next_node_dft[last_q] = q
(12)                        self.prev_node_dft[q] = last_q
(12)                        if last_p == last_q:
(16)                            self.last_descendent_dft[p] = prev_q
(16)                            last_p = prev_q
(12)                        self.prev_node_dft[p] = last_q
(12)                        self.next_node_dft[last_q] = p
(12)                        self.next_node_dft[last_p] = q
(12)                        self.prev_node_dft[q] = last_p
(12)                        self.last_descendent_dft[q] = last_p
(4)                 def add_edge(self, i, p, q):
(8)                     """
(8)                     Add an edge (p, q) to the spanning tree where q is the root of a
subtree.
(8)                     """
(8)                     last_p = self.last_descendent_dft[p]
(8)                     next_last_p = self.next_node_dft[last_p]
(8)                     size_q = self.subtree_size[q]
(8)                     last_q = self.last_descendent_dft[q]
(8)                     self.parent[q] = p
(8)                     self.parent_edge[q] = i
(8)                     self.next_node_dft[last_p] = q
(8)                     self.prev_node_dft[q] = last_p
(8)                     self.prev_node_dft[next_last_p] = last_q
(8)                     self.next_node_dft[last_q] = next_last_p
(8)                     while p is not None:
(12)                        self.subtree_size[p] += size_q
(12)                        if self.last_descendent_dft[p] == last_p:
(16)                            self.last_descendent_dft[p] = last_q
(12)                        p = self.parent[p]
(4)                 def update_potentials(self, i, p, q):
(8)                     """
(8)                     Update the potentials of the nodes in the subtree rooted at a node
(8)                     q connected to its parent p by an edge i.
(8)                     """
(8)                     if q == self.edge_targets[i]:
(12)                        d = self.node_potentials[p] - self.edge_weights[i] -
self.node_potentials[q]
(8)                     else:
(12)                        d = self.node_potentials[p] + self.edge_weights[i] -
self.node_potentials[q]
(8)                     for q in self.trace_subtree(q):
(12)                        self.node_potentials[q] += d
```

```
221: (4)                def reduced_cost(self, i):
222: (8)                    """Returns the reduced cost of an edge i."""
223: (8)                    c = (
224: (12)                       self.edge_weights[i]
225: (12)                       - self.node_potentials[self.edge_sources[i]]
226: (12)                       + self.node_potentials[self.edge_targets[i]]
227: (8)                    )
228: (8)                    return c if self.edge_flow[i] == 0 else -c
229: (4)                def find_entering_edges(self):
230: (8)                    """Yield entering edges until none can be found."""
231: (8)                    if self.edge_count == 0:
232: (12)                       return
233: (8)                    B = int(ceil(sqrt(self.edge_count)))  # pivot block size
234: (8)                    M = (self.edge_count + B - 1) // B  # number of blocks needed to cover
all edges
235: (8)                    m = 0  # number of consecutive blocks without eligible
236: (8)                    f = 0  # first edge in block
237: (8)                    while m < M:
238: (12)                       l = f + B
239: (12)                       if l <= self.edge_count:
240: (16)                           edges = range(f, l)
241: (12)                       else:
242: (16)                           l -= self.edge_count
243: (16)                           edges = chain(range(f, self.edge_count), range(l))
244: (12)                       f = l
245: (12)                       i = min(edges, key=self.reduced_cost)
246: (12)                       c = self.reduced_cost(i)
247: (12)                       if c >= 0:
248: (16)                           m += 1
249: (12)                       else:
250: (16)                           if self.edge_flow[i] == 0:
251: (20)                               p = self.edge_sources[i]
252: (20)                               q = self.edge_targets[i]
253: (16)                           else:
254: (20)                               p = self.edge_targets[i]
255: (20)                               q = self.edge_sources[i]
256: (16)                           yield i, p, q
257: (16)                           m = 0
258: (4)                def residual_capacity(self, i, p):
259: (8)                    """Returns the residual capacity of an edge i in the direction away
260: (8)                    from its endpoint p.
261: (8)                    """
262: (8)                    return (
263: (12)                       self.edge_capacities[i] - self.edge_flow[i]
264: (12)                       if self.edge_sources[i] == p
265: (12)                       else self.edge_flow[i]
266: (8)                    )
267: (4)                def find_leaving_edge(self, Wn, We):
268: (8)                    """Returns the leaving edge in a cycle represented by Wn and We."""
269: (8)                    j, s = min(
270: (12)                       zip(reversed(We), reversed(Wn)),
271: (12)                       key=lambda i_p: self.residual_capacity(*i_p),
272: (8)                    )
273: (8)                    t = self.edge_targets[j] if self.edge_sources[j] == s else
self.edge_sources[j]
274: (8)                    return j, s, t
275: (0)            @not_implemented_for("undirected")
276: (0)            @nx._dispatchable(
277: (4)                node_attrs="demand", edge_attrs={"capacity": float("inf"), "weight": 0}
278: (0)            )
279: (0)            def network_simplex(G, demand="demand", capacity="capacity", weight="weight"):
280: (4)                r"""Find a minimum cost flow satisfying all demands in digraph G.
281: (4)                This is a primal network simplex algorithm that uses the leaving
282: (4)                arc rule to prevent cycling.
283: (4)                G is a digraph with edge costs and capacities and in which nodes
284: (4)                have demand, i.e., they want to send or receive some amount of
285: (4)                flow. A negative demand means that the node wants to send flow, a
286: (4)                positive demand means that the node want to receive flow. A flow on
287: (4)                the digraph G satisfies all demand if the net flow into each node
```

```
    (4)                 is equal to the demand of that node.
    (4)                 Parameters
    (4)                 ----------
    (4)                 G : NetworkX graph
    (8)                     DiGraph on which a minimum cost flow satisfying all demands is
    (8)                     to be found.
    (4)                 demand : string
    (8)                     Nodes of the graph G are expected to have an attribute demand
    (8)                     that indicates how much flow a node wants to send (negative
    (8)                     demand) or receive (positive demand). Note that the sum of the
    (8)                     demands should be 0 otherwise the problem in not feasible. If
    (8)                     this attribute is not present, a node is considered to have 0
    (8)                     demand. Default value: 'demand'.
    (4)                 capacity : string
    (8)                     Edges of the graph G are expected to have an attribute capacity
    (8)                     that indicates how much flow the edge can support. If this
    (8)                     attribute is not present, the edge is considered to have
    (8)                     infinite capacity. Default value: 'capacity'.
    (4)                 weight : string
    (8)                     Edges of the graph G are expected to have an attribute weight
    (8)                     that indicates the cost incurred by sending one unit of flow on
    (8)                     that edge. If not present, the weight is considered to be 0.
    (8)                     Default value: 'weight'.
    (4)                 Returns
    (4)                 -------
    (4)                 flowCost : integer, float
    (8)                     Cost of a minimum cost flow satisfying all demands.
    (4)                 flowDict : dictionary
    (8)                     Dictionary of dictionaries keyed by nodes such that
    (8)                     flowDict[u][v] is the flow edge (u, v).
    (4)                 Raises
    (4)                 ------
    (4)                 NetworkXError
    (8)                     This exception is raised if the input graph is not directed or
    (8)                     not connected.
    (4)                 NetworkXUnfeasible
    (8)                     This exception is raised in the following situations:
    (12)                        * The sum of the demands is not zero. Then, there is no
    (14)                          flow satisfying all demands.
    (12)                        * There is no flow satisfying all demand.
    (4)                 NetworkXUnbounded
    (8)                     This exception is raised if the digraph G has a cycle of
    (8)                     negative cost and infinite capacity. Then, the cost of a flow
    (8)                     satisfying all demands is unbounded below.
    (4)                 Notes
    (4)                 -----
    (4)                 This algorithm is not guaranteed to work if edge weights or demands
    (4)                 are floating point numbers (overflows and roundoff errors can
    (4)                 cause problems). As a workaround you can use integer numbers by
    (4)                 multiplying the relevant edge attributes by a convenient
    (4)                 constant factor (eg 100).
    (4)                 See also
    (4)                 --------
    (4)                 cost_of_flow, max_flow_min_cost, min_cost_flow, min_cost_flow_cost
    (4)                 Examples
    (4)                 --------
    (4)                 A simple example of a min cost flow problem.
    (4)                 >>> G = nx.DiGraph()
    (4)                 >>> G.add_node("a", demand=-5)
    (4)                 >>> G.add_node("d", demand=5)
    (4)                 >>> G.add_edge("a", "b", weight=3, capacity=4)
    (4)                 >>> G.add_edge("a", "c", weight=6, capacity=10)
    (4)                 >>> G.add_edge("b", "d", weight=1, capacity=9)
    (4)                 >>> G.add_edge("c", "d", weight=2, capacity=5)
    (4)                 >>> flowCost, flowDict = nx.network_simplex(G)
    (4)                 >>> flowCost
    (4)                 24
    (4)                 >>> flowDict
    (4)                 {'a': {'b': 4, 'c': 1}, 'd': {}, 'b': {'d': 4}, 'c': {'d': 1}}
```

```
(4)                 The mincost flow algorithm can also be used to solve shortest path
(4)                 problems. To find the shortest path between two nodes u and v,
(4)                 give all edges an infinite capacity, give node u a demand of -1 and
(4)                 node v a demand a 1. Then run the network simplex. The value of a
(4)                 min cost flow will be the distance between u and v and edges
(4)                 carrying positive flow will indicate the path.
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_weighted_edges_from(
(4)                 ...     [
(4)                 ...         ("s", "u", 10),
(4)                 ...         ("s", "x", 5),
(4)                 ...         ("u", "v", 1),
(4)                 ...         ("u", "x", 2),
(4)                 ...         ("v", "y", 1),
(4)                 ...         ("x", "u", 3),
(4)                 ...         ("x", "v", 5),
(4)                 ...         ("x", "y", 2),
(4)                 ...         ("y", "s", 7),
(4)                 ...         ("y", "v", 6),
(4)                 ...     ]
(4)                 ... )
(4)                 >>> G.add_node("s", demand=-1)
(4)                 >>> G.add_node("v", demand=1)
(4)                 >>> flowCost, flowDict = nx.network_simplex(G)
(4)                 >>> flowCost == nx.shortest_path_length(G, "s", "v", weight="weight")
(4)                 True
(4)                 >>> sorted([(u, v) for u in flowDict for v in flowDict[u] if flowDict[u]
[v] > 0])
(4)                 [('s', 'x'), ('u', 'v'), ('x', 'u')]
(4)                 >>> nx.shortest_path(G, "s", "v", weight="weight")
(4)                 ['s', 'x', 'u', 'v']
(4)                 It is possible to change the name of the attributes used for the
(4)                 algorithm.
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_node("p", spam=-4)
(4)                 >>> G.add_node("q", spam=2)
(4)                 >>> G.add_node("a", spam=-2)
(4)                 >>> G.add_node("d", spam=-1)
(4)                 >>> G.add_node("t", spam=2)
(4)                 >>> G.add_node("w", spam=3)
(4)                 >>> G.add_edge("p", "q", cost=7, vacancies=5)
(4)                 >>> G.add_edge("p", "a", cost=1, vacancies=4)
(4)                 >>> G.add_edge("q", "d", cost=2, vacancies=3)
(4)                 >>> G.add_edge("t", "q", cost=1, vacancies=2)
(4)                 >>> G.add_edge("a", "t", cost=2, vacancies=4)
(4)                 >>> G.add_edge("d", "w", cost=3, vacancies=4)
(4)                 >>> G.add_edge("t", "w", cost=4, vacancies=1)
(4)                 >>> flowCost, flowDict = nx.network_simplex(
(4)                 ...     G, demand="spam", capacity="vacancies", weight="cost"
(4)                 ... )
(4)                 >>> flowCost
(4)                 37
(4)                 >>> flowDict
(4)                 {'p': {'q': 2, 'a': 2}, 'q': {'d': 1}, 'a': {'t': 4}, 'd': {'w': 2}, 't':
{'q': 1, 'w': 1}, 'w': {}}
(4)                 References
(4)                 ----------
(4)                 .. [1] Z. Kiraly, P. Kovacs.
(11)                       Efficient implementation of minimum-cost flow algorithms.
(11)                       Acta Universitatis Sapientiae, Informatica 4(1):67--118. 2012.
(4)                 .. [2] R. Barr, F. Glover, D. Klingman.
(11)                       Enhancement of spanning tree labeling procedures for network
(11)                       optimization.
(11)                       INFOR 17(1):16--34. 1979.
(4)                 """
(4)                 if len(G) == 0:
(8)                     raise nx.NetworkXError("graph has no nodes")
(4)                 multigraph = G.is_multigraph()
(4)                 DEAF = _DataEssentialsAndFunctions(
```

```
424: (8)                    G, multigraph, demand=demand, capacity=capacity, weight=weight
425: (4)                )
426: (4)                inf = float("inf")
427: (4)                for u, d in zip(DEAF.node_list, DEAF.node_demands):
428: (8)                    if abs(d) == inf:
429: (12)                       raise nx.NetworkXError(f"node {u!r} has infinite demand")
430: (4)                for e, w in zip(DEAF.edge_indices, DEAF.edge_weights):
431: (8)                    if abs(w) == inf:
432: (12)                       raise nx.NetworkXError(f"edge {e!r} has infinite weight")
433: (4)                if not multigraph:
434: (8)                    edges = nx.selfloop_edges(G, data=True)
435: (4)                else:
436: (8)                    edges = nx.selfloop_edges(G, data=True, keys=True)
437: (4)                for e in edges:
438: (8)                    if abs(e[-1].get(weight, 0)) == inf:
439: (12)                       raise nx.NetworkXError(f"edge {e[:-1]!r} has infinite weight")
440: (4)                if sum(DEAF.node_demands) != 0:
441: (8)                    raise nx.NetworkXUnfeasible("total node demand is not zero")
442: (4)                for e, c in zip(DEAF.edge_indices, DEAF.edge_capacities):
443: (8)                    if c < 0:
444: (12)                       raise nx.NetworkXUnfeasible(f"edge {e!r} has negative capacity")
445: (4)                if not multigraph:
446: (8)                    edges = nx.selfloop_edges(G, data=True)
447: (4)                else:
448: (8)                    edges = nx.selfloop_edges(G, data=True, keys=True)
449: (4)                for e in edges:
450: (8)                    if e[-1].get(capacity, inf) < 0:
451: (12)                       raise nx.NetworkXUnfeasible(f"edge {e[:-1]!r} has negative
capacity")
452: (4)                for i, d in enumerate(DEAF.node_demands):
453: (8)                    if d > 0:
454: (12)                       DEAF.edge_sources.append(-1)
455: (12)                       DEAF.edge_targets.append(i)
456: (8)                    else:
457: (12)                       DEAF.edge_sources.append(i)
458: (12)                       DEAF.edge_targets.append(-1)
459: (4)                faux_inf = (
460: (8)                    3
461: (8)                    * max(
462: (12)                       chain(
463: (16)                           [
464: (20)                               sum(c for c in DEAF.edge_capacities if c < inf),
465: (20)                               sum(abs(w) for w in DEAF.edge_weights),
466: (16)                           ],
467: (16)                           (abs(d) for d in DEAF.node_demands),
468: (12)                       )
469: (8)                    )
470: (8)                    or 1
471: (4)                )
472: (4)                n = len(DEAF.node_list)  # number of nodes
473: (4)                DEAF.edge_weights.extend(repeat(faux_inf, n))
474: (4)                DEAF.edge_capacities.extend(repeat(faux_inf, n))
475: (4)                DEAF.initialize_spanning_tree(n, faux_inf)
476: (4)                for i, p, q in DEAF.find_entering_edges():
477: (8)                    Wn, We = DEAF.find_cycle(i, p, q)
478: (8)                    j, s, t = DEAF.find_leaving_edge(Wn, We)
479: (8)                    DEAF.augment_flow(Wn, We, DEAF.residual_capacity(j, s))
480: (8)                    if i != j:
481: (12)                       if DEAF.parent[t] != s:
482: (16)                           s, t = t, s
483: (12)                       if We.index(i) > We.index(j):
484: (16)                           p, q = q, p
485: (12)                       DEAF.remove_edge(s, t)
486: (12)                       DEAF.make_root(q)
487: (12)                       DEAF.add_edge(i, p, q)
488: (12)                       DEAF.update_potentials(i, p, q)
489: (4)                if any(DEAF.edge_flow[i] != 0 for i in range(-n, 0)):
490: (8)                    raise nx.NetworkXUnfeasible("no flow satisfies all node demands")
491: (4)                if any(DEAF.edge_flow[i] * 2 >= faux_inf for i in range(DEAF.edge_count))
```

```
or any(
492: (8)                    e[-1].get(capacity, inf) == inf and e[-1].get(weight, 0) < 0
493: (8)                    for e in nx.selfloop_edges(G, data=True)
494: (4)                ):
495: (8)                    raise nx.NetworkXUnbounded("negative cycle with infinite capacity
found")
496: (4)                del DEAF.edge_flow[DEAF.edge_count :]
497: (4)                flow_cost = sum(w * x for w, x in zip(DEAF.edge_weights, DEAF.edge_flow))
498: (4)                flow_dict = {n: {} for n in DEAF.node_list}
499: (4)                def add_entry(e):
500: (8)                    """Add a flow dict entry."""
501: (8)                    d = flow_dict[e[0]]
502: (8)                    for k in e[1:-2]:
503: (12)                       try:
504: (16)                           d = d[k]
505: (12)                       except KeyError:
506: (16)                           t = {}
507: (16)                           d[k] = t
508: (16)                           d = t
509: (8)                    d[e[-2]] = e[-1]
510: (4)                DEAF.edge_sources = (
511: (8)                    DEAF.node_list[s] for s in DEAF.edge_sources
512: (4)                )  # Use original nodes.
513: (4)                DEAF.edge_targets = (
514: (8)                    DEAF.node_list[t] for t in DEAF.edge_targets
515: (4)                )  # Use original nodes.
516: (4)                if not multigraph:
517: (8)                    for e in zip(DEAF.edge_sources, DEAF.edge_targets, DEAF.edge_flow):
518: (12)                       add_entry(e)
519: (8)                    edges = G.edges(data=True)
520: (4)                else:
521: (8)                    for e in zip(
522: (12)                       DEAF.edge_sources, DEAF.edge_targets, DEAF.edge_keys,
DEAF.edge_flow
523: (8)                    ):
524: (12)                       add_entry(e)
525: (8)                    edges = G.edges(data=True, keys=True)
526: (4)                for e in edges:
527: (8)                    if e[0] != e[1]:
528: (12)                       if e[-1].get(capacity, inf) == 0:
529: (16)                           add_entry(e[:-1] + (0,))
530: (8)                    else:
531: (12)                       w = e[-1].get(weight, 0)
532: (12)                       if w >= 0:
533: (16)                           add_entry(e[:-1] + (0,))
534: (12)                       else:
535: (16)                           c = e[-1][capacity]
536: (16)                           flow_cost += w * c
537: (16)                           add_entry(e[:-1] + (c,))
538: (4)                return flow_cost, flow_dict

----------------------------------------

File 245 - preflowpush.py:

1: (0)              """
2: (0)              Highest-label preflow-push algorithm for maximum flow problems.
3: (0)              """
4: (0)              from collections import deque
5: (0)              from itertools import islice
6: (0)              import networkx as nx
7: (0)              from ...utils import arbitrary_element
8: (0)              from .utils import (
9: (4)                  CurrentEdge,
10: (4)                 GlobalRelabelThreshold,
11: (4)                 Level,
12: (4)                 build_residual_network,
13: (4)                 detect_unboundedness,
14: (0)             )
```

```
(0)  __all__ = ["preflow_push"]
(0)  def preflow_push_impl(G, s, t, capacity, residual, global_relabel_freq, value_only):
(4)      """Implementation of the highest-label preflow-push algorithm."""
(4)      if s not in G:
(8)          raise nx.NetworkXError(f"node {str(s)} not in graph")
(4)      if t not in G:
(8)          raise nx.NetworkXError(f"node {str(t)} not in graph")
(4)      if s == t:
(8)          raise nx.NetworkXError("source and sink are the same node")
(4)      if global_relabel_freq is None:
(8)          global_relabel_freq = 0
(4)      if global_relabel_freq < 0:
(8)          raise nx.NetworkXError("global_relabel_freq must be nonnegative.")
(4)      if residual is None:
(8)          R = build_residual_network(G, capacity)
(4)      else:
(8)          R = residual
(4)      detect_unboundedness(R, s, t)
(4)      R_nodes = R.nodes
(4)      R_pred = R.pred
(4)      R_succ = R.succ
(4)      for u in R:
(8)          R_nodes[u]["excess"] = 0
(8)          for e in R_succ[u].values():
(12)             e["flow"] = 0
(4)      def reverse_bfs(src):
(8)          """Perform a reverse breadth-first search from src in the residual
(8)          network.
(8)          """
(8)          heights = {src: 0}
(8)          q = deque([(src, 0)])
(8)          while q:
(12)             u, height = q.popleft()
(12)             height += 1
(12)             for v, attr in R_pred[u].items():
(16)                 if v not in heights and attr["flow"] < attr["capacity"]:
(20)                     heights[v] = height
(20)                     q.append((v, height))
(8)          return heights
(4)      heights = reverse_bfs(t)
(4)      if s not in heights:
(8)          R.graph["flow_value"] = 0
(8)          return R
(4)      n = len(R)
(4)      max_height = max(heights[u] for u in heights if u != s)
(4)      heights[s] = n
(4)      grt = GlobalRelabelThreshold(n, R.size(), global_relabel_freq)
(4)      for u in R:
(8)          R_nodes[u]["height"] = heights[u] if u in heights else n + 1
(8)          R_nodes[u]["curr_edge"] = CurrentEdge(R_succ[u])
(4)      def push(u, v, flow):
(8)          """Push flow units of flow from u to v."""
(8)          R_succ[u][v]["flow"] += flow
(8)          R_succ[v][u]["flow"] -= flow
(8)          R_nodes[u]["excess"] -= flow
(8)          R_nodes[v]["excess"] += flow
(4)      for u, attr in R_succ[s].items():
(8)          flow = attr["capacity"]
(8)          if flow > 0:
(12)             push(s, u, flow)
(4)      levels = [Level() for i in range(2 * n)]
(4)      for u in R:
(8)          if u != s and u != t:
(12)             level = levels[R_nodes[u]["height"]]
(12)             if R_nodes[u]["excess"] > 0:
(16)                 level.active.add(u)
(12)             else:
(16)                 level.inactive.add(u)
```

```
83: (4)                 def activate(v):
84: (8)                     """Move a node from the inactive set to the active set of its
level."""
85: (8)                     if v != s and v != t:
86: (12)                        level = levels[R_nodes[v]["height"]]
87: (12)                        if v in level.inactive:
88: (16)                            level.inactive.remove(v)
89: (16)                            level.active.add(v)
90: (4)                 def relabel(u):
91: (8)                     """Relabel a node to create an admissible edge."""
92: (8)                     grt.add_work(len(R_succ[u]))
93: (8)                     return (
94: (12)                        min(
95: (16)                            R_nodes[v]["height"]
96: (16)                            for v, attr in R_succ[u].items()
97: (16)                            if attr["flow"] < attr["capacity"]
98: (12)                        )
99: (12)                        + 1
100: (8)                    )
101: (4)                def discharge(u, is_phase1):
102: (8)                    """Discharge a node until it becomes inactive or, during phase 1 (see
103: (8)                    below), its height reaches at least n. The node is known to have the
104: (8)                    largest height among active nodes.
105: (8)                    """
106: (8)                    height = R_nodes[u]["height"]
107: (8)                    curr_edge = R_nodes[u]["curr_edge"]
108: (8)                    next_height = height
109: (8)                    levels[height].active.remove(u)
110: (8)                    while True:
111: (12)                       v, attr = curr_edge.get()
112: (12)                       if height == R_nodes[v]["height"] + 1 and attr["flow"] <
attr["capacity"]:
113: (16)                           flow = min(R_nodes[u]["excess"], attr["capacity"] -
attr["flow"])
114: (16)                           push(u, v, flow)
115: (16)                           activate(v)
116: (16)                           if R_nodes[u]["excess"] == 0:
117: (20)                               levels[height].inactive.add(u)
118: (20)                               break
119: (12)                       try:
120: (16)                           curr_edge.move_to_next()
121: (12)                       except StopIteration:
122: (16)                           height = relabel(u)
123: (16)                           if is_phase1 and height >= n - 1:
124: (20)                               levels[height].active.add(u)
125: (20)                               break
126: (16)                           next_height = height
127: (8)                    R_nodes[u]["height"] = height
128: (8)                    return next_height
129: (4)                def gap_heuristic(height):
130: (8)                    """Apply the gap heuristic."""
131: (8)                    for level in islice(levels, height + 1, max_height + 1):
132: (12)                       for u in level.active:
133: (16)                           R_nodes[u]["height"] = n + 1
134: (12)                       for u in level.inactive:
135: (16)                           R_nodes[u]["height"] = n + 1
136: (12)                       levels[n + 1].active.update(level.active)
137: (12)                       level.active.clear()
138: (12)                       levels[n + 1].inactive.update(level.inactive)
139: (12)                       level.inactive.clear()
140: (4)                def global_relabel(from_sink):
141: (8)                    """Apply the global relabeling heuristic."""
142: (8)                    src = t if from_sink else s
143: (8)                    heights = reverse_bfs(src)
144: (8)                    if not from_sink:
145: (12)                       del heights[t]
146: (8)                    max_height = max(heights.values())
147: (8)                    if from_sink:
148: (12)                       for u in R:
```

```
149: (16)                            if u not in heights and R_nodes[u]["height"] < n:
150: (20)                                heights[u] = n + 1
151: (8)                    else:
152: (12)                       for u in heights:
153: (16)                           heights[u] += n
154: (12)                       max_height += n
155: (8)                    del heights[src]
156: (8)                    for u, new_height in heights.items():
157: (12)                       old_height = R_nodes[u]["height"]
158: (12)                       if new_height != old_height:
159: (16)                           if u in levels[old_height].active:
160: (20)                               levels[old_height].active.remove(u)
161: (20)                               levels[new_height].active.add(u)
162: (16)                           else:
163: (20)                               levels[old_height].inactive.remove(u)
164: (20)                               levels[new_height].inactive.add(u)
165: (16)                           R_nodes[u]["height"] = new_height
166: (8)                    return max_height
167: (4)                height = max_height
168: (4)                while height > 0:
169: (8)                    while True:
170: (12)                       level = levels[height]
171: (12)                       if not level.active:
172: (16)                           height -= 1
173: (16)                           break
174: (12)                       old_height = height
175: (12)                       old_level = level
176: (12)                       u = arbitrary_element(level.active)
177: (12)                       height = discharge(u, True)
178: (12)                       if grt.is_reached():
179: (16)                           height = global_relabel(True)
180: (16)                           max_height = height
181: (16)                           grt.clear_work()
182: (12)                       elif not old_level.active and not old_level.inactive:
183: (16)                           gap_heuristic(old_height)
184: (16)                           height = old_height - 1
185: (16)                           max_height = height
186: (12)                       else:
187: (16)                           max_height = max(max_height, height)
188: (4)                if value_only:
189: (8)                    R.graph["flow_value"] = R_nodes[t]["excess"]
190: (8)                    return R
191: (4)                height = global_relabel(False)
192: (4)                grt.clear_work()
193: (4)                while height > n:
194: (8)                    while True:
195: (12)                       level = levels[height]
196: (12)                       if not level.active:
197: (16)                           height -= 1
198: (16)                           break
199: (12)                       u = arbitrary_element(level.active)
200: (12)                       height = discharge(u, False)
201: (12)                       if grt.is_reached():
202: (16)                           height = global_relabel(False)
203: (16)                           grt.clear_work()
204: (4)                R.graph["flow_value"] = R_nodes[t]["excess"]
205: (4)                return R
206: (0)            @nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
207: (0)            def preflow_push(
208: (4)                G, s, t, capacity="capacity", residual=None, global_relabel_freq=1,
value_only=False
209: (0)            ):
210: (4)                r"""Find a maximum single-commodity flow using the highest-label
211: (4)                preflow-push algorithm.
212: (4)                This function returns the residual network resulting after computing
213: (4)                the maximum flow. See below for details about the conventions
214: (4)                NetworkX uses for defining residual networks.
215: (4)                This algorithm has a running time of $O(n^2 \sqrt{m})$ for $n$ nodes and
216: (4)                $m$ edges.
```

```
    Parameters
    ----------
    G : NetworkX graph
        Edges of the graph are expected to have an attribute called
        'capacity'. If this attribute is not present, the edge is
        considered to have infinite capacity.
    s : node
        Source node for the flow.
    t : node
        Sink node for the flow.
    capacity : string
        Edges of the graph G are expected to have an attribute capacity
        that indicates how much flow the edge can support. If this
        attribute is not present, the edge is considered to have
        infinite capacity. Default value: 'capacity'.
    residual : NetworkX graph
        Residual network on which the algorithm is to be executed. If None, a
        new residual network is created. Default value: None.
    global_relabel_freq : integer, float
        Relative frequency of applying the global relabeling heuristic to
speed

        up the algorithm. If it is None, the heuristic is disabled. Default
        value: 1.
    value_only : bool
        If False, compute a maximum flow; otherwise, compute a maximum preflow
        which is enough for computing the maximum flow value. Default value:
        False.
    Returns
    -------
    R : NetworkX DiGraph
        Residual network after computing the maximum flow.
    Raises
    ------
    NetworkXError
        The algorithm does not support MultiGraph and MultiDiGraph. If
        the input graph is an instance of one of these two classes, a
        NetworkXError is raised.
    NetworkXUnbounded
        If the graph has a path of infinite capacity, the value of a
        feasible flow on the graph is unbounded above and the function
        raises a NetworkXUnbounded.
    See also
    --------
    :meth:`maximum_flow`
    :meth:`minimum_cut`
    :meth:`edmonds_karp`
    :meth:`shortest_augmenting_path`
    Notes
    -----
    The residual network :samp:`R` from an input graph :samp:`G` has the
    same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
    of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
    self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
    in :samp:`G`. For each node :samp:`u` in :samp:`R`,
    :samp:`R.nodes[u]['excess']` represents the difference between flow into
    :samp:`u` and flow out of :samp:`u`.
    For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
    is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
    in :samp:`G` or zero otherwise. If the capacity is infinite,
    :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
    that does not affect the solution of the problem. This value is stored in
    :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
    :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
    satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
    The flow value, defined as the total flow into :samp:`t`, the sink, is
    stored in :samp:`R.graph['flow_value']`. Reachability to :samp:`t` using
    only edges :samp:`(u, v)` such that
    :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
    :samp:`s`-:samp:`t` cut.
```

```
285: (4)                Examples
286: (4)                --------
287: (4)                >>> from networkx.algorithms.flow import preflow_push
288: (4)                The functions that implement flow algorithms and output a residual
289: (4)                network, such as this one, are not imported to the base NetworkX
290: (4)                namespace, so you have to explicitly import them from the flow package.
291: (4)                >>> G = nx.DiGraph()
292: (4)                >>> G.add_edge("x", "a", capacity=3.0)
293: (4)                >>> G.add_edge("x", "b", capacity=1.0)
294: (4)                >>> G.add_edge("a", "c", capacity=3.0)
295: (4)                >>> G.add_edge("b", "c", capacity=5.0)
296: (4)                >>> G.add_edge("b", "d", capacity=4.0)
297: (4)                >>> G.add_edge("d", "e", capacity=2.0)
298: (4)                >>> G.add_edge("c", "y", capacity=2.0)
299: (4)                >>> G.add_edge("e", "y", capacity=3.0)
300: (4)                >>> R = preflow_push(G, "x", "y")
301: (4)                >>> flow_value = nx.maximum_flow_value(G, "x", "y")
302: (4)                >>> flow_value == R.graph["flow_value"]
303: (4)                True
304: (4)                >>> # preflow_push also stores the maximum flow value
305: (4)                >>> # in the excess attribute of the sink node t
306: (4)                >>> flow_value == R.nodes["y"]["excess"]
307: (4)                True
308: (4)                >>> # For some problems, you might only want to compute a
309: (4)                >>> # maximum preflow.
310: (4)                >>> R = preflow_push(G, "x", "y", value_only=True)
311: (4)                >>> flow_value == R.graph["flow_value"]
312: (4)                True
313: (4)                >>> flow_value == R.nodes["y"]["excess"]
314: (4)                True
315: (4)                """
316: (4)                R = preflow_push_impl(G, s, t, capacity, residual, global_relabel_freq,
value_only)
317: (4)                R.graph["algorithm"] = "preflow_push"
318: (4)                nx._clear_cache(R)
319: (4)                return R

----------------------------------------

File 246 - shortestaugmentingpath.py:

1: (0)              """
2: (0)              Shortest augmenting path algorithm for maximum flow problems.
3: (0)              """
4: (0)              from collections import deque
5: (0)              import networkx as nx
6: (0)              from .edmondskarp import edmonds_karp_core
7: (0)              from .utils import CurrentEdge, build_residual_network
8: (0)              __all__ = ["shortest_augmenting_path"]
9: (0)              def shortest_augmenting_path_impl(G, s, t, capacity, residual, two_phase,
cutoff):
10: (4)                 """Implementation of the shortest augmenting path algorithm."""
11: (4)                 if s not in G:
12: (8)                     raise nx.NetworkXError(f"node {str(s)} not in graph")
13: (4)                 if t not in G:
14: (8)                     raise nx.NetworkXError(f"node {str(t)} not in graph")
15: (4)                 if s == t:
16: (8)                     raise nx.NetworkXError("source and sink are the same node")
17: (4)                 if residual is None:
18: (8)                     R = build_residual_network(G, capacity)
19: (4)                 else:
20: (8)                     R = residual
21: (4)                 R_nodes = R.nodes
22: (4)                 R_pred = R.pred
23: (4)                 R_succ = R.succ
24: (4)                 for u in R:
25: (8)                     for e in R_succ[u].values():
26: (12)                        e["flow"] = 0
27: (4)                 heights = {t: 0}
```

```
(4)                 q = deque([(t, 0)])
(4)                 while q:
(8)                     u, height = q.popleft()
(8)                     height += 1
(8)                     for v, attr in R_pred[u].items():
(12)                        if v not in heights and attr["flow"] < attr["capacity"]:
(16)                            heights[v] = height
(16)                            q.append((v, height))
(4)                 if s not in heights:
(8)                     R.graph["flow_value"] = 0
(8)                     return R
(4)                 n = len(G)
(4)                 m = R.size() / 2
(4)                 for u in R:
(8)                     R_nodes[u]["height"] = heights[u] if u in heights else n
(8)                     R_nodes[u]["curr_edge"] = CurrentEdge(R_succ[u])
(4)                 counts = [0] * (2 * n - 1)
(4)                 for u in R:
(8)                     counts[R_nodes[u]["height"]] += 1
(4)                 inf = R.graph["inf"]
(4)                 def augment(path):
(8)                     """Augment flow along a path from s to t."""
(8)                     flow = inf
(8)                     it = iter(path)
(8)                     u = next(it)
(8)                     for v in it:
(12)                        attr = R_succ[u][v]
(12)                        flow = min(flow, attr["capacity"] - attr["flow"])
(12)                        u = v
(8)                     if flow * 2 > inf:
(12)                        raise nx.NetworkXUnbounded("Infinite capacity path, flow unbounded
above.")
(8)                     it = iter(path)
(8)                     u = next(it)
(8)                     for v in it:
(12)                        R_succ[u][v]["flow"] += flow
(12)                        R_succ[v][u]["flow"] -= flow
(12)                        u = v
(8)                     return flow
(4)                 def relabel(u):
(8)                     """Relabel a node to create an admissible edge."""
(8)                     height = n - 1
(8)                     for v, attr in R_succ[u].items():
(12)                        if attr["flow"] < attr["capacity"]:
(16)                            height = min(height, R_nodes[v]["height"])
(8)                     return height + 1
(4)                 if cutoff is None:
(8)                     cutoff = float("inf")
(4)                 flow_value = 0
(4)                 path = [s]
(4)                 u = s
(4)                 d = n if not two_phase else int(min(m**0.5, 2 * n ** (2.0 / 3)))
(4)                 done = R_nodes[s]["height"] >= d
(4)                 while not done:
(8)                     height = R_nodes[u]["height"]
(8)                     curr_edge = R_nodes[u]["curr_edge"]
(8)                     while True:
(12)                        v, attr = curr_edge.get()
(12)                        if height == R_nodes[v]["height"] + 1 and attr["flow"] <
attr["capacity"]:
(16)                            path.append(v)
(16)                            u = v
(16)                            break
(12)                        try:
(16)                            curr_edge.move_to_next()
(12)                        except StopIteration:
(16)                            counts[height] -= 1
(16)                            if counts[height] == 0:
(20)                                R.graph["flow_value"] = flow_value
```

```
(20)                                return R
(16)                            height = relabel(u)
(16)                            if u == s and height >= d:
(20)                                if not two_phase:
(24)                                    R.graph["flow_value"] = flow_value
(24)                                    return R
(20)                                else:
(24)                                    done = True
(24)                                    break
(16)                            counts[height] += 1
(16)                            R_nodes[u]["height"] = height
(16)                            if u != s:
(20)                                path.pop()
(20)                                u = path[-1]
(20)                                break
(8)                     if u == t:
(12)                        flow_value += augment(path)
(12)                        if flow_value >= cutoff:
(16)                            R.graph["flow_value"] = flow_value
(16)                            return R
(12)                        path = [s]
(12)                        u = s
(4)                 flow_value += edmonds_karp_core(R, s, t, cutoff - flow_value)
(4)                 R.graph["flow_value"] = flow_value
(4)                 return R
(0)             @nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
(0)             def shortest_augmenting_path(
(4)                 G,
(4)                 s,
(4)                 t,
(4)                 capacity="capacity",
(4)                 residual=None,
(4)                 value_only=False,
(4)                 two_phase=False,
(4)                 cutoff=None,
(0)             ):
(4)                 r"""Find a maximum single-commodity flow using the shortest augmenting
path
(4)                 algorithm.
(4)                 This function returns the residual network resulting after computing
(4)                 the maximum flow. See below for details about the conventions
(4)                 NetworkX uses for defining residual networks.
(4)                 This algorithm has a running time of $O(n^2 m)$ for $n$ nodes and $m$
(4)                 edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     Edges of the graph are expected to have an attribute called
(8)                     'capacity'. If this attribute is not present, the edge is
(8)                     considered to have infinite capacity.
(4)                 s : node
(8)                     Source node for the flow.
(4)                 t : node
(8)                     Sink node for the flow.
(4)                 capacity : string
(8)                     Edges of the graph G are expected to have an attribute capacity
(8)                     that indicates how much flow the edge can support. If this
(8)                     attribute is not present, the edge is considered to have
(8)                     infinite capacity. Default value: 'capacity'.
(4)                 residual : NetworkX graph
(8)                     Residual network on which the algorithm is to be executed. If None, a
(8)                     new residual network is created. Default value: None.
(4)                 value_only : bool
(8)                     If True compute only the value of the maximum flow. This parameter
(8)                     will be ignored by this algorithm because it is not applicable.
(4)                 two_phase : bool
(8)                     If True, a two-phase variant is used. The two-phase variant improves
(8)                     the running time on unit-capacity networks from $O(nm)$ to
(8)                     $O(\min(n^{2/3}, m^{1/2}) m)$. Default value: False.
```

```
    cutoff : integer, float
        If specified, the algorithm will terminate when the flow value reaches
        or exceeds the cutoff. In this case, it may be unable to immediately
        determine a minimum cut. Default value: None.
    Returns
    -------
    R : NetworkX DiGraph
        Residual network after computing the maximum flow.
    Raises
    ------
    NetworkXError
        The algorithm does not support MultiGraph and MultiDiGraph. If
        the input graph is an instance of one of these two classes, a
        NetworkXError is raised.
    NetworkXUnbounded
        If the graph has a path of infinite capacity, the value of a
        feasible flow on the graph is unbounded above and the function
        raises a NetworkXUnbounded.
    See also
    --------
    :meth:`maximum_flow`
    :meth:`minimum_cut`
    :meth:`edmonds_karp`
    :meth:`preflow_push`
    Notes
    -----
    The residual network :samp:`R` from an input graph :samp:`G` has the
    same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
    of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
    self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
    in :samp:`G`.
    For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
    is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
    in :samp:`G` or zero otherwise. If the capacity is infinite,
    :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
    that does not affect the solution of the problem. This value is stored in
    :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
    :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
    satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
    The flow value, defined as the total flow into :samp:`t`, the sink, is
    stored in :samp:`R.graph['flow_value']`. If :samp:`cutoff` is not
    specified, reachability to :samp:`t` using only edges :samp:`(u, v)` such
    that :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
    :samp:`s`-:samp:`t` cut.
    Examples
    --------
    >>> from networkx.algorithms.flow import shortest_augmenting_path
    The functions that implement flow algorithms and output a residual
    network, such as this one, are not imported to the base NetworkX
    namespace, so you have to explicitly import them from the flow package.
    >>> G = nx.DiGraph()
    >>> G.add_edge("x", "a", capacity=3.0)
    >>> G.add_edge("x", "b", capacity=1.0)
    >>> G.add_edge("a", "c", capacity=3.0)
    >>> G.add_edge("b", "c", capacity=5.0)
    >>> G.add_edge("b", "d", capacity=4.0)
    >>> G.add_edge("d", "e", capacity=2.0)
    >>> G.add_edge("c", "y", capacity=2.0)
    >>> G.add_edge("e", "y", capacity=3.0)
    >>> R = shortest_augmenting_path(G, "x", "y")
    >>> flow_value = nx.maximum_flow_value(G, "x", "y")
    >>> flow_value
    3.0
    >>> flow_value == R.graph["flow_value"]
    True
    """
    R = shortest_augmenting_path_impl(G, s, t, capacity, residual, two_phase, cutoff)
    R.graph["algorithm"] = "shortest_augmenting_path"
```

```
231: (4)                nx._clear_cache(R)
232: (4)                return R
```

File 247 - utils.py:

```
1: (0)              """
2: (0)              Utility classes and functions for network flow algorithms.
3: (0)              """
4: (0)              from collections import deque
5: (0)              import networkx as nx
6: (0)              __all__ = [
7: (4)                  "CurrentEdge",
8: (4)                  "Level",
9: (4)                  "GlobalRelabelThreshold",
10: (4)                 "build_residual_network",
11: (4)                 "detect_unboundedness",
12: (4)                 "build_flow_dict",
13: (0)             ]
14: (0)             class CurrentEdge:
15: (4)                 """Mechanism for iterating over out-edges incident to a node in a circular
16: (4)                 manner. StopIteration exception is raised when wraparound occurs.
17: (4)                 """
18: (4)                 __slots__ = ("_edges", "_it", "_curr")
19: (4)                 def __init__(self, edges):
20: (8)                     self._edges = edges
21: (8)                     if self._edges:
22: (12)                        self._rewind()
23: (4)                 def get(self):
24: (8)                     return self._curr
25: (4)                 def move_to_next(self):
26: (8)                     try:
27: (12)                        self._curr = next(self._it)
28: (8)                     except StopIteration:
29: (12)                        self._rewind()
30: (12)                        raise
31: (4)                 def _rewind(self):
32: (8)                     self._it = iter(self._edges.items())
33: (8)                     self._curr = next(self._it)
34: (0)             class Level:
35: (4)                 """Active and inactive nodes in a level."""
36: (4)                 __slots__ = ("active", "inactive")
37: (4)                 def __init__(self):
38: (8)                     self.active = set()
39: (8)                     self.inactive = set()
40: (0)             class GlobalRelabelThreshold:
41: (4)                 """Measurement of work before the global relabeling heuristic should be
42: (4)                 applied.
43: (4)                 """
44: (4)                 def __init__(self, n, m, freq):
45: (8)                     self._threshold = (n + m) / freq if freq else float("inf")
46: (8)                     self._work = 0
47: (4)                 def add_work(self, work):
48: (8)                     self._work += work
49: (4)                 def is_reached(self):
50: (8)                     return self._work >= self._threshold
51: (4)                 def clear_work(self):
52: (8)                     self._work = 0
53: (0)             @nx._dispatchable(edge_attrs={"capacity": float("inf")}, returns_graph=True)
54: (0)             def build_residual_network(G, capacity):
55: (4)                 """Build a residual network and initialize a zero flow.
56: (4)                 The residual network :samp:`R` from an input graph :samp:`G` has the
57: (4)                 same nodes as :samp:`G`. :samp:`R` is a DiGraph that contains a pair
58: (4)                 of edges :samp:`(u, v)` and :samp:`(v, u)` iff :samp:`(u, v)` is not a
59: (4)                 self-loop, and at least one of :samp:`(u, v)` and :samp:`(v, u)` exists
60: (4)                 in :samp:`G`.
61: (4)                 For each edge :samp:`(u, v)` in :samp:`R`, :samp:`R[u][v]['capacity']`
62: (4)                 is equal to the capacity of :samp:`(u, v)` in :samp:`G` if it exists
```

```
63: (4)                 in :samp:`G` or zero otherwise. If the capacity is infinite,
64: (4)                 :samp:`R[u][v]['capacity']` will have a high arbitrary finite value
65: (4)                 that does not affect the solution of the problem. This value is stored in
66: (4)                 :samp:`R.graph['inf']`. For each edge :samp:`(u, v)` in :samp:`R`,
67: (4)                 :samp:`R[u][v]['flow']` represents the flow function of :samp:`(u, v)` and
68: (4)                 satisfies :samp:`R[u][v]['flow'] == -R[v][u]['flow']`.
69: (4)                 The flow value, defined as the total flow into :samp:`t`, the sink, is
70: (4)                 stored in :samp:`R.graph['flow_value']`. If :samp:`cutoff` is not
71: (4)                 specified, reachability to :samp:`t` using only edges :samp:`(u, v)` such
72: (4)                 that :samp:`R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum
73: (4)                 :samp:`s`-:samp:`t` cut.
74: (4)                 """
75: (4)                 if G.is_multigraph():
76: (8)                     raise nx.NetworkXError("MultiGraph and MultiDiGraph not supported
(yet).")
77: (4)                 R = nx.DiGraph()
78: (4)                 R.__networkx_cache__ = None  # Disable caching
79: (4)                 R.add_nodes_from(G)
80: (4)                 inf = float("inf")
81: (4)                 edge_list = [
82: (8)                     (u, v, attr)
83: (8)                     for u, v, attr in G.edges(data=True)
84: (8)                     if u != v and attr.get(capacity, inf) > 0
85: (4)                 ]
86: (4)                 inf = (
87: (8)                     3
88: (8)                     * sum(
89: (12)                        attr[capacity]
90: (12)                        for u, v, attr in edge_list
91: (12)                        if capacity in attr and attr[capacity] != inf
92: (8)                     )
93: (8)                     or 1
94: (4)                 )
95: (4)                 if G.is_directed():
96: (8)                     for u, v, attr in edge_list:
97: (12)                        r = min(attr.get(capacity, inf), inf)
98: (12)                        if not R.has_edge(u, v):
99: (16)                            R.add_edge(u, v, capacity=r)
100: (16)                           R.add_edge(v, u, capacity=0)
101: (12)                       else:
102: (16)                           R[u][v]["capacity"] = r
103: (4)                else:
104: (8)                    for u, v, attr in edge_list:
105: (12)                       r = min(attr.get(capacity, inf), inf)
106: (12)                       R.add_edge(u, v, capacity=r)
107: (12)                       R.add_edge(v, u, capacity=r)
108: (4)                R.graph["inf"] = inf
109: (4)                return R
110: (0)            @nx._dispatchable(
111: (4)                graphs="R",
112: (4)                preserve_edge_attrs={"R": {"capacity": float("inf")}},
113: (4)                preserve_graph_attrs=True,
114: (0)            )
115: (0)            def detect_unboundedness(R, s, t):
116: (4)                """Detect an infinite-capacity s-t path in R."""
117: (4)                q = deque([s])
118: (4)                seen = {s}
119: (4)                inf = R.graph["inf"]
120: (4)                while q:
121: (8)                    u = q.popleft()
122: (8)                    for v, attr in R[u].items():
123: (12)                       if attr["capacity"] == inf and v not in seen:
124: (16)                           if v == t:
125: (20)                               raise nx.NetworkXUnbounded(
126: (24)                                   "Infinite capacity path, flow unbounded above."
127: (20)                               )
128: (16)                           seen.add(v)
129: (16)                           q.append(v)
130: (0)            @nx._dispatchable(graphs={"G": 0, "R": 1}, preserve_edge_attrs={"R": {"flow":
```

```
None}})
131: (0)            def build_flow_dict(G, R):
132: (4)                """Build a flow dictionary from a residual network."""
133: (4)                flow_dict = {}
134: (4)                for u in G:
135: (8)                    flow_dict[u] = {v: 0 for v in G[u]}
136: (8)                    flow_dict[u].update(
137: (12)                       (v, attr["flow"]) for v, attr in R[u].items() if attr["flow"] > 0
138: (8)                    )
139: (4)                return flow_dict
```

File 248 - __init__.py:

```
1: (0)
```

File 249 - test_maxflow.py:

```
1: (0)              """Maximum flow algorithms test suite.
2: (0)              """
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.flow import (
6: (4)                  boykov_kolmogorov,
7: (4)                  build_flow_dict,
8: (4)                  build_residual_network,
9: (4)                  dinitz,
10: (4)                 edmonds_karp,
11: (4)                 preflow_push,
12: (4)                 shortest_augmenting_path,
13: (0)             )
14: (0)             flow_funcs = {
15: (4)                 boykov_kolmogorov,
16: (4)                 dinitz,
17: (4)                 edmonds_karp,
18: (4)                 preflow_push,
19: (4)                 shortest_augmenting_path,
20: (0)             }
21: (0)             max_min_funcs = {nx.maximum_flow, nx.minimum_cut}
22: (0)             flow_value_funcs = {nx.maximum_flow_value, nx.minimum_cut_value}
23: (0)             interface_funcs = max_min_funcs & flow_value_funcs
24: (0)             all_funcs = flow_funcs & interface_funcs
25: (0)             def compute_cutset(G, partition):
26: (4)                 reachable, non_reachable = partition
27: (4)                 cutset = set()
28: (4)                 for u, nbrs in ((n, G[n]) for n in reachable):
29: (8)                     cutset.update((u, v) for v in nbrs if v in non_reachable)
30: (4)                 return cutset
31: (0)             def validate_flows(G, s, t, flowDict, solnValue, capacity, flow_func):
32: (4)                 errmsg = f"Assertion failed in function: {flow_func.__name__}"
33: (4)                 assert set(G) == set(flowDict), errmsg
34: (4)                 for u in G:
35: (8)                     assert set(G[u]) == set(flowDict[u]), errmsg
36: (4)                 excess = {u: 0 for u in flowDict}
37: (4)                 for u in flowDict:
38: (8)                     for v, flow in flowDict[u].items():
39: (12)                        if capacity in G[u][v]:
40: (16)                            assert flow <= G[u][v][capacity]
41: (12)                        assert flow >= 0, errmsg
42: (12)                        excess[u] -= flow
43: (12)                        excess[v] += flow
44: (4)                 for u, exc in excess.items():
45: (8)                     if u == s:
46: (12)                        assert exc == -solnValue, errmsg
47: (8)                     elif u == t:
48: (12)                        assert exc == solnValue, errmsg
```

```
49: (8)                     else:
50: (12)                        assert exc == 0, errmsg
51: (0)             def validate_cuts(G, s, t, solnValue, partition, capacity, flow_func):
52: (4)                 errmsg = f"Assertion failed in function: {flow_func.__name__}"
53: (4)                 assert all(n in G for n in partition[0]), errmsg
54: (4)                 assert all(n in G for n in partition[1]), errmsg
55: (4)                 cutset = compute_cutset(G, partition)
56: (4)                 assert all(G.has_edge(u, v) for (u, v) in cutset), errmsg
57: (4)                 assert solnValue == sum(G[u][v][capacity] for (u, v) in cutset), errmsg
58: (4)                 H = G.copy()
59: (4)                 H.remove_edges_from(cutset)
60: (4)                 if not G.is_directed():
61: (8)                     assert not nx.is_connected(H), errmsg
62: (4)                 else:
63: (8)                     assert not nx.is_strongly_connected(H), errmsg
64: (0)             def compare_flows_and_cuts(G, s, t, solnFlows, solnValue,
capacity="capacity"):
65: (4)                 for flow_func in flow_funcs:
66: (8)                     errmsg = f"Assertion failed in function: {flow_func.__name__}"
67: (8)                     R = flow_func(G, s, t, capacity)
68: (8)                     flow_value = R.graph["flow_value"]
69: (8)                     flow_dict = build_flow_dict(G, R)
70: (8)                     assert flow_value == solnValue, errmsg
71: (8)                     validate_flows(G, s, t, flow_dict, solnValue, capacity, flow_func)
72: (8)                     cut_value, partition = nx.minimum_cut(
73: (12)                        G, s, t, capacity=capacity, flow_func=flow_func
74: (8)                     )
75: (8)                     validate_cuts(G, s, t, solnValue, partition, capacity, flow_func)
76: (0)             class TestMaxflowMinCutCommon:
77: (4)                 def test_graph1(self):
78: (8)                     G = nx.Graph()
79: (8)                     G.add_edge(1, 2, capacity=1.0)
80: (8)                     solnFlows = {1: {2: 1.0}, 2: {1: 1.0}}
81: (8)                     compare_flows_and_cuts(G, 1, 2, solnFlows, 1.0)
82: (4)                 def test_graph2(self):
83: (8)                     G = nx.Graph()
84: (8)                     G.add_edge("x", "a", capacity=3.0)
85: (8)                     G.add_edge("x", "b", capacity=1.0)
86: (8)                     G.add_edge("a", "c", capacity=3.0)
87: (8)                     G.add_edge("b", "c", capacity=5.0)
88: (8)                     G.add_edge("b", "d", capacity=4.0)
89: (8)                     G.add_edge("d", "e", capacity=2.0)
90: (8)                     G.add_edge("c", "y", capacity=2.0)
91: (8)                     G.add_edge("e", "y", capacity=3.0)
92: (8)                     H = {
93: (12)                        "x": {"a": 3, "b": 1},
94: (12)                        "a": {"c": 3, "x": 3},
95: (12)                        "b": {"c": 1, "d": 2, "x": 1},
96: (12)                        "c": {"a": 3, "b": 1, "y": 2},
97: (12)                        "d": {"b": 2, "e": 2},
98: (12)                        "e": {"d": 2, "y": 2},
99: (12)                        "y": {"c": 2, "e": 2},
100: (8)                    }
101: (8)                    compare_flows_and_cuts(G, "x", "y", H, 4.0)
102: (4)                def test_digraph1(self):
103: (8)                    G = nx.DiGraph()
104: (8)                    G.add_edge("a", "b", capacity=1000.0)
105: (8)                    G.add_edge("a", "c", capacity=1000.0)
106: (8)                    G.add_edge("b", "c", capacity=1.0)
107: (8)                    G.add_edge("b", "d", capacity=1000.0)
108: (8)                    G.add_edge("c", "d", capacity=1000.0)
109: (8)                    H = {
110: (12)                       "a": {"b": 1000.0, "c": 1000.0},
111: (12)                       "b": {"c": 0, "d": 1000.0},
112: (12)                       "c": {"d": 1000.0},
113: (12)                       "d": {},
114: (8)                    }
115: (8)                    compare_flows_and_cuts(G, "a", "d", H, 2000.0)
116: (4)                def test_digraph2(self):
```

```
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", "b", capacity=2)
(8)                     G.add_edge("s", "c", capacity=1)
(8)                     G.add_edge("c", "d", capacity=1)
(8)                     G.add_edge("d", "a", capacity=1)
(8)                     G.add_edge("b", "a", capacity=2)
(8)                     G.add_edge("a", "t", capacity=2)
(8)                     H = {
(12)                        "s": {"b": 2, "c": 0},
(12)                        "c": {"d": 0},
(12)                        "d": {"a": 0},
(12)                        "b": {"a": 2},
(12)                        "a": {"t": 2},
(12)                        "t": {},
(8)                     }
(8)                     compare_flows_and_cuts(G, "s", "t", H, 2)
(4)                 def test_digraph3(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", "v1", capacity=16.0)
(8)                     G.add_edge("s", "v2", capacity=13.0)
(8)                     G.add_edge("v1", "v2", capacity=10.0)
(8)                     G.add_edge("v2", "v1", capacity=4.0)
(8)                     G.add_edge("v1", "v3", capacity=12.0)
(8)                     G.add_edge("v3", "v2", capacity=9.0)
(8)                     G.add_edge("v2", "v4", capacity=14.0)
(8)                     G.add_edge("v4", "v3", capacity=7.0)
(8)                     G.add_edge("v3", "t", capacity=20.0)
(8)                     G.add_edge("v4", "t", capacity=4.0)
(8)                     H = {
(12)                        "s": {"v1": 12.0, "v2": 11.0},
(12)                        "v2": {"v1": 0, "v4": 11.0},
(12)                        "v1": {"v2": 0, "v3": 12.0},
(12)                        "v3": {"v2": 0, "t": 19.0},
(12)                        "v4": {"v3": 7.0, "t": 4.0},
(12)                        "t": {},
(8)                     }
(8)                     compare_flows_and_cuts(G, "s", "t", H, 23.0)
(4)                 def test_digraph4(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("x", "a", capacity=3.0)
(8)                     G.add_edge("x", "b", capacity=1.0)
(8)                     G.add_edge("a", "c", capacity=3.0)
(8)                     G.add_edge("b", "c", capacity=5.0)
(8)                     G.add_edge("b", "d", capacity=4.0)
(8)                     G.add_edge("d", "e", capacity=2.0)
(8)                     G.add_edge("c", "y", capacity=2.0)
(8)                     G.add_edge("e", "y", capacity=3.0)
(8)                     H = {
(12)                        "x": {"a": 2.0, "b": 1.0},
(12)                        "a": {"c": 2.0},
(12)                        "b": {"c": 0, "d": 1.0},
(12)                        "c": {"y": 2.0},
(12)                        "d": {"e": 1.0},
(12)                        "e": {"y": 1.0},
(12)                        "y": {},
(8)                     }
(8)                     compare_flows_and_cuts(G, "x", "y", H, 3.0)
(4)                 def test_wikipedia_dinitz_example(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", 1, capacity=10)
(8)                     G.add_edge("s", 2, capacity=10)
(8)                     G.add_edge(1, 3, capacity=4)
(8)                     G.add_edge(1, 4, capacity=8)
(8)                     G.add_edge(1, 2, capacity=2)
(8)                     G.add_edge(2, 4, capacity=9)
(8)                     G.add_edge(3, "t", capacity=10)
(8)                     G.add_edge(4, 3, capacity=6)
(8)                     G.add_edge(4, "t", capacity=10)
(8)                     solnFlows = {
```

```
(12)                        1: {2: 0, 3: 4, 4: 6},
(12)                        2: {4: 9},
(12)                        3: {"t": 9},
(12)                        4: {3: 5, "t": 10},
(12)                        "s": {1: 10, 2: 9},
(12)                        "t": {},
(8)                     }
(8)                     compare_flows_and_cuts(G, "s", "t", solnFlows, 19)
(4)                 def test_optional_capacity(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("x", "a", spam=3.0)
(8)                     G.add_edge("x", "b", spam=1.0)
(8)                     G.add_edge("a", "c", spam=3.0)
(8)                     G.add_edge("b", "c", spam=5.0)
(8)                     G.add_edge("b", "d", spam=4.0)
(8)                     G.add_edge("d", "e", spam=2.0)
(8)                     G.add_edge("c", "y", spam=2.0)
(8)                     G.add_edge("e", "y", spam=3.0)
(8)                     solnFlows = {
(12)                        "x": {"a": 2.0, "b": 1.0},
(12)                        "a": {"c": 2.0},
(12)                        "b": {"c": 0, "d": 1.0},
(12)                        "c": {"y": 2.0},
(12)                        "d": {"e": 1.0},
(12)                        "e": {"y": 1.0},
(12)                        "y": {},
(8)                     }
(8)                     solnValue = 3.0
(8)                     s = "x"
(8)                     t = "y"
(8)                     compare_flows_and_cuts(G, s, t, solnFlows, solnValue, capacity="spam")
(4)                 def test_digraph_infcap_edges(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", "a")
(8)                     G.add_edge("s", "b", capacity=30)
(8)                     G.add_edge("a", "c", capacity=25)
(8)                     G.add_edge("b", "c", capacity=12)
(8)                     G.add_edge("a", "t", capacity=60)
(8)                     G.add_edge("c", "t")
(8)                     H = {
(12)                        "s": {"a": 85, "b": 12},
(12)                        "a": {"c": 25, "t": 60},
(12)                        "b": {"c": 12},
(12)                        "c": {"t": 37},
(12)                        "t": {},
(8)                     }
(8)                     compare_flows_and_cuts(G, "s", "t", H, 97)
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", "a", capacity=85)
(8)                     G.add_edge("s", "b", capacity=30)
(8)                     G.add_edge("a", "c")
(8)                     G.add_edge("c", "a")
(8)                     G.add_edge("b", "c", capacity=12)
(8)                     G.add_edge("a", "t", capacity=60)
(8)                     G.add_edge("c", "t", capacity=37)
(8)                     H = {
(12)                        "s": {"a": 85, "b": 12},
(12)                        "a": {"c": 25, "t": 60},
(12)                        "c": {"a": 0, "t": 37},
(12)                        "b": {"c": 12},
(12)                        "t": {},
(8)                     }
(8)                     compare_flows_and_cuts(G, "s", "t", H, 97)
(4)                 def test_digraph_infcap_path(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", "a")
(8)                     G.add_edge("s", "b", capacity=30)
(8)                     G.add_edge("a", "c")
(8)                     G.add_edge("b", "c", capacity=12)
```

```
255: (8)                    G.add_edge("a", "t", capacity=60)
256: (8)                    G.add_edge("c", "t")
257: (8)                    for flow_func in all_funcs:
258: (12)                       pytest.raises(nx.NetworkXUnbounded, flow_func, G, "s", "t")
259: (4)                def test_graph_infcap_edges(self):
260: (8)                    G = nx.Graph()
261: (8)                    G.add_edge("s", "a")
262: (8)                    G.add_edge("s", "b", capacity=30)
263: (8)                    G.add_edge("a", "c", capacity=25)
264: (8)                    G.add_edge("b", "c", capacity=12)
265: (8)                    G.add_edge("a", "t", capacity=60)
266: (8)                    G.add_edge("c", "t")
267: (8)                    H = {
268: (12)                       "s": {"a": 85, "b": 12},
269: (12)                       "a": {"c": 25, "s": 85, "t": 60},
270: (12)                       "b": {"c": 12, "s": 12},
271: (12)                       "c": {"a": 25, "b": 12, "t": 37},
272: (12)                       "t": {"a": 60, "c": 37},
273: (8)                    }
274: (8)                    compare_flows_and_cuts(G, "s", "t", H, 97)
275: (4)                def test_digraph5(self):
276: (8)                    G = nx.DiGraph()
277: (8)                    G.add_edge("s", "a", capacity=2)
278: (8)                    G.add_edge("s", "b", capacity=2)
279: (8)                    G.add_edge("a", "b", capacity=5)
280: (8)                    G.add_edge("a", "t", capacity=1)
281: (8)                    G.add_edge("b", "a", capacity=1)
282: (8)                    G.add_edge("b", "t", capacity=3)
283: (8)                    flowSoln = {
284: (12)                       "a": {"b": 1, "t": 1},
285: (12)                       "b": {"a": 0, "t": 3},
286: (12)                       "s": {"a": 2, "b": 2},
287: (12)                       "t": {},
288: (8)                    }
289: (8)                    compare_flows_and_cuts(G, "s", "t", flowSoln, 4)
290: (4)                def test_disconnected(self):
291: (8)                    G = nx.Graph()
292: (8)                    G.add_weighted_edges_from([(0, 1, 1), (1, 2, 1), (2, 3, 1)],
weight="capacity")
293: (8)                    G.remove_node(1)
294: (8)                    assert nx.maximum_flow_value(G, 0, 3) == 0
295: (8)                    flowSoln = {0: {}, 2: {3: 0}, 3: {2: 0}}
296: (8)                    compare_flows_and_cuts(G, 0, 3, flowSoln, 0)
297: (4)                def test_source_target_not_in_graph(self):
298: (8)                    G = nx.Graph()
299: (8)                    G.add_weighted_edges_from([(0, 1, 1), (1, 2, 1), (2, 3, 1)],
weight="capacity")
300: (8)                    G.remove_node(0)
301: (8)                    for flow_func in all_funcs:
302: (12)                       pytest.raises(nx.NetworkXError, flow_func, G, 0, 3)
303: (8)                    G.add_weighted_edges_from([(0, 1, 1), (1, 2, 1), (2, 3, 1)],
weight="capacity")
304: (8)                    G.remove_node(3)
305: (8)                    for flow_func in all_funcs:
306: (12)                       pytest.raises(nx.NetworkXError, flow_func, G, 0, 3)
307: (4)                def test_source_target_coincide(self):
308: (8)                    G = nx.Graph()
309: (8)                    G.add_node(0)
310: (8)                    for flow_func in all_funcs:
311: (12)                       pytest.raises(nx.NetworkXError, flow_func, G, 0, 0)
312: (4)                def test_multigraphs_raise(self):
313: (8)                    G = nx.MultiGraph()
314: (8)                    M = nx.MultiDiGraph()
315: (8)                    G.add_edges_from([(0, 1), (1, 0)], capacity=True)
316: (8)                    for flow_func in all_funcs:
317: (12)                       pytest.raises(nx.NetworkXError, flow_func, G, 0, 0)
318: (0)            class TestMaxFlowMinCutInterface:
319: (4)                def setup_method(self):
320: (8)                    G = nx.DiGraph()
```

```
(8)                 G.add_edge("x", "a", capacity=3.0)
(8)                 G.add_edge("x", "b", capacity=1.0)
(8)                 G.add_edge("a", "c", capacity=3.0)
(8)                 G.add_edge("b", "c", capacity=5.0)
(8)                 G.add_edge("b", "d", capacity=4.0)
(8)                 G.add_edge("d", "e", capacity=2.0)
(8)                 G.add_edge("c", "y", capacity=2.0)
(8)                 G.add_edge("e", "y", capacity=3.0)
(8)                 self.G = G
(8)                 H = nx.DiGraph()
(8)                 H.add_edge(0, 1, capacity=1.0)
(8)                 H.add_edge(1, 2, capacity=1.0)
(8)                 self.H = H
(4)             def test_flow_func_not_callable(self):
(8)                 elements = ["this_should_be_callable", 10, {1, 2, 3}]
(8)                 G = nx.Graph()
(8)                 G.add_weighted_edges_from([(0, 1, 1), (1, 2, 1), (2, 3, 1)],
weight="capacity")
(8)                 for flow_func in interface_funcs:
(12)                    for element in elements:
(16)                        pytest.raises(nx.NetworkXError, flow_func, G, 0, 1,
flow_func=element)
(16)                        pytest.raises(nx.NetworkXError, flow_func, G, 0, 1,
flow_func=element)
(4)             def test_flow_func_parameters(self):
(8)                 G = self.G
(8)                 fv = 3.0
(8)                 for interface_func in interface_funcs:
(12)                    for flow_func in flow_funcs:
(16)                        errmsg = (
(20)                            f"Assertion failed in function: {flow_func.__name__} "
(20)                            f"in interface {interface_func.__name__}"
(16)                        )
(16)                        result = interface_func(G, "x", "y", flow_func=flow_func)
(16)                        if interface_func in max_min_funcs:
(20)                            result = result[0]
(16)                        assert fv == result, errmsg
(4)             def test_minimum_cut_no_cutoff(self):
(8)                 G = self.G
(8)                 pytest.raises(
(12)                    nx.NetworkXError,
(12)                    nx.minimum_cut,
(12)                    G,
(12)                    "x",
(12)                    "y",
(12)                    flow_func=preflow_push,
(12)                    cutoff=1.0,
(8)                 )
(8)                 pytest.raises(
(12)                    nx.NetworkXError,
(12)                    nx.minimum_cut_value,
(12)                    G,
(12)                    "x",
(12)                    "y",
(12)                    flow_func=preflow_push,
(12)                    cutoff=1.0,
(8)                 )
(4)             def test_kwargs(self):
(8)                 G = self.H
(8)                 fv = 1.0
(8)                 to_test = (
(12)                    (shortest_augmenting_path, {"two_phase": True}),
(12)                    (preflow_push, {"global_relabel_freq": 5}),
(8)                 )
(8)                 for interface_func in interface_funcs:
(12)                    for flow_func, kwargs in to_test:
(16)                        errmsg = (
(20)                            f"Assertion failed in function: {flow_func.__name__} "
(20)                            f"in interface {interface_func.__name__}"
```

```
(16)                            )
(16)                            result = interface_func(G, 0, 2, flow_func=flow_func,
**kwargs)
(16)                            if interface_func in max_min_funcs:
(20)                                result = result[0]
(16)                            assert fv == result, errmsg
(4)                 def test_kwargs_default_flow_func(self):
(8)                     G = self.H
(8)                     for interface_func in interface_funcs:
(12)                        pytest.raises(
(16)                            nx.NetworkXError, interface_func, G, 0, 1,
global_relabel_freq=2
(12)                        )
(4)                 def test_reusing_residual(self):
(8)                     G = self.G
(8)                     fv = 3.0
(8)                     s, t = "x", "y"
(8)                     R = build_residual_network(G, "capacity")
(8)                     for interface_func in interface_funcs:
(12)                        for flow_func in flow_funcs:
(16)                            errmsg = (
(20)                                f"Assertion failed in function: {flow_func.__name__} "
(20)                                f"in interface {interface_func.__name__}"
(16)                            )
(16)                            for i in range(3):
(20)                                result = interface_func(
(24)                                    G, "x", "y", flow_func=flow_func, residual=R
(20)                                )
(20)                                if interface_func in max_min_funcs:
(24)                                    result = result[0]
(20)                                assert fv == result, errmsg
(0)             def test_preflow_push_global_relabel_freq():
(4)                 G = nx.DiGraph()
(4)                 G.add_edge(1, 2, capacity=1)
(4)                 R = preflow_push(G, 1, 2, global_relabel_freq=None)
(4)                 assert R.graph["flow_value"] == 1
(4)                 pytest.raises(nx.NetworkXError, preflow_push, G, 1, 2,
global_relabel_freq=-1)
(0)             def test_preflow_push_makes_enough_space():
(4)                 G = nx.DiGraph()
(4)                 nx.add_path(G, [0, 1, 3], capacity=1)
(4)                 nx.add_path(G, [1, 2, 3], capacity=1)
(4)                 R = preflow_push(G, 0, 3, value_only=False)
(4)                 assert R.graph["flow_value"] == 1
(0)             def test_shortest_augmenting_path_two_phase():
(4)                 k = 5
(4)                 p = 1000
(4)                 G = nx.DiGraph()
(4)                 for i in range(k):
(8)                     G.add_edge("s", (i, 0), capacity=1)
(8)                     nx.add_path(G, ((i, j) for j in range(p)), capacity=1)
(8)                     G.add_edge((i, p - 1), "t", capacity=1)
(4)                 R = shortest_augmenting_path(G, "s", "t", two_phase=True)
(4)                 assert R.graph["flow_value"] == k
(4)                 R = shortest_augmenting_path(G, "s", "t", two_phase=False)
(4)                 assert R.graph["flow_value"] == k
(0)             class TestCutoff:
(4)                 def test_cutoff(self):
(8)                     k = 5
(8)                     p = 1000
(8)                     G = nx.DiGraph()
(8)                     for i in range(k):
(12)                        G.add_edge("s", (i, 0), capacity=2)
(12)                        nx.add_path(G, ((i, j) for j in range(p)), capacity=2)
(12)                        G.add_edge((i, p - 1), "t", capacity=2)
(8)                     R = shortest_augmenting_path(G, "s", "t", two_phase=True, cutoff=k)
(8)                     assert k <= R.graph["flow_value"] <= (2 * k)
(8)                     R = shortest_augmenting_path(G, "s", "t", two_phase=False, cutoff=k)
(8)                     assert k <= R.graph["flow_value"] <= (2 * k)
```

```
453: (8)                    R = edmonds_karp(G, "s", "t", cutoff=k)
454: (8)                    assert k <= R.graph["flow_value"] <= (2 * k)
455: (8)                    R = dinitz(G, "s", "t", cutoff=k)
456: (8)                    assert k <= R.graph["flow_value"] <= (2 * k)
457: (8)                    R = boykov_kolmogorov(G, "s", "t", cutoff=k)
458: (8)                    assert k <= R.graph["flow_value"] <= (2 * k)
459: (4)                def test_complete_graph_cutoff(self):
460: (8)                    G = nx.complete_graph(5)
461: (8)                    nx.set_edge_attributes(G, {(u, v): 1 for u, v in G.edges()},
"capacity")
462: (8)                    for flow_func in [
463: (12)                       shortest_augmenting_path,
464: (12)                       edmonds_karp,
465: (12)                       dinitz,
466: (12)                       boykov_kolmogorov,
467: (8)                    ]:
468: (12)                       for cutoff in [3, 2, 1]:
469: (16)                           result = nx.maximum_flow_value(
470: (20)                               G, 0, 4, flow_func=flow_func, cutoff=cutoff
471: (16)                           )
472: (16)                           assert cutoff == result, f"cutoff error in
{flow_func.__name__}"
```

File 250 - test_gomory_hu.py:

```
1: (0)              from itertools import combinations
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.flow import (
5: (4)                  boykov_kolmogorov,
6: (4)                  dinitz,
7: (4)                  edmonds_karp,
8: (4)                  preflow_push,
9: (4)                  shortest_augmenting_path,
10: (0)             )
11: (0)             flow_funcs = [
12: (4)                 boykov_kolmogorov,
13: (4)                 dinitz,
14: (4)                 edmonds_karp,
15: (4)                 preflow_push,
16: (4)                 shortest_augmenting_path,
17: (0)             ]
18: (0)             class TestGomoryHuTree:
19: (4)                 def minimum_edge_weight(self, T, u, v):
20: (8)                     path = nx.shortest_path(T, u, v, weight="weight")
21: (8)                     return min((T[u][v]["weight"], (u, v)) for (u, v) in zip(path,
path[1:]))
22: (4)                 def compute_cutset(self, G, T_orig, edge):
23: (8)                     T = T_orig.copy()
24: (8)                     T.remove_edge(*edge)
25: (8)                     U, V = list(nx.connected_components(T))
26: (8)                     cutset = set()
27: (8)                     for x, nbrs in ((n, G[n]) for n in U):
28: (12)                        cutset.update((x, y) for y in nbrs if y in V)
29: (8)                     return cutset
30: (4)                 def test_default_flow_function_karate_club_graph(self):
31: (8)                     G = nx.karate_club_graph()
32: (8)                     nx.set_edge_attributes(G, 1, "capacity")
33: (8)                     T = nx.gomory_hu_tree(G)
34: (8)                     assert nx.is_tree(T)
35: (8)                     for u, v in combinations(G, 2):
36: (12)                        cut_value, edge = self.minimum_edge_weight(T, u, v)
37: (12)                        assert nx.minimum_cut_value(G, u, v) == cut_value
38: (4)                 def test_karate_club_graph(self):
39: (8)                     G = nx.karate_club_graph()
40: (8)                     nx.set_edge_attributes(G, 1, "capacity")
41: (8)                     for flow_func in flow_funcs:
```

```
42: (12)                        T = nx.gomory_hu_tree(G, flow_func=flow_func)
43: (12)                        assert nx.is_tree(T)
44: (12)                        for u, v in combinations(G, 2):
45: (16)                            cut_value, edge = self.minimum_edge_weight(T, u, v)
46: (16)                            assert nx.minimum_cut_value(G, u, v) == cut_value
47: (4)                 def test_davis_southern_women_graph(self):
48: (8)                     G = nx.davis_southern_women_graph()
49: (8)                     nx.set_edge_attributes(G, 1, "capacity")
50: (8)                     for flow_func in flow_funcs:
51: (12)                        T = nx.gomory_hu_tree(G, flow_func=flow_func)
52: (12)                        assert nx.is_tree(T)
53: (12)                        for u, v in combinations(G, 2):
54: (16)                            cut_value, edge = self.minimum_edge_weight(T, u, v)
55: (16)                            assert nx.minimum_cut_value(G, u, v) == cut_value
56: (4)                 def test_florentine_families_graph(self):
57: (8)                     G = nx.florentine_families_graph()
58: (8)                     nx.set_edge_attributes(G, 1, "capacity")
59: (8)                     for flow_func in flow_funcs:
60: (12)                        T = nx.gomory_hu_tree(G, flow_func=flow_func)
61: (12)                        assert nx.is_tree(T)
62: (12)                        for u, v in combinations(G, 2):
63: (16)                            cut_value, edge = self.minimum_edge_weight(T, u, v)
64: (16)                            assert nx.minimum_cut_value(G, u, v) == cut_value
65: (4)                 @pytest.mark.slow
66: (4)                 def test_les_miserables_graph_cutset(self):
67: (8)                     G = nx.les_miserables_graph()
68: (8)                     nx.set_edge_attributes(G, 1, "capacity")
69: (8)                     for flow_func in flow_funcs:
70: (12)                        T = nx.gomory_hu_tree(G, flow_func=flow_func)
71: (12)                        assert nx.is_tree(T)
72: (12)                        for u, v in combinations(G, 2):
73: (16)                            cut_value, edge = self.minimum_edge_weight(T, u, v)
74: (16)                            assert nx.minimum_cut_value(G, u, v) == cut_value
75: (4)                 def test_karate_club_graph_cutset(self):
76: (8)                     G = nx.karate_club_graph()
77: (8)                     nx.set_edge_attributes(G, 1, "capacity")
78: (8)                     T = nx.gomory_hu_tree(G)
79: (8)                     assert nx.is_tree(T)
80: (8)                     u, v = 0, 33
81: (8)                     cut_value, edge = self.minimum_edge_weight(T, u, v)
82: (8)                     cutset = self.compute_cutset(G, T, edge)
83: (8)                     assert cut_value == len(cutset)
84: (4)                 def test_wikipedia_example(self):
85: (8)                     G = nx.Graph()
86: (8)                     G.add_weighted_edges_from(
87: (12)                        (
88: (16)                            (0, 1, 1),
89: (16)                            (0, 2, 7),
90: (16)                            (1, 2, 1),
91: (16)                            (1, 3, 3),
92: (16)                            (1, 4, 2),
93: (16)                            (2, 4, 4),
94: (16)                            (3, 4, 1),
95: (16)                            (3, 5, 6),
96: (16)                            (4, 5, 2),
97: (12)                        )
98: (8)                     )
99: (8)                     for flow_func in flow_funcs:
100: (12)                       T = nx.gomory_hu_tree(G, capacity="weight", flow_func=flow_func)
101: (12)                       assert nx.is_tree(T)
102: (12)                       for u, v in combinations(G, 2):
103: (16)                           cut_value, edge = self.minimum_edge_weight(T, u, v)
104: (16)                           assert nx.minimum_cut_value(G, u, v, capacity="weight") ==
cut_value
105: (4)                 def test_directed_raises(self):
106: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
107: (12)                        G = nx.DiGraph()
108: (12)                        T = nx.gomory_hu_tree(G)
109: (4)                 def test_empty_raises(self):
```

```
110: (8)                    with pytest.raises(nx.NetworkXError):
111: (12)                       G = nx.empty_graph()
112: (12)                       T = nx.gomory_hu_tree(G)
```

File 251 - test_mincost.py:

```
1: (0)              import bz2
2: (0)              import importlib.resources
3: (0)              import os
4: (0)              import pickle
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              class TestMinCostFlow:
8: (4)                  def test_simple_digraph(self):
9: (8)                      G = nx.DiGraph()
10: (8)                     G.add_node("a", demand=-5)
11: (8)                     G.add_node("d", demand=5)
12: (8)                     G.add_edge("a", "b", weight=3, capacity=4)
13: (8)                     G.add_edge("a", "c", weight=6, capacity=10)
14: (8)                     G.add_edge("b", "d", weight=1, capacity=9)
15: (8)                     G.add_edge("c", "d", weight=2, capacity=5)
16: (8)                     flowCost, H = nx.network_simplex(G)
17: (8)                     soln = {"a": {"b": 4, "c": 1}, "b": {"d": 4}, "c": {"d": 1}, "d": {}}
18: (8)                     assert flowCost == 24
19: (8)                     assert nx.min_cost_flow_cost(G) == 24
20: (8)                     assert H == soln
21: (8)                     assert nx.min_cost_flow(G) == soln
22: (8)                     assert nx.cost_of_flow(G, H) == 24
23: (8)                     flowCost, H = nx.capacity_scaling(G)
24: (8)                     assert flowCost == 24
25: (8)                     assert nx.cost_of_flow(G, H) == 24
26: (8)                     assert H == soln
27: (4)                 def test_negcycle_infcap(self):
28: (8)                     G = nx.DiGraph()
29: (8)                     G.add_node("s", demand=-5)
30: (8)                     G.add_node("t", demand=5)
31: (8)                     G.add_edge("s", "a", weight=1, capacity=3)
32: (8)                     G.add_edge("a", "b", weight=3)
33: (8)                     G.add_edge("c", "a", weight=-6)
34: (8)                     G.add_edge("b", "d", weight=1)
35: (8)                     G.add_edge("d", "c", weight=-2)
36: (8)                     G.add_edge("d", "t", weight=1, capacity=3)
37: (8)                     pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
38: (8)                     pytest.raises(nx.NetworkXUnbounded, nx.capacity_scaling, G)
39: (4)                 def test_sum_demands_not_zero(self):
40: (8)                     G = nx.DiGraph()
41: (8)                     G.add_node("s", demand=-5)
42: (8)                     G.add_node("t", demand=4)
43: (8)                     G.add_edge("s", "a", weight=1, capacity=3)
44: (8)                     G.add_edge("a", "b", weight=3)
45: (8)                     G.add_edge("a", "c", weight=-6)
46: (8)                     G.add_edge("b", "d", weight=1)
47: (8)                     G.add_edge("c", "d", weight=-2)
48: (8)                     G.add_edge("d", "t", weight=1, capacity=3)
49: (8)                     pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
50: (8)                     pytest.raises(nx.NetworkXUnfeasible, nx.capacity_scaling, G)
51: (4)                 def test_no_flow_satisfying_demands(self):
52: (8)                     G = nx.DiGraph()
53: (8)                     G.add_node("s", demand=-5)
54: (8)                     G.add_node("t", demand=5)
55: (8)                     G.add_edge("s", "a", weight=1, capacity=3)
56: (8)                     G.add_edge("a", "b", weight=3)
57: (8)                     G.add_edge("a", "c", weight=-6)
58: (8)                     G.add_edge("b", "d", weight=1)
59: (8)                     G.add_edge("c", "d", weight=-2)
60: (8)                     G.add_edge("d", "t", weight=1, capacity=3)
61: (8)                     pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
```

```
62: (8)                     pytest.raises(nx.NetworkXUnfeasible, nx.capacity_scaling, G)
63: (4)                 def test_transshipment(self):
64: (8)                     G = nx.DiGraph()
65: (8)                     G.add_node("a", demand=1)
66: (8)                     G.add_node("b", demand=-2)
67: (8)                     G.add_node("c", demand=-2)
68: (8)                     G.add_node("d", demand=3)
69: (8)                     G.add_node("e", demand=-4)
70: (8)                     G.add_node("f", demand=-4)
71: (8)                     G.add_node("g", demand=3)
72: (8)                     G.add_node("h", demand=2)
73: (8)                     G.add_node("r", demand=3)
74: (8)                     G.add_edge("a", "c", weight=3)
75: (8)                     G.add_edge("r", "a", weight=2)
76: (8)                     G.add_edge("b", "a", weight=9)
77: (8)                     G.add_edge("r", "c", weight=0)
78: (8)                     G.add_edge("b", "r", weight=-6)
79: (8)                     G.add_edge("c", "d", weight=5)
80: (8)                     G.add_edge("e", "r", weight=4)
81: (8)                     G.add_edge("e", "f", weight=3)
82: (8)                     G.add_edge("h", "b", weight=4)
83: (8)                     G.add_edge("f", "d", weight=7)
84: (8)                     G.add_edge("f", "h", weight=12)
85: (8)                     G.add_edge("g", "d", weight=12)
86: (8)                     G.add_edge("f", "g", weight=-1)
87: (8)                     G.add_edge("h", "g", weight=-10)
88: (8)                     flowCost, H = nx.network_simplex(G)
89: (8)                     soln = {
90: (12)                        "a": {"c": 0},
91: (12)                        "b": {"a": 0, "r": 2},
92: (12)                        "c": {"d": 3},
93: (12)                        "d": {},
94: (12)                        "e": {"r": 3, "f": 1},
95: (12)                        "f": {"d": 0, "g": 3, "h": 2},
96: (12)                        "g": {"d": 0},
97: (12)                        "h": {"b": 0, "g": 0},
98: (12)                        "r": {"a": 1, "c": 1},
99: (8)                     }
100: (8)                    assert flowCost == 41
101: (8)                    assert nx.min_cost_flow_cost(G) == 41
102: (8)                    assert H == soln
103: (8)                    assert nx.min_cost_flow(G) == soln
104: (8)                    assert nx.cost_of_flow(G, H) == 41
105: (8)                    flowCost, H = nx.capacity_scaling(G)
106: (8)                    assert flowCost == 41
107: (8)                    assert nx.cost_of_flow(G, H) == 41
108: (8)                    assert H == soln
109: (4)                def test_max_flow_min_cost(self):
110: (8)                    G = nx.DiGraph()
111: (8)                    G.add_edge("s", "a", bandwidth=6)
112: (8)                    G.add_edge("s", "c", bandwidth=10, cost=10)
113: (8)                    G.add_edge("a", "b", cost=6)
114: (8)                    G.add_edge("b", "d", bandwidth=8, cost=7)
115: (8)                    G.add_edge("c", "d", cost=10)
116: (8)                    G.add_edge("d", "t", bandwidth=5, cost=5)
117: (8)                    soln = {
118: (12)                       "s": {"a": 5, "c": 0},
119: (12)                       "a": {"b": 5},
120: (12)                       "b": {"d": 5},
121: (12)                       "c": {"d": 0},
122: (12)                       "d": {"t": 5},
123: (12)                       "t": {},
124: (8)                    }
125: (8)                    flow = nx.max_flow_min_cost(G, "s", "t", capacity="bandwidth",
weight="cost")
126: (8)                    assert flow == soln
127: (8)                    assert nx.cost_of_flow(G, flow, weight="cost") == 90
128: (8)                    G.add_edge("t", "s", cost=-100)
129: (8)                    flowCost, flow = nx.capacity_scaling(G, capacity="bandwidth",
```

```
weight="cost")
        G.remove_edge("t", "s")
        assert flowCost == -410
        assert flow["t"]["s"] == 5
        del flow["t"]["s"]
        assert flow == soln
        assert nx.cost_of_flow(G, flow, weight="cost") == 90
    def test_digraph1(self):
        G = nx.DiGraph()
        G.add_node(1, demand=-20)
        G.add_node(4, demand=5)
        G.add_node(5, demand=15)
        G.add_edges_from(
            [
                (1, 2, {"capacity": 15, "weight": 4}),
                (1, 3, {"capacity": 8, "weight": 4}),
                (2, 3, {"weight": 2}),
                (2, 4, {"capacity": 4, "weight": 2}),
                (2, 5, {"capacity": 10, "weight": 6}),
                (3, 4, {"capacity": 15, "weight": 1}),
                (3, 5, {"capacity": 5, "weight": 3}),
                (4, 5, {"weight": 2}),
                (5, 3, {"capacity": 4, "weight": 1}),
            ]
        )
        flowCost, H = nx.network_simplex(G)
        soln = {
            1: {2: 12, 3: 8},
            2: {3: 8, 4: 4, 5: 0},
            3: {4: 11, 5: 5},
            4: {5: 10},
            5: {3: 0},
        }
        assert flowCost == 150
        assert nx.min_cost_flow_cost(G) == 150
        assert H == soln
        assert nx.min_cost_flow(G) == soln
        assert nx.cost_of_flow(G, H) == 150
        flowCost, H = nx.capacity_scaling(G)
        assert flowCost == 150
        assert H == soln
        assert nx.cost_of_flow(G, H) == 150
    def test_digraph2(self):
        G = nx.DiGraph()
        G.add_edge("s", 1, capacity=12)
        G.add_edge("s", 2, capacity=6)
        G.add_edge("s", 3, capacity=14)
        G.add_edge(1, 2, capacity=11, weight=4)
        G.add_edge(2, 3, capacity=9, weight=6)
        G.add_edge(1, 4, capacity=5, weight=5)
        G.add_edge(1, 5, capacity=2, weight=12)
        G.add_edge(2, 5, capacity=4, weight=4)
        G.add_edge(2, 6, capacity=2, weight=6)
        G.add_edge(3, 6, capacity=31, weight=3)
        G.add_edge(4, 5, capacity=18, weight=4)
        G.add_edge(5, 6, capacity=9, weight=5)
        G.add_edge(4, "t", capacity=3)
        G.add_edge(5, "t", capacity=7)
        G.add_edge(6, "t", capacity=22)
        flow = nx.max_flow_min_cost(G, "s", "t")
        soln = {
            1: {2: 6, 4: 5, 5: 1},
            2: {3: 6, 5: 4, 6: 2},
            3: {6: 20},
            4: {5: 2, "t": 3},
            5: {6: 0, "t": 7},
            6: {"t": 22},
            "s": {1: 12, 2: 6, 3: 14},
            "t": {},
```

```
(8)                     }
(8)                     assert flow == soln
(8)                     G.add_edge("t", "s", weight=-100)
(8)                     flowCost, flow = nx.capacity_scaling(G)
(8)                     G.remove_edge("t", "s")
(8)                     assert flow["t"]["s"] == 32
(8)                     assert flowCost == -3007
(8)                     del flow["t"]["s"]
(8)                     assert flow == soln
(8)                     assert nx.cost_of_flow(G, flow) == 193
(4)                 def test_digraph3(self):
(8)                     """Combinatorial Optimization: Algorithms and Complexity,
(8)                     Papadimitriou Steiglitz at page 140 has an example, 7.1, but that
(8)                     admits multiple solutions, so I alter it a bit. From ticket #430
(8)                     by mfrasca."""
(8)                     G = nx.DiGraph()
(8)                     G.add_edge("s", "a")
(8)                     G["s"]["a"].update({0: 2, 1: 4})
(8)                     G.add_edge("s", "b")
(8)                     G["s"]["b"].update({0: 2, 1: 1})
(8)                     G.add_edge("a", "b")
(8)                     G["a"]["b"].update({0: 5, 1: 2})
(8)                     G.add_edge("a", "t")
(8)                     G["a"]["t"].update({0: 1, 1: 5})
(8)                     G.add_edge("b", "a")
(8)                     G["b"]["a"].update({0: 1, 1: 3})
(8)                     G.add_edge("b", "t")
(8)                     G["b"]["t"].update({0: 3, 1: 2})
(8)                     "PS.ex.7.1: testing main function"
(8)                     sol = nx.max_flow_min_cost(G, "s", "t", capacity=0, weight=1)
(8)                     flow = sum(v for v in sol["s"].values())
(8)                     assert 4 == flow
(8)                     assert 23 == nx.cost_of_flow(G, sol, weight=1)
(8)                     assert sol["s"] == {"a": 2, "b": 2}
(8)                     assert sol["a"] == {"b": 1, "t": 1}
(8)                     assert sol["b"] == {"a": 0, "t": 3}
(8)                     assert sol["t"] == {}
(8)                     G.add_edge("t", "s")
(8)                     G["t"]["s"].update({1: -100})
(8)                     flowCost, sol = nx.capacity_scaling(G, capacity=0, weight=1)
(8)                     G.remove_edge("t", "s")
(8)                     flow = sum(v for v in sol["s"].values())
(8)                     assert 4 == flow
(8)                     assert sol["t"]["s"] == 4
(8)                     assert flowCost == -377
(8)                     del sol["t"]["s"]
(8)                     assert sol["s"] == {"a": 2, "b": 2}
(8)                     assert sol["a"] == {"b": 1, "t": 1}
(8)                     assert sol["b"] == {"a": 0, "t": 3}
(8)                     assert sol["t"] == {}
(8)                     assert nx.cost_of_flow(G, sol, weight=1) == 23
(4)                 def test_zero_capacity_edges(self):
(8)                     """Address issue raised in ticket #617 by arv."""
(8)                     G = nx.DiGraph()
(8)                     G.add_edges_from(
(12)                        [
(16)                            (1, 2, {"capacity": 1, "weight": 1}),
(16)                            (1, 5, {"capacity": 1, "weight": 1}),
(16)                            (2, 3, {"capacity": 0, "weight": 1}),
(16)                            (2, 5, {"capacity": 1, "weight": 1}),
(16)                            (5, 3, {"capacity": 2, "weight": 1}),
(16)                            (5, 4, {"capacity": 0, "weight": 1}),
(16)                            (3, 4, {"capacity": 2, "weight": 1}),
(12)                        ]
(8)                     )
(8)                     G.nodes[1]["demand"] = -1
(8)                     G.nodes[2]["demand"] = -1
(8)                     G.nodes[4]["demand"] = 2
(8)                     flowCost, H = nx.network_simplex(G)
```

```
        soln = {1: {2: 0, 5: 1}, 2: {3: 0, 5: 1}, 3: {4: 2}, 4: {}, 5: {3: 2, 4: 0}}
        assert flowCost == 6
        assert nx.min_cost_flow_cost(G) == 6
        assert H == soln
        assert nx.min_cost_flow(G) == soln
        assert nx.cost_of_flow(G, H) == 6
        flowCost, H = nx.capacity_scaling(G)
        assert flowCost == 6
        assert H == soln
        assert nx.cost_of_flow(G, H) == 6
    def test_digon(self):
        """Check if digons are handled properly. Taken from ticket
        nodes = [(1, {}), (2, {"demand": -4}), (3, {"demand": 4})]
        edges = [
            (1, 2, {"capacity": 3, "weight": 600000}),
            (2, 1, {"capacity": 2, "weight": 0}),
            (2, 3, {"capacity": 5, "weight": 714285}),
            (3, 2, {"capacity": 2, "weight": 0}),
        ]
        G = nx.DiGraph(edges)
        G.add_nodes_from(nodes)
        flowCost, H = nx.network_simplex(G)
        soln = {1: {2: 0}, 2: {1: 0, 3: 4}, 3: {2: 0}}
        assert flowCost == 2857140
        assert nx.min_cost_flow_cost(G) == 2857140
        assert H == soln
        assert nx.min_cost_flow(G) == soln
        assert nx.cost_of_flow(G, H) == 2857140
        flowCost, H = nx.capacity_scaling(G)
        assert flowCost == 2857140
        assert H == soln
        assert nx.cost_of_flow(G, H) == 2857140
    def test_deadend(self):
        """Check if one-node cycles are handled properly. Taken from ticket
        G = nx.DiGraph()
        G.add_nodes_from(range(5), demand=0)
        G.nodes[4]["demand"] = -13
        G.nodes[3]["demand"] = 13
        G.add_edges_from([(0, 2), (0, 3), (2, 1)], capacity=20, weight=0.1)
        pytest.raises(nx.NetworkXUnfeasible, nx.min_cost_flow, G)
    def test_infinite_capacity_neg_digon(self):
        """An infinite capacity negative cost digon results in an unbounded
        instance."""
        nodes = [(1, {}), (2, {"demand": -4}), (3, {"demand": 4})]
        edges = [
            (1, 2, {"weight": -600}),
            (2, 1, {"weight": 0}),
            (2, 3, {"capacity": 5, "weight": 714285}),
            (3, 2, {"capacity": 2, "weight": 0}),
        ]
        G = nx.DiGraph(edges)
        G.add_nodes_from(nodes)
        pytest.raises(nx.NetworkXUnbounded, nx.network_simplex, G)
        pytest.raises(nx.NetworkXUnbounded, nx.capacity_scaling, G)
    def test_finite_capacity_neg_digon(self):
        """The digon should receive the maximum amount of flow it can handle.
        Taken from ticket #749 by @chuongdo."""
        G = nx.DiGraph()
        G.add_edge("a", "b", capacity=1, weight=-1)
        G.add_edge("b", "a", capacity=1, weight=-1)
        min_cost = -2
        assert nx.min_cost_flow_cost(G) == min_cost
        flowCost, H = nx.capacity_scaling(G)
        assert flowCost == -2
        assert H == {"a": {"b": 1}, "b": {"a": 1}}
        assert nx.cost_of_flow(G, H) == -2
    def test_multidigraph(self):
        """Multidigraphs are acceptable."""
```

```
(8)         G = nx.MultiDiGraph()
(8)         G.add_weighted_edges_from([(1, 2, 1), (2, 3, 2)], weight="capacity")
(8)         flowCost, H = nx.network_simplex(G)
(8)         assert flowCost == 0
(8)         assert H == {1: {2: {0: 0}}, 2: {3: {0: 0}}, 3: {}}
(8)         flowCost, H = nx.capacity_scaling(G)
(8)         assert flowCost == 0
(8)         assert H == {1: {2: {0: 0}}, 2: {3: {0: 0}}, 3: {}}
(4)     def test_negative_selfloops(self):
(8)         """Negative selfloops should cause an exception if uncapacitated and
(8)         always be saturated otherwise.
(8)         """
(8)         G = nx.DiGraph()
(8)         G.add_edge(1, 1, weight=-1)
(8)         pytest.raises(nx.NetworkXUnbounded, nx.network_simplex, G)
(8)         pytest.raises(nx.NetworkXUnbounded, nx.capacity_scaling, G)
(8)         G[1][1]["capacity"] = 2
(8)         flowCost, H = nx.network_simplex(G)
(8)         assert flowCost == -2
(8)         assert H == {1: {1: 2}}
(8)         flowCost, H = nx.capacity_scaling(G)
(8)         assert flowCost == -2
(8)         assert H == {1: {1: 2}}
(8)         G = nx.MultiDiGraph()
(8)         G.add_edge(1, 1, "x", weight=-1)
(8)         G.add_edge(1, 1, "y", weight=1)
(8)         pytest.raises(nx.NetworkXUnbounded, nx.network_simplex, G)
(8)         pytest.raises(nx.NetworkXUnbounded, nx.capacity_scaling, G)
(8)         G[1][1]["x"]["capacity"] = 2
(8)         flowCost, H = nx.network_simplex(G)
(8)         assert flowCost == -2
(8)         assert H == {1: {1: {"x": 2, "y": 0}}}
(8)         flowCost, H = nx.capacity_scaling(G)
(8)         assert flowCost == -2
(8)         assert H == {1: {1: {"x": 2, "y": 0}}}
(4)     def test_bone_shaped(self):
(8)         G = nx.DiGraph()
(8)         G.add_node(0, demand=-4)
(8)         G.add_node(1, demand=2)
(8)         G.add_node(2, demand=2)
(8)         G.add_node(3, demand=4)
(8)         G.add_node(4, demand=-2)
(8)         G.add_node(5, demand=-2)
(8)         G.add_edge(0, 1, capacity=4)
(8)         G.add_edge(0, 2, capacity=4)
(8)         G.add_edge(4, 3, capacity=4)
(8)         G.add_edge(5, 3, capacity=4)
(8)         G.add_edge(0, 3, capacity=0)
(8)         flowCost, H = nx.network_simplex(G)
(8)         assert flowCost == 0
(8)         assert H == {0: {1: 2, 2: 2, 3: 0}, 1: {}, 2: {}, 3: {}, 4: {3: 2}, 5:
{3: 2}}
(8)         flowCost, H = nx.capacity_scaling(G)
(8)         assert flowCost == 0
(8)         assert H == {0: {1: 2, 2: 2, 3: 0}, 1: {}, 2: {}, 3: {}, 4: {3: 2}, 5:
{3: 2}}
(4)     def test_exceptions(self):
(8)         G = nx.Graph()
(8)         pytest.raises(nx.NetworkXNotImplemented, nx.network_simplex, G)
(8)         pytest.raises(nx.NetworkXNotImplemented, nx.capacity_scaling, G)
(8)         G = nx.MultiGraph()
(8)         pytest.raises(nx.NetworkXNotImplemented, nx.network_simplex, G)
(8)         pytest.raises(nx.NetworkXNotImplemented, nx.capacity_scaling, G)
(8)         G = nx.DiGraph()
(8)         pytest.raises(nx.NetworkXError, nx.network_simplex, G)
(8)         G.add_node(0, demand=float("inf"))
(8)         pytest.raises(nx.NetworkXError, nx.network_simplex, G)
(8)         pytest.raises(nx.NetworkXUnfeasible, nx.capacity_scaling, G)
(8)         G.nodes[0]["demand"] = 0
```

```
402: (8)                    G.add_node(1, demand=0)
403: (8)                    G.add_edge(0, 1, weight=-float("inf"))
404: (8)                    pytest.raises(nx.NetworkXError, nx.network_simplex, G)
405: (8)                    pytest.raises(nx.NetworkXUnfeasible, nx.capacity_scaling, G)
406: (8)                    G[0][1]["weight"] = 0
407: (8)                    G.add_edge(0, 0, weight=float("inf"))
408: (8)                    pytest.raises(nx.NetworkXError, nx.network_simplex, G)
409: (8)                    G[0][0]["weight"] = 0
410: (8)                    G[0][1]["capacity"] = -1
411: (8)                    pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
412: (8)                    G[0][1]["capacity"] = 0
413: (8)                    G[0][0]["capacity"] = -1
414: (8)                    pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
415: (4)                def test_large(self):
416: (8)                    fname = (
417: (12)                       importlib.resources.files("networkx.algorithms.flow.tests")
418: (12)                       / "netgen-2.gpickle.bz2"
419: (8)                    )
420: (8)                    with bz2.BZ2File(fname, "rb") as f:
421: (12)                       G = pickle.load(f)
422: (8)                    flowCost, flowDict = nx.network_simplex(G)
423: (8)                    assert 6749969302 == flowCost
424: (8)                    assert 6749969302 == nx.cost_of_flow(G, flowDict)
425: (8)                    flowCost, flowDict = nx.capacity_scaling(G)
426: (8)                    assert 6749969302 == flowCost
427: (8)                    assert 6749969302 == nx.cost_of_flow(G, flowDict)

----------------------------------------

File 252 - test_maxflow_large_graph.py:

1: (0)              """Maximum flow algorithms test suite on large graphs.
2: (0)              """
3: (0)              import bz2
4: (0)              import importlib.resources
5: (0)              import os
6: (0)              import pickle
7: (0)              import pytest
8: (0)              import networkx as nx
9: (0)              from networkx.algorithms.flow import (
10: (4)                 boykov_kolmogorov,
11: (4)                 build_flow_dict,
12: (4)                 build_residual_network,
13: (4)                 dinitz,
14: (4)                 edmonds_karp,
15: (4)                 preflow_push,
16: (4)                 shortest_augmenting_path,
17: (0)             )
18: (0)             flow_funcs = [
19: (4)                 boykov_kolmogorov,
20: (4)                 dinitz,
21: (4)                 edmonds_karp,
22: (4)                 preflow_push,
23: (4)                 shortest_augmenting_path,
24: (0)             ]
25: (0)             def gen_pyramid(N):
26: (4)                 G = nx.DiGraph()
27: (4)                 for i in range(N - 1):
28: (8)                     cap = 1.0 / (i + 2)
29: (8)                     for j in range(i + 1):
30: (12)                        G.add_edge((i, j), (i + 1, j), capacity=cap)
31: (12)                        cap = 1.0 / (i + 1) - cap
32: (12)                        G.add_edge((i, j), (i + 1, j + 1), capacity=cap)
33: (12)                        cap = 1.0 / (i + 2) - cap
34: (4)                 for j in range(N):
35: (8)                     G.add_edge((N - 1, j), "t")
36: (4)                 return G
37: (0)             def read_graph(name):
38: (4)                 fname = (
```

```
39: (8)                     importlib.resources.files("networkx.algorithms.flow.tests")
40: (8)                     / f"{name}.gpickle.bz2"
41: (4)                 )
42: (4)                 with bz2.BZ2File(fname, "rb") as f:
43: (8)                     G = pickle.load(f)
44: (4)                 return G
45: (0)             def validate_flows(G, s, t, soln_value, R, flow_func):
46: (4)                 flow_value = R.graph["flow_value"]
47: (4)                 flow_dict = build_flow_dict(G, R)
48: (4)                 errmsg = f"Assertion failed in function: {flow_func.__name__}"
49: (4)                 assert soln_value == flow_value, errmsg
50: (4)                 assert set(G) == set(flow_dict), errmsg
51: (4)                 for u in G:
52: (8)                     assert set(G[u]) == set(flow_dict[u]), errmsg
53: (4)                 excess = {u: 0 for u in flow_dict}
54: (4)                 for u in flow_dict:
55: (8)                     for v, flow in flow_dict[u].items():
56: (12)                        assert flow <= G[u][v].get("capacity", float("inf")), errmsg
57: (12)                        assert flow >= 0, errmsg
58: (12)                        excess[u] -= flow
59: (12)                        excess[v] += flow
60: (4)                 for u, exc in excess.items():
61: (8)                     if u == s:
62: (12)                        assert exc == -soln_value, errmsg
63: (8)                     elif u == t:
64: (12)                        assert exc == soln_value, errmsg
65: (8)                     else:
66: (12)                        assert exc == 0, errmsg
67: (0)             class TestMaxflowLargeGraph:
68: (4)                 def test_complete_graph(self):
69: (8)                     N = 50
70: (8)                     G = nx.complete_graph(N)
71: (8)                     nx.set_edge_attributes(G, 5, "capacity")
72: (8)                     R = build_residual_network(G, "capacity")
73: (8)                     kwargs = {"residual": R}
74: (8)                     for flow_func in flow_funcs:
75: (12)                        kwargs["flow_func"] = flow_func
76: (12)                        errmsg = f"Assertion failed in function: {flow_func.__name__}"
77: (12)                        flow_value = nx.maximum_flow_value(G, 1, 2, **kwargs)
78: (12)                        assert flow_value == 5 * (N - 1), errmsg
79: (4)                 def test_pyramid(self):
80: (8)                     N = 10
81: (8)                     G = gen_pyramid(N)
82: (8)                     R = build_residual_network(G, "capacity")
83: (8)                     kwargs = {"residual": R}
84: (8)                     for flow_func in flow_funcs:
85: (12)                        kwargs["flow_func"] = flow_func
86: (12)                        errmsg = f"Assertion failed in function: {flow_func.__name__}"
87: (12)                        flow_value = nx.maximum_flow_value(G, (0, 0), "t", **kwargs)
88: (12)                        assert flow_value == pytest.approx(1.0, abs=1e-7)
89: (4)                 def test_gl1(self):
90: (8)                     G = read_graph("gl1")
91: (8)                     s = 1
92: (8)                     t = len(G)
93: (8)                     R = build_residual_network(G, "capacity")
94: (8)                     kwargs = {"residual": R}
95: (8)                     flow_func = flow_funcs[0]
96: (8)                     validate_flows(G, s, t, 156545, flow_func(G, s, t, **kwargs),
flow_func)
97: (4)                 @pytest.mark.slow
98: (4)                 def test_gw1(self):
99: (8)                     G = read_graph("gw1")
100: (8)                    s = 1
101: (8)                    t = len(G)
102: (8)                    R = build_residual_network(G, "capacity")
103: (8)                    kwargs = {"residual": R}
104: (8)                    for flow_func in flow_funcs:
105: (12)                       validate_flows(G, s, t, 1202018, flow_func(G, s, t, **kwargs),
flow_func)
```

```
106: (4)                def test_wlm3(self):
107: (8)                    G = read_graph("wlm3")
108: (8)                    s = 1
109: (8)                    t = len(G)
110: (8)                    R = build_residual_network(G, "capacity")
111: (8)                    kwargs = {"residual": R}
112: (8)                    flow_func = flow_funcs[0]
113: (8)                    validate_flows(G, s, t, 11875108, flow_func(G, s, t, **kwargs),
flow_func)
114: (4)                def test_preflow_push_global_relabel(self):
115: (8)                    G = read_graph("gw1")
116: (8)                    R = preflow_push(G, 1, len(G), global_relabel_freq=50)
117: (8)                    assert R.graph["flow_value"] == 1202018
```

File 253 - test_networksimplex.py:

```
1: (0)              import bz2
2: (0)              import importlib.resources
3: (0)              import os
4: (0)              import pickle
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              @pytest.fixture
8: (0)              def simple_flow_graph():
9: (4)                  G = nx.DiGraph()
10: (4)                 G.add_node("a", demand=0)
11: (4)                 G.add_node("b", demand=-5)
12: (4)                 G.add_node("c", demand=50000000)
13: (4)                 G.add_node("d", demand=-49999995)
14: (4)                 G.add_edge("a", "b", weight=3, capacity=4)
15: (4)                 G.add_edge("a", "c", weight=6, capacity=10)
16: (4)                 G.add_edge("b", "d", weight=1, capacity=9)
17: (4)                 G.add_edge("c", "d", weight=2, capacity=5)
18: (4)                 return G
19: (0)             @pytest.fixture
20: (0)             def simple_no_flow_graph():
21: (4)                 G = nx.DiGraph()
22: (4)                 G.add_node("s", demand=-5)
23: (4)                 G.add_node("t", demand=5)
24: (4)                 G.add_edge("s", "a", weight=1, capacity=3)
25: (4)                 G.add_edge("a", "b", weight=3)
26: (4)                 G.add_edge("a", "c", weight=-6)
27: (4)                 G.add_edge("b", "d", weight=1)
28: (4)                 G.add_edge("c", "d", weight=-2)
29: (4)                 G.add_edge("d", "t", weight=1, capacity=3)
30: (4)                 return G
31: (0)             def get_flowcost_from_flowdict(G, flowDict):
32: (4)                 """Returns flow cost calculated from flow dictionary"""
33: (4)                 flowCost = 0
34: (4)                 for u in flowDict:
35: (8)                     for v in flowDict[u]:
36: (12)                        flowCost += flowDict[u][v] * G[u][v]["weight"]
37: (4)                 return flowCost
38: (0)             def test_infinite_demand_raise(simple_flow_graph):
39: (4)                 G = simple_flow_graph
40: (4)                 inf = float("inf")
41: (4)                 nx.set_node_attributes(G, {"a": {"demand": inf}})
42: (4)                 pytest.raises(nx.NetworkXError, nx.network_simplex, G)
43: (0)             def test_neg_infinite_demand_raise(simple_flow_graph):
44: (4)                 G = simple_flow_graph
45: (4)                 inf = float("inf")
46: (4)                 nx.set_node_attributes(G, {"a": {"demand": -inf}})
47: (4)                 pytest.raises(nx.NetworkXError, nx.network_simplex, G)
48: (0)             def test_infinite_weight_raise(simple_flow_graph):
49: (4)                 G = simple_flow_graph
50: (4)                 inf = float("inf")
51: (4)                 nx.set_edge_attributes(
```

```
52: (8)                     G, {("a", "b"): {"weight": inf}, ("b", "d"): {"weight": inf}}
53: (4)                 )
54: (4)                 pytest.raises(nx.NetworkXError, nx.network_simplex, G)
55: (0)             def test_nonzero_net_demand_raise(simple_flow_graph):
56: (4)                 G = simple_flow_graph
57: (4)                 nx.set_node_attributes(G, {"b": {"demand": -4}})
58: (4)                 pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
59: (0)             def test_negative_capacity_raise(simple_flow_graph):
60: (4)                 G = simple_flow_graph
61: (4)                 nx.set_edge_attributes(G, {("a", "b"): {"weight": 1}, ("b", "d"):
{"capacity": -9}})
62: (4)                 pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
63: (0)             def test_no_flow_satisfying_demands(simple_no_flow_graph):
64: (4)                 G = simple_no_flow_graph
65: (4)                 pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
66: (0)             def test_sum_demands_not_zero(simple_no_flow_graph):
67: (4)                 G = simple_no_flow_graph
68: (4)                 nx.set_node_attributes(G, {"t": {"demand": 4}})
69: (4)                 pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
70: (0)             def test_google_or_tools_example():
71: (4)                 """
72: (4)                 https://developers.google.com/optimization/flow/mincostflow
73: (4)                 """
74: (4)                 G = nx.DiGraph()
75: (4)                 start_nodes = [0, 0, 1, 1, 1, 2, 2, 3, 4]
76: (4)                 end_nodes = [1, 2, 2, 3, 4, 3, 4, 4, 2]
77: (4)                 capacities = [15, 8, 20, 4, 10, 15, 4, 20, 5]
78: (4)                 unit_costs = [4, 4, 2, 2, 6, 1, 3, 2, 3]
79: (4)                 supplies = [20, 0, 0, -5, -15]
80: (4)                 answer = 150
81: (4)                 for i in range(len(supplies)):
82: (8)                     G.add_node(i, demand=(-1) * supplies[i])  # supplies are negative of
demand
83: (4)                 for i in range(len(start_nodes)):
84: (8)                     G.add_edge(
85: (12)                        start_nodes[i], end_nodes[i], weight=unit_costs[i],
capacity=capacities[i]
86: (8)                     )
87: (4)                 flowCost, flowDict = nx.network_simplex(G)
88: (4)                 assert flowCost == answer
89: (4)                 assert flowCost == get_flowcost_from_flowdict(G, flowDict)
90: (0)             def test_google_or_tools_example2():
91: (4)                 """
92: (4)                 https://developers.google.com/optimization/flow/mincostflow
93: (4)                 """
94: (4)                 G = nx.DiGraph()
95: (4)                 start_nodes = [0, 0, 1, 1, 1, 2, 2, 3, 4, 3]
96: (4)                 end_nodes = [1, 2, 2, 3, 4, 3, 4, 4, 2, 5]
97: (4)                 capacities = [15, 8, 20, 4, 10, 15, 4, 20, 5, 10]
98: (4)                 unit_costs = [4, 4, 2, 2, 6, 1, 3, 2, 3, 4]
99: (4)                 supplies = [23, 0, 0, -5, -15, -3]
100: (4)                answer = 183
101: (4)                for i in range(len(supplies)):
102: (8)                    G.add_node(i, demand=(-1) * supplies[i])  # supplies are negative of
demand
103: (4)                for i in range(len(start_nodes)):
104: (8)                    G.add_edge(
105: (12)                       start_nodes[i], end_nodes[i], weight=unit_costs[i],
capacity=capacities[i]
106: (8)                    )
107: (4)                flowCost, flowDict = nx.network_simplex(G)
108: (4)                assert flowCost == answer
109: (4)                assert flowCost == get_flowcost_from_flowdict(G, flowDict)
110: (0)            def test_large():
111: (4)                fname = (
112: (8)                    importlib.resources.files("networkx.algorithms.flow.tests")
113: (8)                    / "netgen-2.gpickle.bz2"
114: (4)                )
115: (4)                with bz2.BZ2File(fname, "rb") as f:
```

```
(8)                     G = pickle.load(f)
(4)                 flowCost, flowDict = nx.network_simplex(G)
(4)                 assert 6749969302 == flowCost
(4)                 assert 6749969302 == nx.cost_of_flow(G, flowDict)
(0)             def test_simple_digraph():
(4)                 G = nx.DiGraph()
(4)                 G.add_node("a", demand=-5)
(4)                 G.add_node("d", demand=5)
(4)                 G.add_edge("a", "b", weight=3, capacity=4)
(4)                 G.add_edge("a", "c", weight=6, capacity=10)
(4)                 G.add_edge("b", "d", weight=1, capacity=9)
(4)                 G.add_edge("c", "d", weight=2, capacity=5)
(4)                 flowCost, H = nx.network_simplex(G)
(4)                 soln = {"a": {"b": 4, "c": 1}, "b": {"d": 4}, "c": {"d": 1}, "d": {}}
(4)                 assert flowCost == 24
(4)                 assert nx.min_cost_flow_cost(G) == 24
(4)                 assert H == soln
(0)             def test_negcycle_infcap():
(4)                 G = nx.DiGraph()
(4)                 G.add_node("s", demand=-5)
(4)                 G.add_node("t", demand=5)
(4)                 G.add_edge("s", "a", weight=1, capacity=3)
(4)                 G.add_edge("a", "b", weight=3)
(4)                 G.add_edge("c", "a", weight=-6)
(4)                 G.add_edge("b", "d", weight=1)
(4)                 G.add_edge("d", "c", weight=-2)
(4)                 G.add_edge("d", "t", weight=1, capacity=3)
(4)                 pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
(0)             def test_transshipment():
(4)                 G = nx.DiGraph()
(4)                 G.add_node("a", demand=1)
(4)                 G.add_node("b", demand=-2)
(4)                 G.add_node("c", demand=-2)
(4)                 G.add_node("d", demand=3)
(4)                 G.add_node("e", demand=-4)
(4)                 G.add_node("f", demand=-4)
(4)                 G.add_node("g", demand=3)
(4)                 G.add_node("h", demand=2)
(4)                 G.add_node("r", demand=3)
(4)                 G.add_edge("a", "c", weight=3)
(4)                 G.add_edge("r", "a", weight=2)
(4)                 G.add_edge("b", "a", weight=9)
(4)                 G.add_edge("r", "c", weight=0)
(4)                 G.add_edge("b", "r", weight=-6)
(4)                 G.add_edge("c", "d", weight=5)
(4)                 G.add_edge("e", "r", weight=4)
(4)                 G.add_edge("e", "f", weight=3)
(4)                 G.add_edge("h", "b", weight=4)
(4)                 G.add_edge("f", "d", weight=7)
(4)                 G.add_edge("f", "h", weight=12)
(4)                 G.add_edge("g", "d", weight=12)
(4)                 G.add_edge("f", "g", weight=-1)
(4)                 G.add_edge("h", "g", weight=-10)
(4)                 flowCost, H = nx.network_simplex(G)
(4)                 soln = {
(8)                     "a": {"c": 0},
(8)                     "b": {"a": 0, "r": 2},
(8)                     "c": {"d": 3},
(8)                     "d": {},
(8)                     "e": {"r": 3, "f": 1},
(8)                     "f": {"d": 0, "g": 3, "h": 2},
(8)                     "g": {"d": 0},
(8)                     "h": {"b": 0, "g": 0},
(8)                     "r": {"a": 1, "c": 1},
(4)                 }
(4)                 assert flowCost == 41
(4)                 assert H == soln
(0)             def test_digraph1():
(4)                 G = nx.DiGraph()
```

```
(4)                 G.add_node(1, demand=-20)
(4)                 G.add_node(4, demand=5)
(4)                 G.add_node(5, demand=15)
(4)                 G.add_edges_from(
(8)                     [
(12)                        (1, 2, {"capacity": 15, "weight": 4}),
(12)                        (1, 3, {"capacity": 8, "weight": 4}),
(12)                        (2, 3, {"weight": 2}),
(12)                        (2, 4, {"capacity": 4, "weight": 2}),
(12)                        (2, 5, {"capacity": 10, "weight": 6}),
(12)                        (3, 4, {"capacity": 15, "weight": 1}),
(12)                        (3, 5, {"capacity": 5, "weight": 3}),
(12)                        (4, 5, {"weight": 2}),
(12)                        (5, 3, {"capacity": 4, "weight": 1}),
(8)                     ]
(4)                 )
(4)                 flowCost, H = nx.network_simplex(G)
(4)                 soln = {
(8)                     1: {2: 12, 3: 8},
(8)                     2: {3: 8, 4: 4, 5: 0},
(8)                     3: {4: 11, 5: 5},
(8)                     4: {5: 10},
(8)                     5: {3: 0},
(4)                 }
(4)                 assert flowCost == 150
(4)                 assert nx.min_cost_flow_cost(G) == 150
(4)                 assert H == soln
(0)             def test_zero_capacity_edges():
(4)                 """Address issue raised in ticket #617 by arv."""
(4)                 G = nx.DiGraph()
(4)                 G.add_edges_from(
(8)                     [
(12)                        (1, 2, {"capacity": 1, "weight": 1}),
(12)                        (1, 5, {"capacity": 1, "weight": 1}),
(12)                        (2, 3, {"capacity": 0, "weight": 1}),
(12)                        (2, 5, {"capacity": 1, "weight": 1}),
(12)                        (5, 3, {"capacity": 2, "weight": 1}),
(12)                        (5, 4, {"capacity": 0, "weight": 1}),
(12)                        (3, 4, {"capacity": 2, "weight": 1}),
(8)                     ]
(4)                 )
(4)                 G.nodes[1]["demand"] = -1
(4)                 G.nodes[2]["demand"] = -1
(4)                 G.nodes[4]["demand"] = 2
(4)                 flowCost, H = nx.network_simplex(G)
(4)                 soln = {1: {2: 0, 5: 1}, 2: {3: 0, 5: 1}, 3: {4: 2}, 4: {}, 5: {3: 2, 4:
0}}
(4)                 assert flowCost == 6
(4)                 assert nx.min_cost_flow_cost(G) == 6
(4)                 assert H == soln
(0)             def test_digon():
(4)                 """Check if digons are handled properly. Taken from ticket
(4)                 nodes = [(1, {}), (2, {"demand": -4}), (3, {"demand": 4})]
(4)                 edges = [
(8)                     (1, 2, {"capacity": 3, "weight": 600000}),
(8)                     (2, 1, {"capacity": 2, "weight": 0}),
(8)                     (2, 3, {"capacity": 5, "weight": 714285}),
(8)                     (3, 2, {"capacity": 2, "weight": 0}),
(4)                 ]
(4)                 G = nx.DiGraph(edges)
(4)                 G.add_nodes_from(nodes)
(4)                 flowCost, H = nx.network_simplex(G)
(4)                 soln = {1: {2: 0}, 2: {1: 0, 3: 4}, 3: {2: 0}}
(4)                 assert flowCost == 2857140
(0)             def test_deadend():
(4)                 """Check if one-node cycles are handled properly. Taken from ticket
(4)                 G = nx.DiGraph()
(4)                 G.add_nodes_from(range(5), demand=0)
(4)                 G.nodes[4]["demand"] = -13
```

```
253: (4)                G.nodes[3]["demand"] = 13
254: (4)                G.add_edges_from([(0, 2), (0, 3), (2, 1)], capacity=20, weight=0.1)
255: (4)                pytest.raises(nx.NetworkXUnfeasible, nx.network_simplex, G)
256: (0)            def test_infinite_capacity_neg_digon():
257: (4)                """An infinite capacity negative cost digon results in an unbounded
258: (4)                instance."""
259: (4)                nodes = [(1, {}), (2, {"demand": -4}), (3, {"demand": 4})]
260: (4)                edges = [
261: (8)                    (1, 2, {"weight": -600}),
262: (8)                    (2, 1, {"weight": 0}),
263: (8)                    (2, 3, {"capacity": 5, "weight": 714285}),
264: (8)                    (3, 2, {"capacity": 2, "weight": 0}),
265: (4)                ]
266: (4)                G = nx.DiGraph(edges)
267: (4)                G.add_nodes_from(nodes)
268: (4)                pytest.raises(nx.NetworkXUnbounded, nx.network_simplex, G)
269: (0)            def test_multidigraph():
270: (4)                """Multidigraphs are acceptable."""
271: (4)                G = nx.MultiDiGraph()
272: (4)                G.add_weighted_edges_from([(1, 2, 1), (2, 3, 2)], weight="capacity")
273: (4)                flowCost, H = nx.network_simplex(G)
274: (4)                assert flowCost == 0
275: (4)                assert H == {1: {2: {0: 0}}, 2: {3: {0: 0}}, 3: {}}
276: (0)            def test_negative_selfloops():
277: (4)                """Negative selfloops should cause an exception if uncapacitated and
278: (4)                always be saturated otherwise.
279: (4)                """
280: (4)                G = nx.DiGraph()
281: (4)                G.add_edge(1, 1, weight=-1)
282: (4)                pytest.raises(nx.NetworkXUnbounded, nx.network_simplex, G)
283: (4)                G[1][1]["capacity"] = 2
284: (4)                flowCost, H = nx.network_simplex(G)
285: (4)                assert flowCost == -2
286: (4)                assert H == {1: {1: 2}}
287: (4)                G = nx.MultiDiGraph()
288: (4)                G.add_edge(1, 1, "x", weight=-1)
289: (4)                G.add_edge(1, 1, "y", weight=1)
290: (4)                pytest.raises(nx.NetworkXUnbounded, nx.network_simplex, G)
291: (4)                G[1][1]["x"]["capacity"] = 2
292: (4)                flowCost, H = nx.network_simplex(G)
293: (4)                assert flowCost == -2
294: (4)                assert H == {1: {1: {"x": 2, "y": 0}}}
295: (0)            def test_bone_shaped():
296: (4)                G = nx.DiGraph()
297: (4)                G.add_node(0, demand=-4)
298: (4)                G.add_node(1, demand=2)
299: (4)                G.add_node(2, demand=2)
300: (4)                G.add_node(3, demand=4)
301: (4)                G.add_node(4, demand=-2)
302: (4)                G.add_node(5, demand=-2)
303: (4)                G.add_edge(0, 1, capacity=4)
304: (4)                G.add_edge(0, 2, capacity=4)
305: (4)                G.add_edge(4, 3, capacity=4)
306: (4)                G.add_edge(5, 3, capacity=4)
307: (4)                G.add_edge(0, 3, capacity=0)
308: (4)                flowCost, H = nx.network_simplex(G)
309: (4)                assert flowCost == 0
310: (4)                assert H == {0: {1: 2, 2: 2, 3: 0}, 1: {}, 2: {}, 3: {}, 4: {3: 2}, 5: {3:
2}}
311: (0)            def test_graphs_type_exceptions():
312: (4)                G = nx.Graph()
313: (4)                pytest.raises(nx.NetworkXNotImplemented, nx.network_simplex, G)
314: (4)                G = nx.MultiGraph()
315: (4)                pytest.raises(nx.NetworkXNotImplemented, nx.network_simplex, G)
316: (4)                G = nx.DiGraph()
317: (4)                pytest.raises(nx.NetworkXError, nx.network_simplex, G)

----------------------------------------
```

File 254 - __init__.py:

```
1: (0)              from networkx.algorithms.isomorphism.isomorph import *
2: (0)              from networkx.algorithms.isomorphism.vf2userfunc import *
3: (0)              from networkx.algorithms.isomorphism.matchhelpers import *
4: (0)              from networkx.algorithms.isomorphism.temporalisomorphvf2 import *
5: (0)              from networkx.algorithms.isomorphism.ismags import *
6: (0)              from networkx.algorithms.isomorphism.tree_isomorphism import *
7: (0)              from networkx.algorithms.isomorphism.vf2pp import *
```

File 255 - ismags.py:

```
1: (0)              """
2: (0)              ISMAGS Algorithm
3: (0)              ================
4: (0)              Provides a Python implementation of the ISMAGS algorithm. [1]_
5: (0)              It is capable of finding (subgraph) isomorphisms between two graphs, taking
the
6: (0)              symmetry of the subgraph into account. In most cases the VF2 algorithm is
7: (0)              faster (at least on small graphs) than this implementation, but in some cases
8: (0)              there is an exponential number of isomorphisms that are symmetrically
9: (0)              equivalent. In that case, the ISMAGS algorithm will provide only one solution
10: (0)             per symmetry group.
11: (0)             >>> petersen = nx.petersen_graph()
12: (0)             >>> ismags = nx.isomorphism.ISMAGS(petersen, petersen)
13: (0)             >>> isomorphisms = list(ismags.isomorphisms_iter(symmetry=False))
14: (0)             >>> len(isomorphisms)
15: (0)             120
16: (0)             >>> isomorphisms = list(ismags.isomorphisms_iter(symmetry=True))
17: (0)             >>> answer = [{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}]
18: (0)             >>> answer == isomorphisms
19: (0)             True
20: (0)             In addition, this implementation also provides an interface to find the
21: (0)             largest common induced subgraph [2]_ between any two graphs, again taking
22: (0)             symmetry into account. Given `graph` and `subgraph` the algorithm will remove
23: (0)             nodes from the `subgraph` until `subgraph` is isomorphic to a subgraph of
24: (0)             `graph`. Since only the symmetry of `subgraph` is taken into account it is
25: (0)             worth thinking about how you provide your graphs:
26: (0)             >>> graph1 = nx.path_graph(4)
27: (0)             >>> graph2 = nx.star_graph(3)
28: (0)             >>> ismags = nx.isomorphism.ISMAGS(graph1, graph2)
29: (0)             >>> ismags.is_isomorphic()
30: (0)             False
31: (0)             >>> largest_common_subgraph = list(ismags.largest_common_subgraph())
32: (0)             >>> answer = [{1: 0, 0: 1, 2: 2}, {2: 0, 1: 1, 3: 2}]
33: (0)             >>> answer == largest_common_subgraph
34: (0)             True
35: (0)             >>> ismags2 = nx.isomorphism.ISMAGS(graph2, graph1)
36: (0)             >>> largest_common_subgraph = list(ismags2.largest_common_subgraph())
37: (0)             >>> answer = [
38: (0)             ...     {1: 0, 0: 1, 2: 2},
39: (0)             ...     {1: 0, 0: 1, 3: 2},
40: (0)             ...     {2: 0, 0: 1, 1: 2},
41: (0)             ...     {2: 0, 0: 1, 3: 2},
42: (0)             ...     {3: 0, 0: 1, 1: 2},
43: (0)             ...     {3: 0, 0: 1, 2: 2},
44: (0)             ... ]
45: (0)             >>> answer == largest_common_subgraph
46: (0)             True
47: (0)             However, when not taking symmetry into account, it doesn't matter:
48: (0)             >>> largest_common_subgraph =
list(ismags.largest_common_subgraph(symmetry=False))
49: (0)             >>> answer = [
50: (0)             ...     {1: 0, 0: 1, 2: 2},
51: (0)             ...     {1: 0, 2: 1, 0: 2},
52: (0)             ...     {2: 0, 1: 1, 3: 2},
53: (0)             ...     {2: 0, 3: 1, 1: 2},
```

```
54: (0)             ...     {1: 0, 0: 1, 2: 3},
55: (0)             ...     {1: 0, 2: 1, 0: 3},
56: (0)             ...     {2: 0, 1: 1, 3: 3},
57: (0)             ...     {2: 0, 3: 1, 1: 3},
58: (0)             ...     {1: 0, 0: 2, 2: 3},
59: (0)             ...     {1: 0, 2: 2, 0: 3},
60: (0)             ...     {2: 0, 1: 2, 3: 3},
61: (0)             ...     {2: 0, 3: 2, 1: 3},
62: (0)             ... ]
63: (0)             >>> answer == largest_common_subgraph
64: (0)             True
65: (0)             >>> largest_common_subgraph =
list(ismags2.largest_common_subgraph(symmetry=False))
66: (0)             >>> answer = [
67: (0)             ...     {1: 0, 0: 1, 2: 2},
68: (0)             ...     {1: 0, 0: 1, 3: 2},
69: (0)             ...     {2: 0, 0: 1, 1: 2},
70: (0)             ...     {2: 0, 0: 1, 3: 2},
71: (0)             ...     {3: 0, 0: 1, 1: 2},
72: (0)             ...     {3: 0, 0: 1, 2: 2},
73: (0)             ...     {1: 1, 0: 2, 2: 3},
74: (0)             ...     {1: 1, 0: 2, 3: 3},
75: (0)             ...     {2: 1, 0: 2, 1: 3},
76: (0)             ...     {2: 1, 0: 2, 3: 3},
77: (0)             ...     {3: 1, 0: 2, 1: 3},
78: (0)             ...     {3: 1, 0: 2, 2: 3},
79: (0)             ... ]
80: (0)             >>> answer == largest_common_subgraph
81: (0)             True
82: (0)             Notes
83: (0)             -----
84: (0)             - The current implementation works for undirected graphs only. The algorithm
85: (2)               in general should work for directed graphs as well though.
86: (0)             - Node keys for both provided graphs need to be fully orderable as well as
87: (2)               hashable.
88: (0)             - Node and edge equality is assumed to be transitive: if A is equal to B, and
89: (2)               B is equal to C, then A is equal to C.
90: (0)             References
91: (0)             ----------
92: (0)             .. [1] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle,
93: (3)                M. Pickavet, "The Index-Based Subgraph Matching Algorithm with General
94: (3)                Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph
95: (3)                Enumeration", PLoS One 9(5): e97896, 2014.
96: (3)                https://doi.org/10.1371/journal.pone.0097896
97: (0)             .. [2] https://en.wikipedia.org/wiki/Maximum_common_induced_subgraph
98: (0)             """
99: (0)             __all__ = ["ISMAGS"]
100: (0)            import itertools
101: (0)            from collections import Counter, defaultdict
102: (0)            from functools import reduce, wraps
103: (0)            def are_all_equal(iterable):
104: (4)                """
105: (4)                Returns ``True`` if and only if all elements in `iterable` are equal; and
106: (4)                ``False`` otherwise.
107: (4)                Parameters
108: (4)                ----------
109: (4)                iterable: collections.abc.Iterable
110: (8)                    The container whose elements will be checked.
111: (4)                Returns
112: (4)                -------
113: (4)                bool
114: (8)                    ``True`` iff all elements in `iterable` compare equal, ``False``
115: (8)                    otherwise.
116: (4)                """
117: (4)                try:
118: (8)                    shape = iterable.shape
119: (4)                except AttributeError:
120: (8)                    pass
121: (4)                else:
```

```
122: (8)                    if len(shape) > 1:
123: (12)                       message = "The function does not works on multidimensional
arrays."
124: (12)                       raise NotImplementedError(message) from None
125: (4)                iterator = iter(iterable)
126: (4)                first = next(iterator, None)
127: (4)                return all(item == first for item in iterator)
128: (0)            def make_partitions(items, test):
129: (4)                """
130: (4)                Partitions items into sets based on the outcome of ``test(item1, item2)``.
131: (4)                Pairs of items for which `test` returns `True` end up in the same set.
132: (4)                Parameters
133: (4)                ----------
134: (4)                items : collections.abc.Iterable[collections.abc.Hashable]
135: (8)                    Items to partition
136: (4)                test : collections.abc.Callable[collections.abc.Hashable,
collections.abc.Hashable]
137: (8)                    A function that will be called with 2 arguments, taken from items.
138: (8)                    Should return `True` if those 2 items need to end up in the same
139: (8)                    partition, and `False` otherwise.
140: (4)                Returns
141: (4)                -------
142: (4)                list[set]
143: (8)                    A list of sets, with each set containing part of the items in `items`,
144: (8)                    such that ``all(test(*pair) for pair in  itertools.combinations(set,
2))
145: (8)                    == True``
146: (4)                Notes
147: (4)                -----
148: (4)                The function `test` is assumed to be transitive: if ``test(a, b)`` and
149: (4)                ``test(b, c)`` return ``True``, then ``test(a, c)`` must also be ``True``.
150: (4)                """
151: (4)                partitions = []
152: (4)                for item in items:
153: (8)                    for partition in partitions:
154: (12)                       p_item = next(iter(partition))
155: (12)                       if test(item, p_item):
156: (16)                           partition.add(item)
157: (16)                           break
158: (8)                    else:  # No break
159: (12)                       partitions.append({item})
160: (4)                return partitions
161: (0)            def partition_to_color(partitions):
162: (4)                """
163: (4)                Creates a dictionary that maps each item in each partition to the index of
164: (4)                the partition to which it belongs.
165: (4)                Parameters
166: (4)                ----------
167: (4)                partitions: collections.abc.Sequence[collections.abc.Iterable]
168: (8)                    As returned by :func:`make_partitions`.
169: (4)                Returns
170: (4)                -------
171: (4)                dict
172: (4)                """
173: (4)                colors = {}
174: (4)                for color, keys in enumerate(partitions):
175: (8)                    for key in keys:
176: (12)                       colors[key] = color
177: (4)                return colors
178: (0)            def intersect(collection_of_sets):
179: (4)                """
180: (4)                Given an collection of sets, returns the intersection of those sets.
181: (4)                Parameters
182: (4)                ----------
183: (4)                collection_of_sets: collections.abc.Collection[set]
184: (8)                    A collection of sets.
185: (4)                Returns
186: (4)                -------
187: (4)                set
```

```
188: (8)                    An intersection of all sets in `collection_of_sets`. Will have the
same
189: (8)                    type as the item initially taken from `collection_of_sets`.
190: (4)                """
191: (4)                collection_of_sets = list(collection_of_sets)
192: (4)                first = collection_of_sets.pop()
193: (4)                out = reduce(set.intersection, collection_of_sets, set(first))
194: (4)                return type(first)(out)
195: (0)            class ISMAGS:
196: (4)                """
197: (4)                Implements the ISMAGS subgraph matching algorithm. [1]_ ISMAGS stands for
198: (4)                "Index-based Subgraph Matching Algorithm with General Symmetries". As the
199: (4)                name implies, it is symmetry aware and will only generate non-symmetric
200: (4)                isomorphisms.
201: (4)                Notes
202: (4)                -----
203: (4)                The implementation imposes additional conditions compared to the VF2
204: (4)                algorithm on the graphs provided and the comparison functions
205: (4)                (:attr:`node_equality` and :attr:`edge_equality`):
206: (5)                 - Node keys in both graphs must be orderable as well as hashable.
207: (5)                 - Equality must be transitive: if A is equal to B, and B is equal to C,
208: (7)                   then A must be equal to C.
209: (4)                Attributes
210: (4)                ----------
211: (4)                graph: networkx.Graph
212: (4)                subgraph: networkx.Graph
213: (4)                node_equality: collections.abc.Callable
214: (8)                    The function called to see if two nodes should be considered equal.
215: (8)                    It's signature looks like this:
216: (8)                    ``f(graph1: networkx.Graph, node1, graph2: networkx.Graph, node2) ->
bool``.
217: (8)                    `node1` is a node in `graph1`, and `node2` a node in `graph2`.
218: (8)                    Constructed from the argument `node_match`.
219: (4)                edge_equality: collections.abc.Callable
220: (8)                    The function called to see if two edges should be considered equal.
221: (8)                    It's signature looks like this:
222: (8)                    ``f(graph1: networkx.Graph, edge1, graph2: networkx.Graph, edge2) ->
bool``.
223: (8)                    `edge1` is an edge in `graph1`, and `edge2` an edge in `graph2`.
224: (8)                    Constructed from the argument `edge_match`.
225: (4)                References
226: (4)                ----------
227: (4)                .. [1] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle,
228: (7)                   M. Pickavet, "The Index-Based Subgraph Matching Algorithm with General
229: (7)                   Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph
230: (7)                   Enumeration", PLoS One 9(5): e97896, 2014.
231: (7)                   https://doi.org/10.1371/journal.pone.0097896
232: (4)                """
233: (4)                def __init__(self, graph, subgraph, node_match=None, edge_match=None,
cache=None):
234: (8)                    """
235: (8)                    Parameters
236: (8)                    ----------
237: (8)                    graph: networkx.Graph
238: (8)                    subgraph: networkx.Graph
239: (8)                    node_match: collections.abc.Callable or None
240: (12)                       Function used to determine whether two nodes are equivalent. Its
241: (12)                       signature should look like ``f(n1: dict, n2: dict) -> bool``, with
242: (12)                       `n1` and `n2` node property dicts. See also
243: (12)                       :func:`~networkx.algorithms.isomorphism.categorical_node_match`
and
244: (12)                       friends.
245: (12)                       If `None`, all nodes are considered equal.
246: (8)                    edge_match: collections.abc.Callable or None
247: (12)                       Function used to determine whether two edges are equivalent. Its
248: (12)                       signature should look like ``f(e1: dict, e2: dict) -> bool``, with
249: (12)                       `e1` and `e2` edge property dicts. See also
250: (12)                       :func:`~networkx.algorithms.isomorphism.categorical_edge_match`
and
```

```
251: (12)                        friends.
252: (12)                        If `None`, all edges are considered equal.
253: (8)                    cache: collections.abc.Mapping
254: (12)                        A cache used for caching graph symmetries.
255: (8)                    """
256: (8)                    self.graph = graph
257: (8)                    self.subgraph = subgraph
258: (8)                    self._symmetry_cache = cache
259: (8)                    self._sgn_partitions_ = None
260: (8)                    self._sge_partitions_ = None
261: (8)                    self._sgn_colors_ = None
262: (8)                    self._sge_colors_ = None
263: (8)                    self._gn_partitions_ = None
264: (8)                    self._ge_partitions_ = None
265: (8)                    self._gn_colors_ = None
266: (8)                    self._ge_colors_ = None
267: (8)                    self._node_compat_ = None
268: (8)                    self._edge_compat_ = None
269: (8)                    if node_match is None:
270: (12)                       self.node_equality = self._node_match_maker(lambda n1, n2: True)
271: (12)                       self._sgn_partitions_ = [set(self.subgraph.nodes)]
272: (12)                       self._gn_partitions_ = [set(self.graph.nodes)]
273: (12)                       self._node_compat_ = {0: 0}
274: (8)                    else:
275: (12)                       self.node_equality = self._node_match_maker(node_match)
276: (8)                    if edge_match is None:
277: (12)                       self.edge_equality = self._edge_match_maker(lambda e1, e2: True)
278: (12)                       self._sge_partitions_ = [set(self.subgraph.edges)]
279: (12)                       self._ge_partitions_ = [set(self.graph.edges)]
280: (12)                       self._edge_compat_ = {0: 0}
281: (8)                    else:
282: (12)                       self.edge_equality = self._edge_match_maker(edge_match)
283: (4)                @property
284: (4)                def _sgn_partitions(self):
285: (8)                    if self._sgn_partitions_ is None:
286: (12)                       def nodematch(node1, node2):
287: (16)                           return self.node_equality(self.subgraph, node1, self.subgraph,
node2)
288: (12)                       self._sgn_partitions_ = make_partitions(self.subgraph.nodes,
nodematch)
289: (8)                    return self._sgn_partitions_
290: (4)                @property
291: (4)                def _sge_partitions(self):
292: (8)                    if self._sge_partitions_ is None:
293: (12)                       def edgematch(edge1, edge2):
294: (16)                           return self.edge_equality(self.subgraph, edge1, self.subgraph,
edge2)
295: (12)                       self._sge_partitions_ = make_partitions(self.subgraph.edges,
edgematch)
296: (8)                    return self._sge_partitions_
297: (4)                @property
298: (4)                def _gn_partitions(self):
299: (8)                    if self._gn_partitions_ is None:
300: (12)                       def nodematch(node1, node2):
301: (16)                           return self.node_equality(self.graph, node1, self.graph,
node2)
302: (12)                       self._gn_partitions_ = make_partitions(self.graph.nodes,
nodematch)
303: (8)                    return self._gn_partitions_
304: (4)                @property
305: (4)                def _ge_partitions(self):
306: (8)                    if self._ge_partitions_ is None:
307: (12)                       def edgematch(edge1, edge2):
308: (16)                           return self.edge_equality(self.graph, edge1, self.graph,
edge2)
309: (12)                       self._ge_partitions_ = make_partitions(self.graph.edges,
edgematch)
310: (8)                    return self._ge_partitions_
311: (4)                @property
```

```
(4)     def _sgn_colors(self):
(8)         if self._sgn_colors_ is None:
(12)            self._sgn_colors_ = partition_to_color(self._sgn_partitions)
(8)         return self._sgn_colors_
(4)     @property
(4)     def _sge_colors(self):
(8)         if self._sge_colors_ is None:
(12)            self._sge_colors_ = partition_to_color(self._sge_partitions)
(8)         return self._sge_colors_
(4)     @property
(4)     def _gn_colors(self):
(8)         if self._gn_colors_ is None:
(12)            self._gn_colors_ = partition_to_color(self._gn_partitions)
(8)         return self._gn_colors_
(4)     @property
(4)     def _ge_colors(self):
(8)         if self._ge_colors_ is None:
(12)            self._ge_colors_ = partition_to_color(self._ge_partitions)
(8)         return self._ge_colors_
(4)     @property
(4)     def _node_compatibility(self):
(8)         if self._node_compat_ is not None:
(12)            return self._node_compat_
(8)         self._node_compat_ = {}
(8)         for sgn_part_color, gn_part_color in itertools.product(
(12)            range(len(self._sgn_partitions)), range(len(self._gn_partitions))
(8)         ):
(12)            sgn = next(iter(self._sgn_partitions[sgn_part_color]))
(12)            gn = next(iter(self._gn_partitions[gn_part_color]))
(12)            if self.node_equality(self.subgraph, sgn, self.graph, gn):
(16)                self._node_compat_[sgn_part_color] = gn_part_color
(8)         return self._node_compat_
(4)     @property
(4)     def _edge_compatibility(self):
(8)         if self._edge_compat_ is not None:
(12)            return self._edge_compat_
(8)         self._edge_compat_ = {}
(8)         for sge_part_color, ge_part_color in itertools.product(
(12)            range(len(self._sge_partitions)), range(len(self._ge_partitions))
(8)         ):
(12)            sge = next(iter(self._sge_partitions[sge_part_color]))
(12)            ge = next(iter(self._ge_partitions[ge_part_color]))
(12)            if self.edge_equality(self.subgraph, sge, self.graph, ge):
(16)                self._edge_compat_[sge_part_color] = ge_part_color
(8)         return self._edge_compat_
(4)     @staticmethod
(4)     def _node_match_maker(cmp):
(8)         @wraps(cmp)
(8)         def comparer(graph1, node1, graph2, node2):
(12)            return cmp(graph1.nodes[node1], graph2.nodes[node2])
(8)         return comparer
(4)     @staticmethod
(4)     def _edge_match_maker(cmp):
(8)         @wraps(cmp)
(8)         def comparer(graph1, edge1, graph2, edge2):
(12)            return cmp(graph1.edges[edge1], graph2.edges[edge2])
(8)         return comparer
(4)     def find_isomorphisms(self, symmetry=True):
(8)         """Find all subgraph isomorphisms between subgraph and graph
(8)         Finds isomorphisms where :attr:`subgraph` <= :attr:`graph`.
(8)         Parameters
(8)         ----------
(8)         symmetry: bool
(12)            Whether symmetry should be taken into account. If False, found
(12)            isomorphisms may be symmetrically equivalent.
(8)         Yields
(8)         ------
(8)         dict
(12)            The found isomorphism mappings of {graph_node: subgraph_node}.
```

```
381: (8)                    """
382: (8)                    if not self.subgraph:
383: (12)                       yield {}
384: (12)                       return
385: (8)                    elif not self.graph:
386: (12)                       return
387: (8)                    elif len(self.graph) < len(self.subgraph):
388: (12)                       return
389: (8)                    if symmetry:
390: (12)                       _, cosets = self.analyze_symmetry(
391: (16)                           self.subgraph, self._sgn_partitions, self._sge_colors
392: (12)                       )
393: (12)                       constraints = self._make_constraints(cosets)
394: (8)                    else:
395: (12)                       constraints = []
396: (8)                    candidates = self._find_nodecolor_candidates()
397: (8)                    la_candidates = self._get_lookahead_candidates()
398: (8)                    for sgn in self.subgraph:
399: (12)                       extra_candidates = la_candidates[sgn]
400: (12)                       if extra_candidates:
401: (16)                           candidates[sgn] = candidates[sgn] |
{frozenset(extra_candidates)}
402: (8)                    if any(candidates.values()):
403: (12)                       start_sgn = min(candidates, key=lambda n: min(candidates[n],
key=len))
404: (12)                       candidates[start_sgn] = (intersect(candidates[start_sgn]),)
405: (12)                       yield from self._map_nodes(start_sgn, candidates, constraints)
406: (8)                    else:
407: (12)                       return
408: (4)                @staticmethod
409: (4)                def _find_neighbor_color_count(graph, node, node_color, edge_color):
410: (8)                    """
411: (8)                    For `node` in `graph`, count the number of edges of a specific color
412: (8)                    it has to nodes of a specific color.
413: (8)                    """
414: (8)                    counts = Counter()
415: (8)                    neighbors = graph[node]
416: (8)                    for neighbor in neighbors:
417: (12)                       n_color = node_color[neighbor]
418: (12)                       if (node, neighbor) in edge_color:
419: (16)                           e_color = edge_color[node, neighbor]
420: (12)                       else:
421: (16)                           e_color = edge_color[neighbor, node]
422: (12)                       counts[e_color, n_color] += 1
423: (8)                    return counts
424: (4)                def _get_lookahead_candidates(self):
425: (8)                    """
426: (8)                    Returns a mapping of {subgraph node: collection of graph nodes} for
427: (8)                    which the graph nodes are feasible candidates for the subgraph node,
as
428: (8)                    determined by looking ahead one edge.
429: (8)                    """
430: (8)                    g_counts = {}
431: (8)                    for gn in self.graph:
432: (12)                       g_counts[gn] = self._find_neighbor_color_count(
433: (16)                           self.graph, gn, self._gn_colors, self._ge_colors
434: (12)                       )
435: (8)                    candidates = defaultdict(set)
436: (8)                    for sgn in self.subgraph:
437: (12)                       sg_count = self._find_neighbor_color_count(
438: (16)                           self.subgraph, sgn, self._sgn_colors, self._sge_colors
439: (12)                       )
440: (12)                       new_sg_count = Counter()
441: (12)                       for (sge_color, sgn_color), count in sg_count.items():
442: (16)                           try:
443: (20)                               ge_color = self._edge_compatibility[sge_color]
444: (20)                               gn_color = self._node_compatibility[sgn_color]
445: (16)                           except KeyError:
446: (20)                               pass
```

```
447: (16)                            else:
448: (20)                                new_sg_count[ge_color, gn_color] = count
449: (12)                        for gn, g_count in g_counts.items():
450: (16)                            if all(new_sg_count[x] <= g_count[x] for x in new_sg_count):
451: (20)                                candidates[sgn].add(gn)
452: (8)                    return candidates
453: (4)                def largest_common_subgraph(self, symmetry=True):
454: (8)                    """
455: (8)                    Find the largest common induced subgraphs between :attr:`subgraph` and
456: (8)                    :attr:`graph`.
457: (8)                    Parameters
458: (8)                    ----------
459: (8)                    symmetry: bool
460: (12)                       Whether symmetry should be taken into account. If False, found
461: (12)                       largest common subgraphs may be symmetrically equivalent.
462: (8)                    Yields
463: (8)                    ------
464: (8)                    dict
465: (12)                       The found isomorphism mappings of {graph_node: subgraph_node}.
466: (8)                    """
467: (8)                    if not self.subgraph:
468: (12)                       yield {}
469: (12)                       return
470: (8)                    elif not self.graph:
471: (12)                       return
472: (8)                    if symmetry:
473: (12)                       _, cosets = self.analyze_symmetry(
474: (16)                           self.subgraph, self._sgn_partitions, self._sge_colors
475: (12)                       )
476: (12)                       constraints = self._make_constraints(cosets)
477: (8)                    else:
478: (12)                       constraints = []
479: (8)                    candidates = self._find_nodecolor_candidates()
480: (8)                    if any(candidates.values()):
481: (12)                       yield from self._largest_common_subgraph(candidates, constraints)
482: (8)                    else:
483: (12)                       return
484: (4)                def analyze_symmetry(self, graph, node_partitions, edge_colors):
485: (8)                    """
486: (8)                    Find a minimal set of permutations and corresponding co-sets that
487: (8)                    describe the symmetry of `graph`, given the node and edge equalities
488: (8)                    given by `node_partitions` and `edge_colors`, respectively.
489: (8)                    Parameters
490: (8)                    ----------
491: (8)                    graph : networkx.Graph
492: (12)                       The graph whose symmetry should be analyzed.
493: (8)                    node_partitions : list of sets
494: (12)                       A list of sets containing node keys. Node keys in the same set
495: (12)                       are considered equivalent. Every node key in `graph` should be in
496: (12)                       exactly one of the sets. If all nodes are equivalent, this should
497: (12)                       be ``[set(graph.nodes)]``.
498: (8)                    edge_colors : dict mapping edges to their colors
499: (12)                       A dict mapping every edge in `graph` to its corresponding color.
500: (12)                       Edges with the same color are considered equivalent. If all edges
501: (12)                       are equivalent, this should be ``{e: 0 for e in graph.edges}``.
502: (8)                    Returns
503: (8)                    -------
504: (8)                    set[frozenset]
505: (12)                       The found permutations. This is a set of frozensets of pairs of
node
506: (12)                       keys which can be exchanged without changing :attr:`subgraph`.
507: (8)                    dict[collections.abc.Hashable, set[collections.abc.Hashable]]
508: (12)                       The found co-sets. The co-sets is a dictionary of
509: (12)                       ``{node key: set of node keys}``.
510: (12)                       Every key-value pair describes which ``values`` can be
interchanged
511: (12)                       without changing nodes less than ``key``.
512: (8)                    """
513: (8)                    if self._symmetry_cache is not None:
```

```
(12)                        key = hash(
(16)                            (
(20)                                tuple(graph.nodes),
(20)                                tuple(graph.edges),
(20)                                tuple(map(tuple, node_partitions)),
(20)                                tuple(edge_colors.items()),
(16)                            )
(12)                        )
(12)                        if key in self._symmetry_cache:
(16)                            return self._symmetry_cache[key]
(8)                     node_partitions = list(
(12)                        self._refine_node_partitions(graph, node_partitions, edge_colors)
(8)                     )
(8)                     assert len(node_partitions) == 1
(8)                     node_partitions = node_partitions[0]
(8)                     permutations, cosets = self._process_ordered_pair_partitions(
(12)                        graph, node_partitions, node_partitions, edge_colors
(8)                     )
(8)                     if self._symmetry_cache is not None:
(12)                        self._symmetry_cache[key] = permutations, cosets
(8)                     return permutations, cosets
(4)                 def is_isomorphic(self, symmetry=False):
(8)                     """
(8)                     Returns True if :attr:`graph` is isomorphic to :attr:`subgraph` and
(8)                     False otherwise.
(8)                     Returns
(8)                     -------
(8)                     bool
(8)                     """
(8)                     return len(self.subgraph) == len(self.graph) and
self.subgraph_is_isomorphic(
(12)                        symmetry
(8)                     )
(4)                 def subgraph_is_isomorphic(self, symmetry=False):
(8)                     """
(8)                     Returns True if a subgraph of :attr:`graph` is isomorphic to
(8)                     :attr:`subgraph` and False otherwise.
(8)                     Returns
(8)                     -------
(8)                     bool
(8)                     """
(8)                     isom = next(self.subgraph_isomorphisms_iter(symmetry=symmetry), None)
(8)                     return isom is not None
(4)                 def isomorphisms_iter(self, symmetry=True):
(8)                     """
(8)                     Does the same as :meth:`find_isomorphisms` if :attr:`graph` and
(8)                     :attr:`subgraph` have the same number of nodes.
(8)                     """
(8)                     if len(self.graph) == len(self.subgraph):
(12)                        yield from self.subgraph_isomorphisms_iter(symmetry=symmetry)
(4)                 def subgraph_isomorphisms_iter(self, symmetry=True):
(8)                     """Alternative name for :meth:`find_isomorphisms`."""
(8)                     return self.find_isomorphisms(symmetry)
(4)                 def _find_nodecolor_candidates(self):
(8)                     """
(8)                     Per node in subgraph find all nodes in graph that have the same color.
(8)                     """
(8)                     candidates = defaultdict(set)
(8)                     for sgn in self.subgraph.nodes:
(12)                        sgn_color = self._sgn_colors[sgn]
(12)                        if sgn_color in self._node_compatibility:
(16)                            gn_color = self._node_compatibility[sgn_color]
(16)                            candidates[sgn].add(frozenset(self._gn_partitions[gn_color]))
(12)                        else:
(16)                            candidates[sgn].add(frozenset())
(8)                     candidates = dict(candidates)
(8)                     for sgn, options in candidates.items():
(12)                        candidates[sgn] = frozenset(options)
(8)                     return candidates
```

```
(4)                 @staticmethod
(4)                 def _make_constraints(cosets):
(8)                     """
(8)                     Turn cosets into constraints.
(8)                     """
(8)                     constraints = []
(8)                     for node_i, node_ts in cosets.items():
(12)                        for node_t in node_ts:
(16)                            if node_i != node_t:
(20)                                constraints.append((node_i, node_t))
(8)                     return constraints
(4)                 @staticmethod
(4)                 def _find_node_edge_color(graph, node_colors, edge_colors):
(8)                     """
(8)                     For every node in graph, come up with a color that combines 1) the
(8)                     color of the node, and 2) the number of edges of a color to each type
(8)                     of node.
(8)                     """
(8)                     counts = defaultdict(lambda: defaultdict(int))
(8)                     for node1, node2 in graph.edges:
(12)                        if (node1, node2) in edge_colors:
(16)                            ecolor = edge_colors[node1, node2]
(12)                        else:
(16)                            ecolor = edge_colors[node2, node1]
(12)                        counts[node1][ecolor, node_colors[node2]] += 1
(12)                        counts[node2][ecolor, node_colors[node1]] += 1
(8)                     node_edge_colors = {}
(8)                     for node in graph.nodes:
(12)                        node_edge_colors[node] = node_colors[node],
set(counts[node].items())
(8)                     return node_edge_colors
(4)                 @staticmethod
(4)                 def _get_permutations_by_length(items):
(8)                     """
(8)                     Get all permutations of items, but only permute items with the same
(8)                     length.
(8)                     >>> found = list(ISMAGS._get_permutations_by_length([[1], [2], [3, 4],
[4, 5]]))
(8)                     >>> answer = [
(8)                     ...     (([1], [2]), ([3, 4], [4, 5])),
(8)                     ...     (([1], [2]), ([4, 5], [3, 4])),
(8)                     ...     (([2], [1]), ([3, 4], [4, 5])),
(8)                     ...     (([2], [1]), ([4, 5], [3, 4])),
(8)                     ... ]
(8)                     >>> found == answer
(8)                     True
(8)                     """
(8)                     by_len = defaultdict(list)
(8)                     for item in items:
(12)                        by_len[len(item)].append(item)
(8)                     yield from itertools.product(
(12)                        *(itertools.permutations(by_len[l]) for l in sorted(by_len))
(8)                     )
(4)                 @classmethod
(4)                 def _refine_node_partitions(cls, graph, node_partitions, edge_colors,
branch=False):
(8)                     """
(8)                     Given a partition of nodes in graph, make the partitions smaller such
(8)                     that all nodes in a partition have 1) the same color, and 2) the same
(8)                     number of edges to specific other partitions.
(8)                     """
(8)                     def equal_color(node1, node2):
(12)                        return node_edge_colors[node1] == node_edge_colors[node2]
(8)                     node_partitions = list(node_partitions)
(8)                     node_colors = partition_to_color(node_partitions)
(8)                     node_edge_colors = cls._find_node_edge_color(graph, node_colors,
edge_colors)
(8)                     if all(
(12)                        are_all_equal(node_edge_colors[node] for node in partition)
```

```
647: (12)                        for partition in node_partitions
648: (8)                    ):
649: (12)                        yield node_partitions
650: (12)                        return
651: (8)                    new_partitions = []
652: (8)                    output = [new_partitions]
653: (8)                    for partition in node_partitions:
654: (12)                        if not are_all_equal(node_edge_colors[node] for node in
partition):
655: (16)                            refined = make_partitions(partition, equal_color)
656: (16)                            if (
657: (20)                                branch
658: (20)                                and len(refined) != 1
659: (20)                                and len({len(r) for r in refined}) != len([len(r) for r in
refined])
660: (16)                            ):
661: (20)                                permutations = cls._get_permutations_by_length(refined)
662: (20)                                new_output = []
663: (20)                                for n_p in output:
664: (24)                                    for permutation in permutations:
665: (28)                                        new_output.append(n_p + list(permutation[0]))
666: (20)                                output = new_output
667: (16)                            else:
668: (20)                                for n_p in output:
669: (24)                                    n_p.extend(sorted(refined, key=len))
670: (12)                        else:
671: (16)                            for n_p in output:
672: (20)                                n_p.append(partition)
673: (8)                    for n_p in output:
674: (12)                        yield from cls._refine_node_partitions(graph, n_p, edge_colors,
branch)
675: (4)                def _edges_of_same_color(self, sgn1, sgn2):
676: (8)                    """
677: (8)                    Returns all edges in :attr:`graph` that have the same colour as the
678: (8)                    edge between sgn1 and sgn2 in :attr:`subgraph`.
679: (8)                    """
680: (8)                    if (sgn1, sgn2) in self._sge_colors:
681: (12)                        sge_color = self._sge_colors[sgn1, sgn2]
682: (8)                    else:
683: (12)                        sge_color = self._sge_colors[sgn2, sgn1]
684: (8)                    if sge_color in self._edge_compatibility:
685: (12)                        ge_color = self._edge_compatibility[sge_color]
686: (12)                        g_edges = self._ge_partitions[ge_color]
687: (8)                    else:
688: (12)                        g_edges = []
689: (8)                    return g_edges
690: (4)                def _map_nodes(self, sgn, candidates, constraints, mapping=None,
to_be_mapped=None):
691: (8)                    """
692: (8)                    Find all subgraph isomorphisms honoring constraints.
693: (8)                    """
694: (8)                    if mapping is None:
695: (12)                        mapping = {}
696: (8)                    else:
697: (12)                        mapping = mapping.copy()
698: (8)                    if to_be_mapped is None:
699: (12)                        to_be_mapped = set(self.subgraph.nodes)
700: (8)                    sgn_candidates = intersect(candidates[sgn])
701: (8)                    candidates[sgn] = frozenset([sgn_candidates])
702: (8)                    for gn in sgn_candidates:
703: (12)                        if gn in mapping.values() or sgn not in to_be_mapped:
704: (16)                            continue  # pragma: no cover
705: (12)                        mapping[sgn] = gn
706: (12)                        if to_be_mapped == set(mapping.keys()):
707: (16)                            yield {v: k for k, v in mapping.items()}
708: (16)                            continue
709: (12)                        left_to_map = to_be_mapped - set(mapping.keys())
710: (12)                        new_candidates = candidates.copy()
711: (12)                        sgn_nbrs = set(self.subgraph[sgn])
```

```
712: (12)                        not_gn_nbrs = set(self.graph.nodes) - set(self.graph[gn])
713: (12)                        for sgn2 in left_to_map:
714: (16)                            if sgn2 not in sgn_nbrs:
715: (20)                                gn2_options = not_gn_nbrs
716: (16)                            else:
717: (20)                                g_edges = self._edges_of_same_color(sgn, sgn2)
718: (20)                                gn2_options = {n for e in g_edges for n in e if gn in e}
719: (16)                            new_candidates[sgn2] = new_candidates[sgn2].union(
720: (20)                                [frozenset(gn2_options)]
721: (16)                            )
722: (16)                            if (sgn, sgn2) in constraints:
723: (20)                                gn2_options = {gn2 for gn2 in self.graph if gn2 > gn}
724: (16)                            elif (sgn2, sgn) in constraints:
725: (20)                                gn2_options = {gn2 for gn2 in self.graph if gn2 < gn}
726: (16)                            else:
727: (20)                                continue  # pragma: no cover
728: (16)                            new_candidates[sgn2] = new_candidates[sgn2].union(
729: (20)                                [frozenset(gn2_options)]
730: (16)                            )
731: (12)                        next_sgn = min(left_to_map, key=lambda n: min(new_candidates[n],
key=len))
732: (12)                        yield from self._map_nodes(
733: (16)                            next_sgn,
734: (16)                            new_candidates,
735: (16)                            constraints,
736: (16)                            mapping=mapping,
737: (16)                            to_be_mapped=to_be_mapped,
738: (12)                        )
739: (4)                    def _largest_common_subgraph(self, candidates, constraints,
to_be_mapped=None):
740: (8)                        """
741: (8)                        Find all largest common subgraphs honoring constraints.
742: (8)                        """
743: (8)                        if to_be_mapped is None:
744: (12)                           to_be_mapped = {frozenset(self.subgraph.nodes)}
745: (8)                        current_size = len(next(iter(to_be_mapped), []))
746: (8)                        found_iso = False
747: (8)                        if current_size <= len(self.graph):
748: (12)                           for nodes in sorted(to_be_mapped, key=sorted):
749: (16)                               next_sgn = min(nodes, key=lambda n: min(candidates[n],
key=len))
750: (16)                               isomorphs = self._map_nodes(
751: (20)                                   next_sgn, candidates, constraints, to_be_mapped=nodes
752: (16)                               )
753: (16)                               try:
754: (20)                                   item = next(isomorphs)
755: (16)                               except StopIteration:
756: (20)                                   pass
757: (16)                               else:
758: (20)                                   yield item
759: (20)                                   yield from isomorphs
760: (20)                                   found_iso = True
761: (8)                        if found_iso or current_size == 1:
762: (12)                           return
763: (8)                        left_to_be_mapped = set()
764: (8)                        for nodes in to_be_mapped:
765: (12)                           for sgn in nodes:
766: (16)                               new_nodes = self._remove_node(sgn, nodes, constraints)
767: (16)                               left_to_be_mapped.add(new_nodes)
768: (8)                        yield from self._largest_common_subgraph(
769: (12)                           candidates, constraints, to_be_mapped=left_to_be_mapped
770: (8)                        )
771: (4)                    @staticmethod
772: (4)                    def _remove_node(node, nodes, constraints):
773: (8)                        """
774: (8)                        Returns a new set where node has been removed from nodes, subject to
775: (8)                        symmetry constraints. We know, that for every constraint we have
776: (8)                        those subgraph nodes are equal. So whenever we would remove the
777: (8)                        lower part of a constraint, remove the higher instead.
```

```
(8)                     """
(8)                     while True:
(12)                        for low, high in constraints:
(16)                            if low == node and high in nodes:
(20)                                node = high
(20)                                break
(12)                        else:  # no break, couldn't find node in constraints
(16)                            break
(8)                     return frozenset(nodes - {node})
(4)                 @staticmethod
(4)                 def _find_permutations(top_partitions, bottom_partitions):
(8)                     """
(8)                     Return the pairs of top/bottom partitions where the partitions are
(8)                     different. Ensures that all partitions in both top and bottom
(8)                     partitions have size 1.
(8)                     """
(8)                     permutations = set()
(8)                     for top, bot in zip(top_partitions, bottom_partitions):
(12)                        if len(top) != 1 or len(bot) != 1:
(16)                            raise IndexError(
(20)                                "Not all nodes are coupled. This is"
(20)                                f" impossible: {top_partitions}, {bottom_partitions}"
(16)                            )
(12)                        if top != bot:
(16)                            permutations.add(frozenset((next(iter(top)),
next(iter(bot)))))
(8)                     return permutations
(4)                 @staticmethod
(4)                 def _update_orbits(orbits, permutations):
(8)                     """
(8)                     Update orbits based on permutations. Orbits is modified in place.
(8)                     For every pair of items in permutations their respective orbits are
(8)                     merged.
(8)                     """
(8)                     for permutation in permutations:
(12)                        node, node2 = permutation
(12)                        first = second = None
(12)                        for idx, orbit in enumerate(orbits):
(16)                            if first is not None and second is not None:
(20)                                break
(16)                            if node in orbit:
(20)                                first = idx
(16)                            if node2 in orbit:
(20)                                second = idx
(12)                        if first != second:
(16)                            orbits[first].update(orbits[second])
(16)                            del orbits[second]
(4)                 def _couple_nodes(
(8)                     self,
(8)                     top_partitions,
(8)                     bottom_partitions,
(8)                     pair_idx,
(8)                     t_node,
(8)                     b_node,
(8)                     graph,
(8)                     edge_colors,
(4)                 ):
(8)                     """
(8)                     Generate new partitions from top and bottom_partitions where t_node is
(8)                     coupled to b_node. pair_idx is the index of the partitions where t_
and
(8)                     b_node can be found.
(8)                     """
(8)                     t_partition = top_partitions[pair_idx]
(8)                     b_partition = bottom_partitions[pair_idx]
(8)                     assert t_node in t_partition and b_node in b_partition
(8)                     new_top_partitions = [top.copy() for top in top_partitions]
(8)                     new_bottom_partitions = [bot.copy() for bot in bottom_partitions]
(8)                     new_t_groups = {t_node}, t_partition - {t_node}
```

```
845: (8)                    new_b_groups = {b_node}, b_partition - {b_node}
846: (8)                    del new_top_partitions[pair_idx]
847: (8)                    del new_bottom_partitions[pair_idx]
848: (8)                    new_top_partitions[pair_idx:pair_idx] = new_t_groups
849: (8)                    new_bottom_partitions[pair_idx:pair_idx] = new_b_groups
850: (8)                    new_top_partitions = self._refine_node_partitions(
851: (12)                       graph, new_top_partitions, edge_colors
852: (8)                    )
853: (8)                    new_bottom_partitions = self._refine_node_partitions(
854: (12)                       graph, new_bottom_partitions, edge_colors, branch=True
855: (8)                    )
856: (8)                    new_top_partitions = list(new_top_partitions)
857: (8)                    assert len(new_top_partitions) == 1
858: (8)                    new_top_partitions = new_top_partitions[0]
859: (8)                    for bot in new_bottom_partitions:
860: (12)                       yield list(new_top_partitions), bot
861: (4)                def _process_ordered_pair_partitions(
862: (8)                    self,
863: (8)                    graph,
864: (8)                    top_partitions,
865: (8)                    bottom_partitions,
866: (8)                    edge_colors,
867: (8)                    orbits=None,
868: (8)                    cosets=None,
869: (4)                ):
870: (8)                    """
871: (8)                    Processes ordered pair partitions as per the reference paper. Finds
and
872: (8)                    returns all permutations and cosets that leave the graph unchanged.
873: (8)                    """
874: (8)                    if orbits is None:
875: (12)                       orbits = [{node} for node in graph.nodes]
876: (8)                    else:
877: (12)                       orbits = orbits
878: (8)                    if cosets is None:
879: (12)                       cosets = {}
880: (8)                    else:
881: (12)                       cosets = cosets.copy()
882: (8)                    assert all(
883: (12)                       len(t_p) == len(b_p) for t_p, b_p in zip(top_partitions,
bottom_partitions)
884: (8)                    )
885: (8)                    if all(len(top) == 1 for top in top_partitions):
886: (12)                       permutations = self._find_permutations(top_partitions,
bottom_partitions)
887: (12)                       self._update_orbits(orbits, permutations)
888: (12)                       if permutations:
889: (16)                           return [permutations], cosets
890: (12)                       else:
891: (16)                           return [], cosets
892: (8)                    permutations = []
893: (8)                    unmapped_nodes = {
894: (12)                       (node, idx)
895: (12)                       for idx, t_partition in enumerate(top_partitions)
896: (12)                       for node in t_partition
897: (12)                       if len(t_partition) > 1
898: (8)                    }
899: (8)                    node, pair_idx = min(unmapped_nodes)
900: (8)                    b_partition = bottom_partitions[pair_idx]
901: (8)                    for node2 in sorted(b_partition):
902: (12)                       if len(b_partition) == 1:
903: (16)                           continue
904: (12)                       if node != node2 and any(
905: (16)                           node in orbit and node2 in orbit for orbit in orbits
906: (12)                       ):
907: (16)                           continue
908: (12)                       partitions = self._couple_nodes(
909: (16)                           top_partitions,
910: (16)                           bottom_partitions,
```

```
911: (16)                            pair_idx,
912: (16)                            node,
913: (16)                            node2,
914: (16)                            graph,
915: (16)                            edge_colors,
916: (12)                        )
917: (12)                        for opp in partitions:
918: (16)                            new_top_partitions, new_bottom_partitions = opp
919: (16)                            new_perms, new_cosets = self._process_ordered_pair_partitions(
920: (20)                                graph,
921: (20)                                new_top_partitions,
922: (20)                                new_bottom_partitions,
923: (20)                                edge_colors,
924: (20)                                orbits,
925: (20)                                cosets,
926: (16)                            )
927: (16)                            permutations += new_perms
928: (16)                            cosets.update(new_cosets)
929: (8)                    mapped = {
930: (12)                        k
931: (12)                        for top, bottom in zip(top_partitions, bottom_partitions)
932: (12)                        for k in top
933: (12)                        if len(top) == 1 and top == bottom
934: (8)                    }
935: (8)                    ks = {k for k in graph.nodes if k < node}
936: (8)                    find_coset = ks <= mapped and node not in cosets
937: (8)                    if find_coset:
938: (12)                       for orbit in orbits:
939: (16)                           if node in orbit:
940: (20)                               cosets[node] = orbit.copy()
941: (8)                    return permutations, cosets
```

File 256 - isomorph.py:

```
1: (0)              """
2: (0)              Graph isomorphism functions.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.exception import NetworkXError
6: (0)              __all__ = [
7: (4)                  "could_be_isomorphic",
8: (4)                  "fast_could_be_isomorphic",
9: (4)                  "faster_could_be_isomorphic",
10: (4)                 "is_isomorphic",
11: (0)             ]
12: (0)             @nx._dispatchable(graphs={"G1": 0, "G2": 1})
13: (0)             def could_be_isomorphic(G1, G2):
14: (4)                 """Returns False if graphs are definitely not isomorphic.
15: (4)                 True does NOT guarantee isomorphism.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G1, G2 : graphs
19: (7)                    The two graphs G1 and G2 must be the same type.
20: (4)                 Notes
21: (4)                 -----
22: (4)                 Checks for matching degree, triangle, and number of cliques sequences.
23: (4)                 The triangle sequence contains the number of triangles each node is part
of.
24: (4)                 The clique sequence contains for each node the number of maximal cliques
25: (4)                 involving that node.
26: (4)                 """
27: (4)                 if G1.order() != G2.order():
28: (8)                     return False
29: (4)                 d1 = G1.degree()
30: (4)                 t1 = nx.triangles(G1)
31: (4)                 clqs_1 = list(nx.find_cliques(G1))
32: (4)                 c1 = {n: sum(1 for c in clqs_1 if n in c) for n in G1}  # number of
```

```
cliques
33: (4)                 props1 = [[d, t1[v], c1[v]] for v, d in d1]
34: (4)                 props1.sort()
35: (4)                 d2 = G2.degree()
36: (4)                 t2 = nx.triangles(G2)
37: (4)                 clqs_2 = list(nx.find_cliques(G2))
38: (4)                 c2 = {n: sum(1 for c in clqs_2 if n in c) for n in G2}  # number of
cliques
39: (4)                 props2 = [[d, t2[v], c2[v]] for v, d in d2]
40: (4)                 props2.sort()
41: (4)                 if props1 != props2:
42: (8)                     return False
43: (4)                 return True
44: (0)             graph_could_be_isomorphic = could_be_isomorphic
45: (0)             @nx._dispatchable(graphs={"G1": 0, "G2": 1})
46: (0)             def fast_could_be_isomorphic(G1, G2):
47: (4)                 """Returns False if graphs are definitely not isomorphic.
48: (4)                 True does NOT guarantee isomorphism.
49: (4)                 Parameters
50: (4)                 ----------
51: (4)                 G1, G2 : graphs
52: (7)                    The two graphs G1 and G2 must be the same type.
53: (4)                 Notes
54: (4)                 -----
55: (4)                 Checks for matching degree and triangle sequences. The triangle
56: (4)                 sequence contains the number of triangles each node is part of.
57: (4)                 """
58: (4)                 if G1.order() != G2.order():
59: (8)                     return False
60: (4)                 d1 = G1.degree()
61: (4)                 t1 = nx.triangles(G1)
62: (4)                 props1 = [[d, t1[v]] for v, d in d1]
63: (4)                 props1.sort()
64: (4)                 d2 = G2.degree()
65: (4)                 t2 = nx.triangles(G2)
66: (4)                 props2 = [[d, t2[v]] for v, d in d2]
67: (4)                 props2.sort()
68: (4)                 if props1 != props2:
69: (8)                     return False
70: (4)                 return True
71: (0)             fast_graph_could_be_isomorphic = fast_could_be_isomorphic
72: (0)             @nx._dispatchable(graphs={"G1": 0, "G2": 1})
73: (0)             def faster_could_be_isomorphic(G1, G2):
74: (4)                 """Returns False if graphs are definitely not isomorphic.
75: (4)                 True does NOT guarantee isomorphism.
76: (4)                 Parameters
77: (4)                 ----------
78: (4)                 G1, G2 : graphs
79: (7)                    The two graphs G1 and G2 must be the same type.
80: (4)                 Notes
81: (4)                 -----
82: (4)                 Checks for matching degree sequences.
83: (4)                 """
84: (4)                 if G1.order() != G2.order():
85: (8)                     return False
86: (4)                 d1 = sorted(d for n, d in G1.degree())
87: (4)                 d2 = sorted(d for n, d in G2.degree())
88: (4)                 if d1 != d2:
89: (8)                     return False
90: (4)                 return True
91: (0)             faster_graph_could_be_isomorphic = faster_could_be_isomorphic
92: (0)             @nx._dispatchable(
93: (4)                 graphs={"G1": 0, "G2": 1},
94: (4)                 preserve_edge_attrs="edge_match",
95: (4)                 preserve_node_attrs="node_match",
96: (0)             )
97: (0)             def is_isomorphic(G1, G2, node_match=None, edge_match=None):
98: (4)                 """Returns True if the graphs G1 and G2 are isomorphic and False
otherwise.
```

```
(4)                 Parameters
(4)                 ----------
(4)                 G1, G2: graphs
(8)                     The two graphs G1 and G2 must be the same type.
(4)                 node_match : callable
(8)                     A function that returns True if node n1 in G1 and n2 in G2 should
(8)                     be considered equal during the isomorphism test.
(8)                     If node_match is not specified then node attributes are not
considered.
(8)                     The function will be called like
(11)                       node_match(G1.nodes[n1], G2.nodes[n2]).
(8)                     That is, the function will receive the node attribute dictionaries
(8)                     for n1 and n2 as inputs.
(4)                 edge_match : callable
(8)                     A function that returns True if the edge attribute dictionary
(8)                     for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should
(8)                     be considered equal during the isomorphism test.  If edge_match is
(8)                     not specified then edge attributes are not considered.
(8)                     The function will be called like
(11)                       edge_match(G1[u1][v1], G2[u2][v2]).
(8)                     That is, the function will receive the edge attribute dictionaries
(8)                     of the edges under consideration.
(4)                 Notes
(4)                 -----
(4)                 Uses the vf2 algorithm [1]_.
(4)                 Examples
(4)                 --------
(4)                 >>> import networkx.algorithms.isomorphism as iso
(4)                 For digraphs G1 and G2, using 'weight' edge attribute (default: 1)
(4)                 >>> G1 = nx.DiGraph()
(4)                 >>> G2 = nx.DiGraph()
(4)                 >>> nx.add_path(G1, [1, 2, 3, 4], weight=1)
(4)                 >>> nx.add_path(G2, [10, 20, 30, 40], weight=2)
(4)                 >>> em = iso.numerical_edge_match("weight", 1)
(4)                 >>> nx.is_isomorphic(G1, G2)  # no weights considered
(4)                 True
(4)                 >>> nx.is_isomorphic(G1, G2, edge_match=em)  # match weights
(4)                 False
(4)                 For multidigraphs G1 and G2, using 'fill' node attribute (default: '')
(4)                 >>> G1 = nx.MultiDiGraph()
(4)                 >>> G2 = nx.MultiDiGraph()
(4)                 >>> G1.add_nodes_from([1, 2, 3], fill="red")
(4)                 >>> G2.add_nodes_from([10, 20, 30, 40], fill="red")
(4)                 >>> nx.add_path(G1, [1, 2, 3, 4], weight=3, linewidth=2.5)
(4)                 >>> nx.add_path(G2, [10, 20, 30, 40], weight=3)
(4)                 >>> nm = iso.categorical_node_match("fill", "red")
(4)                 >>> nx.is_isomorphic(G1, G2, node_match=nm)
(4)                 True
(4)                 For multidigraphs G1 and G2, using 'weight' edge attribute (default: 7)
(4)                 >>> G1.add_edge(1, 2, weight=7)
(4)                 1
(4)                 >>> G2.add_edge(10, 20)
(4)                 1
(4)                 >>> em = iso.numerical_multiedge_match("weight", 7, rtol=1e-6)
(4)                 >>> nx.is_isomorphic(G1, G2, edge_match=em)
(4)                 True
(4)                 For multigraphs G1 and G2, using 'weight' and 'linewidth' edge attributes
(4)                 with default values 7 and 2.5. Also using 'fill' node attribute with
(4)                 default value 'red'.
(4)                 >>> em = iso.numerical_multiedge_match(["weight", "linewidth"], [7, 2.5])
(4)                 >>> nm = iso.categorical_node_match("fill", "red")
(4)                 >>> nx.is_isomorphic(G1, G2, edge_match=em, node_match=nm)
(4)                 True
(4)                 See Also
(4)                 --------
(4)                 numerical_node_match, numerical_edge_match, numerical_multiedge_match
(4)                 categorical_node_match, categorical_edge_match,
categorical_multiedge_match
(4)                 References
```

```
166: (4)                ----------
167: (4)                .. [1]  L. P. Cordella, P. Foggia, C. Sansone, M. Vento,
168: (7)                   "An Improved Algorithm for Matching Large Graphs",
169: (7)                   3rd IAPR-TC15 Workshop  on Graph-based Representations in
170: (7)                   Pattern Recognition, Cuen, pp. 149-159, 2001.
171: (7)
https://www.researchgate.net/publication/200034365_An_Improved_Algorithm_for_Matching_Large_Graphs
172: (4)                """
173: (4)                if G1.is_directed() and G2.is_directed():
174: (8)                    GM = nx.algorithms.isomorphism.DiGraphMatcher
175: (4)                elif (not G1.is_directed()) and (not G2.is_directed()):
176: (8)                    GM = nx.algorithms.isomorphism.GraphMatcher
177: (4)                else:
178: (8)                    raise NetworkXError("Graphs G1 and G2 are not of the same type.")
179: (4)                gm = GM(G1, G2, node_match=node_match, edge_match=edge_match)
180: (4)                return gm.is_isomorphic()

----------------------------------------

File 257 - isomorphvf2.py:

1: (0)              """
2: (0)              *************
3: (0)              VF2 Algorithm
4: (0)              *************
5: (0)              An implementation of VF2 algorithm for graph isomorphism testing.
6: (0)              The simplest interface to use this module is to call the
7: (0)              :func:`is_isomorphic <networkx.algorithms.isomorphism.is_isomorphic>`
8: (0)              function.
9: (0)              Introduction
10: (0)             ------------
11: (0)             The GraphMatcher and DiGraphMatcher are responsible for matching
12: (0)             graphs or directed graphs in a predetermined manner.  This
13: (0)             usually means a check for an isomorphism, though other checks
14: (0)             are also possible.  For example, a subgraph of one graph
15: (0)             can be checked for isomorphism to a second graph.
16: (0)             Matching is done via syntactic feasibility. It is also possible
17: (0)             to check for semantic feasibility. Feasibility, then, is defined
18: (0)             as the logical AND of the two functions.
19: (0)             To include a semantic check, the (Di)GraphMatcher class should be
20: (0)             subclassed, and the
21: (0)             :meth:`semantic_feasibility
<networkx.algorithms.isomorphism.GraphMatcher.semantic_feasibility>`
22: (0)             function should be redefined.  By default, the semantic feasibility function
always
23: (0)             returns ``True``.  The effect of this is that semantics are not
24: (0)             considered in the matching of G1 and G2.
25: (0)             Examples
26: (0)             --------
27: (0)             Suppose G1 and G2 are isomorphic graphs. Verification is as follows:
28: (0)             >>> from networkx.algorithms import isomorphism
29: (0)             >>> G1 = nx.path_graph(4)
30: (0)             >>> G2 = nx.path_graph(4)
31: (0)             >>> GM = isomorphism.GraphMatcher(G1, G2)
32: (0)             >>> GM.is_isomorphic()
33: (0)             True
34: (0)             GM.mapping stores the isomorphism mapping from G1 to G2.
35: (0)             >>> GM.mapping
36: (0)             {0: 0, 1: 1, 2: 2, 3: 3}
37: (0)             Suppose G1 and G2 are isomorphic directed graphs.
38: (0)             Verification is as follows:
39: (0)             >>> G1 = nx.path_graph(4, create_using=nx.DiGraph())
40: (0)             >>> G2 = nx.path_graph(4, create_using=nx.DiGraph())
41: (0)             >>> DiGM = isomorphism.DiGraphMatcher(G1, G2)
42: (0)             >>> DiGM.is_isomorphic()
43: (0)             True
44: (0)             DiGM.mapping stores the isomorphism mapping from G1 to G2.
45: (0)             >>> DiGM.mapping
46: (0)             {0: 0, 1: 1, 2: 2, 3: 3}
```

```
47: (0)             Subgraph Isomorphism
48: (0)             --------------------
49: (0)             Graph theory literature can be ambiguous about the meaning of the
50: (0)             above statement, and we seek to clarify it now.
51: (0)             In the VF2 literature, a mapping `M` is said to be a graph-subgraph
52: (0)             isomorphism iff `M` is an isomorphism between `G2` and a subgraph of `G1`.
53: (0)             Thus, to say that `G1` and `G2` are graph-subgraph isomorphic is to say
54: (0)             that a subgraph of `G1` is isomorphic to `G2`.
55: (0)             Other literature uses the phrase 'subgraph isomorphic' as in '`G1` does
56: (0)             not have a subgraph isomorphic to `G2`'.  Another use is as an in adverb
57: (0)             for isomorphic.  Thus, to say that `G1` and `G2` are subgraph isomorphic
58: (0)             is to say that a subgraph of `G1` is isomorphic to `G2`.
59: (0)             Finally, the term 'subgraph' can have multiple meanings. In this
60: (0)             context, 'subgraph' always means a 'node-induced subgraph'. Edge-induced
61: (0)             subgraph isomorphisms are not directly supported, but one should be
62: (0)             able to perform the check by making use of
63: (0)             :func:`line_graph <networkx.generators.line.line_graph>`. For
64: (0)             subgraphs which are not induced, the term 'monomorphism' is preferred
65: (0)             over 'isomorphism'.
66: (0)             Let ``G = (N, E)`` be a graph with a set of nodes `N` and set of edges `E`.
67: (0)             If ``G' = (N', E')`` is a subgraph, then:
68: (4)                 `N'` is a subset of `N` and
69: (4)                 `E'` is a subset of `E`.
70: (0)             If ``G' = (N', E')`` is a node-induced subgraph, then:
71: (4)                 `N'` is a subset of `N` and
72: (4)                 `E'` is the subset of edges in `E` relating nodes in `N'`.
73: (0)             If `G' = (N', E')` is an edge-induced subgraph, then:
74: (4)                 `N'` is the subset of nodes in `N` related by edges in `E'` and
75: (4)                 `E'` is a subset of `E`.
76: (0)             If `G' = (N', E')` is a monomorphism, then:
77: (4)                 `N'` is a subset of `N` and
78: (4)                 `E'` is a subset of the set of edges in `E` relating nodes in `N'`.
79: (0)             Note that if `G'` is a node-induced subgraph of `G`, then it is always a
80: (0)             subgraph monomorphism of `G`, but the opposite is not always true, as a
81: (0)             monomorphism can have fewer edges.
82: (0)             References
83: (0)             ----------
84: (0)             [1]   Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento,
85: (6)                   "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs",
86: (6)                   IEEE Transactions on Pattern Analysis and Machine Intelligence,
87: (6)                   vol. 26,  no. 10,  pp. 1367-1372,  Oct.,  2004.
88: (6)                   http://ieeexplore.ieee.org/iel5/34/29305/01323804.pdf
89: (0)             [2]   L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "An Improved
90: (6)                   Algorithm for Matching Large Graphs", 3rd IAPR-TC15 Workshop
91: (6)                   on Graph-based Representations in Pattern Recognition, Cuen,
92: (6)                   pp. 149-159, 2001.
93: (6)
https://www.researchgate.net/publication/200034365_An_Improved_Algorithm_for_Matching_Large_Graphs
94: (0)             See Also
95: (0)             --------
96: (0)             :meth:`semantic_feasibility
<networkx.algorithms.isomorphism.GraphMatcher.semantic_feasibility>`
97: (0)             :meth:`syntactic_feasibility
<networkx.algorithms.isomorphism.GraphMatcher.syntactic_feasibility>`
98: (0)             Notes
99: (0)             -----
100: (0)            The implementation handles both directed and undirected graphs as well
101: (0)            as multigraphs.
102: (0)            In general, the subgraph isomorphism problem is NP-complete whereas the
103: (0)            graph isomorphism problem is most likely not NP-complete (although no
104: (0)            polynomial-time algorithm is known to exist).
105: (0)            """
106: (0)            import sys
107: (0)            __all__ = ["GraphMatcher", "DiGraphMatcher"]
108: (0)            class GraphMatcher:
109: (4)                """Implementation of VF2 algorithm for matching undirected graphs.
110: (4)                Suitable for Graph and MultiGraph instances.
111: (4)                """
112: (4)                def __init__(self, G1, G2):
```

```
(8)                 """Initialize GraphMatcher.
(8)                 Parameters
(8)                 ----------
(8)                 G1,G2: NetworkX Graph or MultiGraph instances.
(11)                   The two graphs to check for isomorphism or monomorphism.
(8)                 Examples
(8)                 --------
(8)                 To create a GraphMatcher which checks for syntactic feasibility:
(8)                 >>> from networkx.algorithms import isomorphism
(8)                 >>> G1 = nx.path_graph(4)
(8)                 >>> G2 = nx.path_graph(4)
(8)                 >>> GM = isomorphism.GraphMatcher(G1, G2)
(8)                 """
(8)                 self.G1 = G1
(8)                 self.G2 = G2
(8)                 self.G1_nodes = set(G1.nodes())
(8)                 self.G2_nodes = set(G2.nodes())
(8)                 self.G2_node_order = {n: i for i, n in enumerate(G2)}
(8)                 self.old_recursion_limit = sys.getrecursionlimit()
(8)                 expected_max_recursion_level = len(self.G2)
(8)                 if self.old_recursion_limit < 1.5 * expected_max_recursion_level:
(12)                    sys.setrecursionlimit(int(1.5 * expected_max_recursion_level))
(8)                 self.test = "graph"
(8)                 self.initialize()
(4)             def reset_recursion_limit(self):
(8)                 """Restores the recursion limit."""
(8)                 sys.setrecursionlimit(self.old_recursion_limit)
(4)             def candidate_pairs_iter(self):
(8)                 """Iterator over candidate pairs of nodes in G1 and G2."""
(8)                 G1_nodes = self.G1_nodes
(8)                 G2_nodes = self.G2_nodes
(8)                 min_key = self.G2_node_order.__getitem__
(8)                 T1_inout = [node for node in self.inout_1 if node not in self.core_1]
(8)                 T2_inout = [node for node in self.inout_2 if node not in self.core_2]
(8)                 if T1_inout and T2_inout:
(12)                    node_2 = min(T2_inout, key=min_key)
(12)                    for node_1 in T1_inout:
(16)                        yield node_1, node_2
(8)                 else:
(12)                    if 1:  # as inferred from [1], correct
(16)                        other_node = min(G2_nodes - set(self.core_2), key=min_key)
(16)                        for node in self.G1:
(20)                            if node not in self.core_1:
(24)                                yield node, other_node
(4)             def initialize(self):
(8)                 """Reinitializes the state of the algorithm.
(8)                 This method should be redefined if using something other than GMState.
(8)                 If only subclassing GraphMatcher, a redefinition is not necessary.
(8)                 """
(8)                 self.core_1 = {}
(8)                 self.core_2 = {}
(8)                 self.inout_1 = {}
(8)                 self.inout_2 = {}
(8)                 self.state = GMState(self)
(8)                 self.mapping = self.core_1.copy()
(4)             def is_isomorphic(self):
(8)                 """Returns True if G1 and G2 are isomorphic graphs."""
(8)                 if self.G1.order() != self.G2.order():
(12)                    return False
(8)                 d1 = sorted(d for n, d in self.G1.degree())
(8)                 d2 = sorted(d for n, d in self.G2.degree())
(8)                 if d1 != d2:
(12)                    return False
(8)                 try:
(12)                    x = next(self.isomorphisms_iter())
(12)                    return True
(8)                 except StopIteration:
(12)                    return False
(4)             def isomorphisms_iter(self):
```

```
182: (8)                    """Generator over isomorphisms between G1 and G2."""
183: (8)                    self.test = "graph"
184: (8)                    self.initialize()
185: (8)                    yield from self.match()
186: (4)                def match(self):
187: (8)                    """Extends the isomorphism mapping.
188: (8)                    This function is called recursively to determine if a complete
189: (8)                    isomorphism can be found between G1 and G2.  It cleans up the class
190: (8)                    variables after each recursive call. If an isomorphism is found,
191: (8)                    we yield the mapping.
192: (8)                    """
193: (8)                    if len(self.core_1) == len(self.G2):
194: (12)                       self.mapping = self.core_1.copy()
195: (12)                       yield self.mapping
196: (8)                    else:
197: (12)                       for G1_node, G2_node in self.candidate_pairs_iter():
198: (16)                           if self.syntactic_feasibility(G1_node, G2_node):
199: (20)                               if self.semantic_feasibility(G1_node, G2_node):
200: (24)                                   newstate = self.state.__class__(self, G1_node,
G2_node)
201: (24)                                   yield from self.match()
202: (24)                                   newstate.restore()
203: (4)                def semantic_feasibility(self, G1_node, G2_node):
204: (8)                    """Returns True if adding (G1_node, G2_node) is semantically feasible.
205: (8)                    The semantic feasibility function should return True if it is
206: (8)                    acceptable to add the candidate pair (G1_node, G2_node) to the current
207: (8)                    partial isomorphism mapping.   The logic should focus on semantic
208: (8)                    information contained in the edge data or a formalized node class.
209: (8)                    By acceptable, we mean that the subsequent mapping can still become a
210: (8)                    complete isomorphism mapping.  Thus, if adding the candidate pair
211: (8)                    definitely makes it so that the subsequent mapping cannot become a
212: (8)                    complete isomorphism mapping, then this function must return False.
213: (8)                    The default semantic feasibility function always returns True. The
214: (8)                    effect is that semantics are not considered in the matching of G1
215: (8)                    and G2.
216: (8)                    The semantic checks might differ based on the what type of test is
217: (8)                    being performed.  A keyword description of the test is stored in
218: (8)                    self.test.  Here is a quick description of the currently implemented
219: (8)                    tests::
220: (10)                     test='graph'
221: (12)                       Indicates that the graph matcher is looking for a graph-graph
222: (12)                       isomorphism.
223: (10)                     test='subgraph'
224: (12)                       Indicates that the graph matcher is looking for a subgraph-graph
225: (12)                       isomorphism such that a subgraph of G1 is isomorphic to G2.
226: (10)                     test='mono'
227: (12)                       Indicates that the graph matcher is looking for a subgraph-graph
228: (12)                       monomorphism such that a subgraph of G1 is monomorphic to G2.
229: (8)                    Any subclass which redefines semantic_feasibility() must maintain
230: (8)                    the above form to keep the match() method functional. Implementations
231: (8)                    should consider multigraphs.
232: (8)                    """
233: (8)                    return True
234: (4)                def subgraph_is_isomorphic(self):
235: (8)                    """Returns True if a subgraph of G1 is isomorphic to G2."""
236: (8)                    try:
237: (12)                       x = next(self.subgraph_isomorphisms_iter())
238: (12)                       return True
239: (8)                    except StopIteration:
240: (12)                       return False
241: (4)                def subgraph_is_monomorphic(self):
242: (8)                    """Returns True if a subgraph of G1 is monomorphic to G2."""
243: (8)                    try:
244: (12)                       x = next(self.subgraph_monomorphisms_iter())
245: (12)                       return True
246: (8)                    except StopIteration:
247: (12)                       return False
248: (4)                def subgraph_isomorphisms_iter(self):
249: (8)                    """Generator over isomorphisms between a subgraph of G1 and G2."""
```

```
(8)                     self.test = "subgraph"
(8)                     self.initialize()
(8)                     yield from self.match()
(4)                 def subgraph_monomorphisms_iter(self):
(8)                     """Generator over monomorphisms between a subgraph of G1 and G2."""
(8)                     self.test = "mono"
(8)                     self.initialize()
(8)                     yield from self.match()
(4)                 def syntactic_feasibility(self, G1_node, G2_node):
(8)                     """Returns True if adding (G1_node, G2_node) is syntactically
feasible.
(8)                     This function returns True if it is adding the candidate pair
(8)                     to the current partial isomorphism/monomorphism mapping is allowable.
(8)                     The addition is allowable if the inclusion of the candidate pair does
(8)                     not make it impossible for an isomorphism/monomorphism to be found.
(8)                     """
(8)                     if self.test == "mono":
(12)                        if self.G1.number_of_edges(G1_node, G1_node) <
self.G2.number_of_edges(
(16)                            G2_node, G2_node
(12)                        ):
(16)                            return False
(8)                     else:
(12)                        if self.G1.number_of_edges(G1_node, G1_node) !=
self.G2.number_of_edges(
(16)                            G2_node, G2_node
(12)                        ):
(16)                            return False
(8)                     if self.test != "mono":
(12)                        for neighbor in self.G1[G1_node]:
(16)                            if neighbor in self.core_1:
(20)                                if self.core_1[neighbor] not in self.G2[G2_node]:
(24)                                    return False
(20)                                elif self.G1.number_of_edges(
(24)                                    neighbor, G1_node
(20)                                ) != self.G2.number_of_edges(self.core_1[neighbor],
G2_node):
(24)                                    return False
(8)                     for neighbor in self.G2[G2_node]:
(12)                        if neighbor in self.core_2:
(16)                            if self.core_2[neighbor] not in self.G1[G1_node]:
(20)                                return False
(16)                            elif self.test == "mono":
(20)                                if self.G1.number_of_edges(
(24)                                    self.core_2[neighbor], G1_node
(20)                                ) < self.G2.number_of_edges(neighbor, G2_node):
(24)                                    return False
(16)                            else:
(20)                                if self.G1.number_of_edges(
(24)                                    self.core_2[neighbor], G1_node
(20)                                ) != self.G2.number_of_edges(neighbor, G2_node):
(24)                                    return False
(8)                     if self.test != "mono":
(12)                        num1 = 0
(12)                        for neighbor in self.G1[G1_node]:
(16)                            if (neighbor in self.inout_1) and (neighbor not in
self.core_1):
(20)                                num1 += 1
(12)                        num2 = 0
(12)                        for neighbor in self.G2[G2_node]:
(16)                            if (neighbor in self.inout_2) and (neighbor not in
self.core_2):
(20)                                num2 += 1
(12)                        if self.test == "graph":
(16)                            if num1 != num2:
(20)                                return False
(12)                        else:  # self.test == 'subgraph'
(16)                            if not (num1 >= num2):
(20)                                return False
```

```
(12)                        num1 = 0
(12)                        for neighbor in self.G1[G1_node]:
(16)                            if neighbor not in self.inout_1:
(20)                                num1 += 1
(12)                        num2 = 0
(12)                        for neighbor in self.G2[G2_node]:
(16)                            if neighbor not in self.inout_2:
(20)                                num2 += 1
(12)                        if self.test == "graph":
(16)                            if num1 != num2:
(20)                                return False
(12)                        else:  # self.test == 'subgraph'
(16)                            if not (num1 >= num2):
(20)                                return False
(8)                     return True
(0)             class DiGraphMatcher(GraphMatcher):
(4)                 """Implementation of VF2 algorithm for matching directed graphs.
(4)                 Suitable for DiGraph and MultiDiGraph instances.
(4)                 """
(4)                 def __init__(self, G1, G2):
(8)                     """Initialize DiGraphMatcher.
(8)                     G1 and G2 should be nx.Graph or nx.MultiGraph instances.
(8)                     Examples
(8)                     --------
(8)                     To create a GraphMatcher which checks for syntactic feasibility:
(8)                     >>> from networkx.algorithms import isomorphism
(8)                     >>> G1 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
(8)                     >>> G2 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
(8)                     >>> DiGM = isomorphism.DiGraphMatcher(G1, G2)
(8)                     """
(8)                     super().__init__(G1, G2)
(4)                 def candidate_pairs_iter(self):
(8)                     """Iterator over candidate pairs of nodes in G1 and G2."""
(8)                     G1_nodes = self.G1_nodes
(8)                     G2_nodes = self.G2_nodes
(8)                     min_key = self.G2_node_order.__getitem__
(8)                     T1_out = [node for node in self.out_1 if node not in self.core_1]
(8)                     T2_out = [node for node in self.out_2 if node not in self.core_2]
(8)                     if T1_out and T2_out:
(12)                        node_2 = min(T2_out, key=min_key)
(12)                        for node_1 in T1_out:
(16)                            yield node_1, node_2
(8)                     else:  # as suggested by [1], correct
(12)                        T1_in = [node for node in self.in_1 if node not in self.core_1]
(12)                        T2_in = [node for node in self.in_2 if node not in self.core_2]
(12)                        if T1_in and T2_in:
(16)                            node_2 = min(T2_in, key=min_key)
(16)                            for node_1 in T1_in:
(20)                                yield node_1, node_2
(12)                        else:  # as inferred from [1], correct
(16)                            node_2 = min(G2_nodes - set(self.core_2), key=min_key)
(16)                            for node_1 in G1_nodes:
(20)                                if node_1 not in self.core_1:
(24)                                    yield node_1, node_2
(4)                 def initialize(self):
(8)                     """Reinitializes the state of the algorithm.
(8)                     This method should be redefined if using something other than
DiGMState.
(8)                     If only subclassing GraphMatcher, a redefinition is not necessary.
(8)                     """
(8)                     self.core_1 = {}
(8)                     self.core_2 = {}
(8)                     self.in_1 = {}
(8)                     self.in_2 = {}
(8)                     self.out_1 = {}
(8)                     self.out_2 = {}
(8)                     self.state = DiGMState(self)
(8)                     self.mapping = self.core_1.copy()
(4)                 def syntactic_feasibility(self, G1_node, G2_node):
```

```
(8)                     """Returns True if adding (G1_node, G2_node) is syntactically
feasible.
(8)                     This function returns True if it is adding the candidate pair
(8)                     to the current partial isomorphism/monomorphism mapping is allowable.
(8)                     The addition is allowable if the inclusion of the candidate pair does
(8)                     not make it impossible for an isomorphism/monomorphism to be found.
(8)                     """
(8)                     if self.test == "mono":
(12)                        if self.G1.number_of_edges(G1_node, G1_node) <
self.G2.number_of_edges(
(16)                            G2_node, G2_node
(12)                        ):
(16)                            return False
(8)                     else:
(12)                        if self.G1.number_of_edges(G1_node, G1_node) !=
self.G2.number_of_edges(
(16)                            G2_node, G2_node
(12)                        ):
(16)                            return False
(8)                     if self.test != "mono":
(12)                        for predecessor in self.G1.pred[G1_node]:
(16)                            if predecessor in self.core_1:
(20)                                if self.core_1[predecessor] not in self.G2.pred[G2_node]:
(24)                                    return False
(20)                                elif self.G1.number_of_edges(
(24)                                    predecessor, G1_node
(20)                                ) != self.G2.number_of_edges(self.core_1[predecessor],
G2_node):
(24)                                    return False
(8)                     for predecessor in self.G2.pred[G2_node]:
(12)                        if predecessor in self.core_2:
(16)                            if self.core_2[predecessor] not in self.G1.pred[G1_node]:
(20)                                return False
(16)                            elif self.test == "mono":
(20)                                if self.G1.number_of_edges(
(24)                                    self.core_2[predecessor], G1_node
(20)                                ) < self.G2.number_of_edges(predecessor, G2_node):
(24)                                    return False
(16)                            else:
(20)                                if self.G1.number_of_edges(
(24)                                    self.core_2[predecessor], G1_node
(20)                                ) != self.G2.number_of_edges(predecessor, G2_node):
(24)                                    return False
(8)                     if self.test != "mono":
(12)                        for successor in self.G1[G1_node]:
(16)                            if successor in self.core_1:
(20)                                if self.core_1[successor] not in self.G2[G2_node]:
(24)                                    return False
(20)                                elif self.G1.number_of_edges(
(24)                                    G1_node, successor
(20)                                ) != self.G2.number_of_edges(G2_node,
self.core_1[successor]):
(24)                                    return False
(8)                     for successor in self.G2[G2_node]:
(12)                        if successor in self.core_2:
(16)                            if self.core_2[successor] not in self.G1[G1_node]:
(20)                                return False
(16)                            elif self.test == "mono":
(20)                                if self.G1.number_of_edges(
(24)                                    G1_node, self.core_2[successor]
(20)                                ) < self.G2.number_of_edges(G2_node, successor):
(24)                                    return False
(16)                            else:
(20)                                if self.G1.number_of_edges(
(24)                                    G1_node, self.core_2[successor]
(20)                                ) != self.G2.number_of_edges(G2_node, successor):
(24)                                    return False
(8)                     if self.test != "mono":
(12)                        num1 = 0
```

```
445: (12)                        for predecessor in self.G1.pred[G1_node]:
446: (16)                            if (predecessor in self.in_1) and (predecessor not in
self.core_1):
447: (20)                                num1 += 1
448: (12)                        num2 = 0
449: (12)                        for predecessor in self.G2.pred[G2_node]:
450: (16)                            if (predecessor in self.in_2) and (predecessor not in
self.core_2):
451: (20)                                num2 += 1
452: (12)                        if self.test == "graph":
453: (16)                            if num1 != num2:
454: (20)                                return False
455: (12)                        else:  # self.test == 'subgraph'
456: (16)                            if not (num1 >= num2):
457: (20)                                return False
458: (12)                        num1 = 0
459: (12)                        for successor in self.G1[G1_node]:
460: (16)                            if (successor in self.in_1) and (successor not in
self.core_1):
461: (20)                                num1 += 1
462: (12)                        num2 = 0
463: (12)                        for successor in self.G2[G2_node]:
464: (16)                            if (successor in self.in_2) and (successor not in
self.core_2):
465: (20)                                num2 += 1
466: (12)                        if self.test == "graph":
467: (16)                            if num1 != num2:
468: (20)                                return False
469: (12)                        else:  # self.test == 'subgraph'
470: (16)                            if not (num1 >= num2):
471: (20)                                return False
472: (12)                        num1 = 0
473: (12)                        for predecessor in self.G1.pred[G1_node]:
474: (16)                            if (predecessor in self.out_1) and (predecessor not in
self.core_1):
475: (20)                                num1 += 1
476: (12)                        num2 = 0
477: (12)                        for predecessor in self.G2.pred[G2_node]:
478: (16)                            if (predecessor in self.out_2) and (predecessor not in
self.core_2):
479: (20)                                num2 += 1
480: (12)                        if self.test == "graph":
481: (16)                            if num1 != num2:
482: (20)                                return False
483: (12)                        else:  # self.test == 'subgraph'
484: (16)                            if not (num1 >= num2):
485: (20)                                return False
486: (12)                        num1 = 0
487: (12)                        for successor in self.G1[G1_node]:
488: (16)                            if (successor in self.out_1) and (successor not in
self.core_1):
489: (20)                                num1 += 1
490: (12)                        num2 = 0
491: (12)                        for successor in self.G2[G2_node]:
492: (16)                            if (successor in self.out_2) and (successor not in
self.core_2):
493: (20)                                num2 += 1
494: (12)                        if self.test == "graph":
495: (16)                            if num1 != num2:
496: (20)                                return False
497: (12)                        else:  # self.test == 'subgraph'
498: (16)                            if not (num1 >= num2):
499: (20)                                return False
500: (12)                        num1 = 0
501: (12)                        for predecessor in self.G1.pred[G1_node]:
502: (16)                            if (predecessor not in self.in_1) and (predecessor not in
self.out_1):
503: (20)                                num1 += 1
504: (12)                        num2 = 0
```

```
 505: (12)                        for predecessor in self.G2.pred[G2_node]:
 506: (16)                            if (predecessor not in self.in_2) and (predecessor not in
self.out_2):
 507: (20)                                num2 += 1
 508: (12)                        if self.test == "graph":
 509: (16)                            if num1 != num2:
 510: (20)                                return False
 511: (12)                        else:  # self.test == 'subgraph'
 512: (16)                            if not (num1 >= num2):
 513: (20)                                return False
 514: (12)                        num1 = 0
 515: (12)                        for successor in self.G1[G1_node]:
 516: (16)                            if (successor not in self.in_1) and (successor not in
self.out_1):
 517: (20)                                num1 += 1
 518: (12)                        num2 = 0
 519: (12)                        for successor in self.G2[G2_node]:
 520: (16)                            if (successor not in self.in_2) and (successor not in
self.out_2):
 521: (20)                                num2 += 1
 522: (12)                        if self.test == "graph":
 523: (16)                            if num1 != num2:
 524: (20)                                return False
 525: (12)                        else:  # self.test == 'subgraph'
 526: (16)                            if not (num1 >= num2):
 527: (20)                                return False
 528: (8)                    return True
 529: (0)            class GMState:
 530: (4)                """Internal representation of state for the GraphMatcher class.
 531: (4)                This class is used internally by the GraphMatcher class.  It is used
 532: (4)                only to store state specific data. There will be at most G2.order() of
 533: (4)                these objects in memory at a time, due to the depth-first search
 534: (4)                strategy employed by the VF2 algorithm.
 535: (4)                """
 536: (4)                def __init__(self, GM, G1_node=None, G2_node=None):
 537: (8)                    """Initializes GMState object.
 538: (8)                    Pass in the GraphMatcher to which this GMState belongs and the
 539: (8)                    new node pair that will be added to the GraphMatcher's current
 540: (8)                    isomorphism mapping.
 541: (8)                    """
 542: (8)                    self.GM = GM
 543: (8)                    self.G1_node = None
 544: (8)                    self.G2_node = None
 545: (8)                    self.depth = len(GM.core_1)
 546: (8)                    if G1_node is None or G2_node is None:
 547: (12)                       GM.core_1 = {}
 548: (12)                       GM.core_2 = {}
 549: (12)                       GM.inout_1 = {}
 550: (12)                       GM.inout_2 = {}
 551: (8)                    if G1_node is not None and G2_node is not None:
 552: (12)                       GM.core_1[G1_node] = G2_node
 553: (12)                       GM.core_2[G2_node] = G1_node
 554: (12)                       self.G1_node = G1_node
 555: (12)                       self.G2_node = G2_node
 556: (12)                       self.depth = len(GM.core_1)
 557: (12)                       if G1_node not in GM.inout_1:
 558: (16)                           GM.inout_1[G1_node] = self.depth
 559: (12)                       if G2_node not in GM.inout_2:
 560: (16)                           GM.inout_2[G2_node] = self.depth
 561: (12)                       new_nodes = set()
 562: (12)                       for node in GM.core_1:
 563: (16)                           new_nodes.update(
 564: (20)                               [neighbor for neighbor in GM.G1[node] if neighbor not in
GM.core_1]
 565: (16)                           )
 566: (12)                       for node in new_nodes:
 567: (16)                           if node not in GM.inout_1:
 568: (20)                               GM.inout_1[node] = self.depth
 569: (12)                       new_nodes = set()
```

```
(12)                        for node in GM.core_2:
(16)                            new_nodes.update(
(20)                                [neighbor for neighbor in GM.G2[node] if neighbor not in GM.core_2]
(16)                            )
(12)                        for node in new_nodes:
(16)                            if node not in GM.inout_2:
(20)                                GM.inout_2[node] = self.depth
(4)                 def restore(self):
(8)                     """Deletes the GMState object and restores the class variables."""
(8)                     if self.G1_node is not None and self.G2_node is not None:
(12)                        del self.GM.core_1[self.G1_node]
(12)                        del self.GM.core_2[self.G2_node]
(8)                     for vector in (self.GM.inout_1, self.GM.inout_2):
(12)                        for node in list(vector.keys()):
(16)                            if vector[node] == self.depth:
(20)                                del vector[node]
(0)             class DiGMState:
(4)                 """Internal representation of state for the DiGraphMatcher class.
(4)                 This class is used internally by the DiGraphMatcher class.  It is used
(4)                 only to store state specific data. There will be at most G2.order() of
(4)                 these objects in memory at a time, due to the depth-first search
(4)                 strategy employed by the VF2 algorithm.
(4)                 """
(4)                 def __init__(self, GM, G1_node=None, G2_node=None):
(8)                     """Initializes DiGMState object.
(8)                     Pass in the DiGraphMatcher to which this DiGMState belongs and the
(8)                     new node pair that will be added to the GraphMatcher's current
(8)                     isomorphism mapping.
(8)                     """
(8)                     self.GM = GM
(8)                     self.G1_node = None
(8)                     self.G2_node = None
(8)                     self.depth = len(GM.core_1)
(8)                     if G1_node is None or G2_node is None:
(12)                        GM.core_1 = {}
(12)                        GM.core_2 = {}
(12)                        GM.in_1 = {}
(12)                        GM.in_2 = {}
(12)                        GM.out_1 = {}
(12)                        GM.out_2 = {}
(8)                     if G1_node is not None and G2_node is not None:
(12)                        GM.core_1[G1_node] = G2_node
(12)                        GM.core_2[G2_node] = G1_node
(12)                        self.G1_node = G1_node
(12)                        self.G2_node = G2_node
(12)                        self.depth = len(GM.core_1)
(12)                        for vector in (GM.in_1, GM.out_1):
(16)                            if G1_node not in vector:
(20)                                vector[G1_node] = self.depth
(12)                        for vector in (GM.in_2, GM.out_2):
(16)                            if G2_node not in vector:
(20)                                vector[G2_node] = self.depth
(12)                        new_nodes = set()
(12)                        for node in GM.core_1:
(16)                            new_nodes.update(
(20)                                [
(24)                                    predecessor
(24)                                    for predecessor in GM.G1.predecessors(node)
(24)                                    if predecessor not in GM.core_1
(20)                                ]
(16)                            )
(12)                        for node in new_nodes:
(16)                            if node not in GM.in_1:
(20)                                GM.in_1[node] = self.depth
(12)                        new_nodes = set()
(12)                        for node in GM.core_2:
(16)                            new_nodes.update(
(20)                                [
```

```
638: (24)                                   predecessor
639: (24)                                   for predecessor in GM.G2.predecessors(node)
640: (24)                                   if predecessor not in GM.core_2
641: (20)                               ]
642: (16)                           )
643: (12)                       for node in new_nodes:
644: (16)                           if node not in GM.in_2:
645: (20)                               GM.in_2[node] = self.depth
646: (12)                       new_nodes = set()
647: (12)                       for node in GM.core_1:
648: (16)                           new_nodes.update(
649: (20)                               [
650: (24)                                   successor
651: (24)                                   for successor in GM.G1.successors(node)
652: (24)                                   if successor not in GM.core_1
653: (20)                               ]
654: (16)                           )
655: (12)                       for node in new_nodes:
656: (16)                           if node not in GM.out_1:
657: (20)                               GM.out_1[node] = self.depth
658: (12)                       new_nodes = set()
659: (12)                       for node in GM.core_2:
660: (16)                           new_nodes.update(
661: (20)                               [
662: (24)                                   successor
663: (24)                                   for successor in GM.G2.successors(node)
664: (24)                                   if successor not in GM.core_2
665: (20)                               ]
666: (16)                           )
667: (12)                       for node in new_nodes:
668: (16)                           if node not in GM.out_2:
669: (20)                               GM.out_2[node] = self.depth
670: (4)                def restore(self):
671: (8)                    """Deletes the DiGMState object and restores the class variables."""
672: (8)                    if self.G1_node is not None and self.G2_node is not None:
673: (12)                       del self.GM.core_1[self.G1_node]
674: (12)                       del self.GM.core_2[self.G2_node]
675: (8)                    for vector in (self.GM.in_1, self.GM.in_2, self.GM.out_1,
self.GM.out_2):
676: (12)                       for node in list(vector.keys()):
677: (16)                           if vector[node] == self.depth:
678: (20)                               del vector[node]
```

File 258 - matchhelpers.py:

```
1: (0)              """Functions which help end users define customize node_match and
2: (0)              edge_match functions to use during isomorphism checks.
3: (0)              """
4: (0)              import math
5: (0)              import types
6: (0)              from itertools import permutations
7: (0)              __all__ = [
8: (4)                  "categorical_node_match",
9: (4)                  "categorical_edge_match",
10: (4)                 "categorical_multiedge_match",
11: (4)                 "numerical_node_match",
12: (4)                 "numerical_edge_match",
13: (4)                 "numerical_multiedge_match",
14: (4)                 "generic_node_match",
15: (4)                 "generic_edge_match",
16: (4)                 "generic_multiedge_match",
17: (0)             ]
18: (0)             def copyfunc(f, name=None):
19: (4)                 """Returns a deepcopy of a function."""
20: (4)                 return types.FunctionType(
21: (8)                     f.__code__, f.__globals__, name or f.__name__, f.__defaults__,
f.__closure__
```

```
 22: (4)                 )
 23: (0)             def allclose(x, y, rtol=1.0000000000000001e-05, atol=1e-08):
 24: (4)                 """Returns True if x and y are sufficiently close, elementwise.
 25: (4)                 Parameters
 26: (4)                 ----------
 27: (4)                 rtol : float
 28: (8)                     The relative error tolerance.
 29: (4)                 atol : float
 30: (8)                     The absolute error tolerance.
 31: (4)                 """
 32: (4)                 return all(math.isclose(xi, yi, rel_tol=rtol, abs_tol=atol) for xi, yi in
zip(x, y))
 33: (0)             categorical_doc = """
 34: (0)             Returns a comparison function for a categorical node attribute.
 35: (0)             The value(s) of the attr(s) must be hashable and comparable via the ==
 36: (0)             operator since they are placed into a set([]) object.  If the sets from
 37: (0)             G1 and G2 are the same, then the constructed function returns True.
 38: (0)             Parameters
 39: (0)             ----------
 40: (0)             attr : string | list
 41: (4)                 The categorical node attribute to compare, or a list of categorical
 42: (4)                 node attributes to compare.
 43: (0)             default : value | list
 44: (4)                 The default value for the categorical node attribute, or a list of
 45: (4)                 default values for the categorical node attributes.
 46: (0)             Returns
 47: (0)             -------
 48: (0)             match : function
 49: (4)                 The customized, categorical `node_match` function.
 50: (0)             Examples
 51: (0)             --------
 52: (0)             >>> import networkx.algorithms.isomorphism as iso
 53: (0)             >>> nm = iso.categorical_node_match("size", 1)
 54: (0)             >>> nm = iso.categorical_node_match(["color", "size"], ["red", 2])
 55: (0)             """
 56: (0)             def categorical_node_match(attr, default):
 57: (4)                 if isinstance(attr, str):
 58: (8)                     def match(data1, data2):
 59: (12)                        return data1.get(attr, default) == data2.get(attr, default)
 60: (4)                 else:
 61: (8)                     attrs = list(zip(attr, default))  # Python 3
 62: (8)                     def match(data1, data2):
 63: (12)                        return all(data1.get(attr, d) == data2.get(attr, d) for attr, d in
attrs)
 64: (4)                 return match
 65: (0)             categorical_edge_match = copyfunc(categorical_node_match,
"categorical_edge_match")
 66: (0)             def categorical_multiedge_match(attr, default):
 67: (4)                 if isinstance(attr, str):
 68: (8)                     def match(datasets1, datasets2):
 69: (12)                        values1 = {data.get(attr, default) for data in datasets1.values()}
 70: (12)                        values2 = {data.get(attr, default) for data in datasets2.values()}
 71: (12)                        return values1 == values2
 72: (4)                 else:
 73: (8)                     attrs = list(zip(attr, default))  # Python 3
 74: (8)                     def match(datasets1, datasets2):
 75: (12)                        values1 = set()
 76: (12)                        for data1 in datasets1.values():
 77: (16)                            x = tuple(data1.get(attr, d) for attr, d in attrs)
 78: (16)                            values1.add(x)
 79: (12)                        values2 = set()
 80: (12)                        for data2 in datasets2.values():
 81: (16)                            x = tuple(data2.get(attr, d) for attr, d in attrs)
 82: (16)                            values2.add(x)
 83: (12)                        return values1 == values2
 84: (4)                 return match
 85: (0)             categorical_node_match.__doc__ = categorical_doc
 86: (0)             categorical_edge_match.__doc__ = categorical_doc.replace("node", "edge")
 87: (0)             tmpdoc = categorical_doc.replace("node", "edge")
```

```
88: (0)             tmpdoc = tmpdoc.replace("categorical_edge_match",
"categorical_multiedge_match")
89: (0)             categorical_multiedge_match.__doc__ = tmpdoc
90: (0)             numerical_doc = """
91: (0)             Returns a comparison function for a numerical node attribute.
92: (0)             The value(s) of the attr(s) must be numerical and sortable.  If the
93: (0)             sorted list of values from G1 and G2 are the same within some
94: (0)             tolerance, then the constructed function returns True.
95: (0)             Parameters
96: (0)             ----------
97: (0)             attr : string | list
98: (4)                 The numerical node attribute to compare, or a list of numerical
99: (4)                 node attributes to compare.
100: (0)            default : value | list
101: (4)                The default value for the numerical node attribute, or a list of
102: (4)                default values for the numerical node attributes.
103: (0)            rtol : float
104: (4)                The relative error tolerance.
105: (0)            atol : float
106: (4)                The absolute error tolerance.
107: (0)            Returns
108: (0)            -------
109: (0)            match : function
110: (4)                The customized, numerical `node_match` function.
111: (0)            Examples
112: (0)            --------
113: (0)            >>> import networkx.algorithms.isomorphism as iso
114: (0)            >>> nm = iso.numerical_node_match("weight", 1.0)
115: (0)            >>> nm = iso.numerical_node_match(["weight", "linewidth"], [0.25, 0.5])
116: (0)            """
117: (0)            def numerical_node_match(attr, default, rtol=1.0000000000000001e-05, atol=1e-
08):
118: (4)                if isinstance(attr, str):
119: (8)                    def match(data1, data2):
120: (12)                       return math.isclose(
121: (16)                           data1.get(attr, default),
122: (16)                           data2.get(attr, default),
123: (16)                           rel_tol=rtol,
124: (16)                           abs_tol=atol,
125: (12)                       )
126: (4)                else:
127: (8)                    attrs = list(zip(attr, default))  # Python 3
128: (8)                    def match(data1, data2):
129: (12)                       values1 = [data1.get(attr, d) for attr, d in attrs]
130: (12)                       values2 = [data2.get(attr, d) for attr, d in attrs]
131: (12)                       return allclose(values1, values2, rtol=rtol, atol=atol)
132: (4)                return match
133: (0)            numerical_edge_match = copyfunc(numerical_node_match, "numerical_edge_match")
134: (0)            def numerical_multiedge_match(attr, default, rtol=1.0000000000000001e-05,
atol=1e-08):
135: (4)                if isinstance(attr, str):
136: (8)                    def match(datasets1, datasets2):
137: (12)                       values1 = sorted(data.get(attr, default) for data in
datasets1.values())
138: (12)                       values2 = sorted(data.get(attr, default) for data in
datasets2.values())
139: (12)                       return allclose(values1, values2, rtol=rtol, atol=atol)
140: (4)                else:
141: (8)                    attrs = list(zip(attr, default))  # Python 3
142: (8)                    def match(datasets1, datasets2):
143: (12)                       values1 = []
144: (12)                       for data1 in datasets1.values():
145: (16)                           x = tuple(data1.get(attr, d) for attr, d in attrs)
146: (16)                           values1.append(x)
147: (12)                       values2 = []
148: (12)                       for data2 in datasets2.values():
149: (16)                           x = tuple(data2.get(attr, d) for attr, d in attrs)
150: (16)                           values2.append(x)
151: (12)                       values1.sort()
```

```
(12)                        values2.sort()
(12)                        for xi, yi in zip(values1, values2):
(16)                            if not allclose(xi, yi, rtol=rtol, atol=atol):
(20)                                return False
(12)                        else:
(16)                            return True
(4)                 return match
(0)             numerical_node_match.__doc__ = numerical_doc
(0)             numerical_edge_match.__doc__ = numerical_doc.replace("node", "edge")
(0)             tmpdoc = numerical_doc.replace("node", "edge")
(0)             tmpdoc = tmpdoc.replace("numerical_edge_match", "numerical_multiedge_match")
(0)             numerical_multiedge_match.__doc__ = tmpdoc
(0)             generic_doc = """
(0)             Returns a comparison function for a generic attribute.
(0)             The value(s) of the attr(s) are compared using the specified
(0)             operators. If all the attributes are equal, then the constructed
(0)             function returns True.
(0)             Parameters
(0)             ----------
(0)             attr : string | list
(4)                 The node attribute to compare, or a list of node attributes
(4)                 to compare.
(0)             default : value | list
(4)                 The default value for the node attribute, or a list of
(4)                 default values for the node attributes.
(0)             op : callable | list
(4)                 The operator to use when comparing attribute values, or a list
(4)                 of operators to use when comparing values for each attribute.
(0)             Returns
(0)             -------
(0)             match : function
(4)                 The customized, generic `node_match` function.
(0)             Examples
(0)             --------
(0)             >>> from operator import eq
(0)             >>> from math import isclose
(0)             >>> from networkx.algorithms.isomorphism import generic_node_match
(0)             >>> nm = generic_node_match("weight", 1.0, isclose)
(0)             >>> nm = generic_node_match("color", "red", eq)
(0)             >>> nm = generic_node_match(["weight", "color"], [1.0, "red"], [isclose, eq])
(0)             """
(0)             def generic_node_match(attr, default, op):
(4)                 if isinstance(attr, str):
(8)                     def match(data1, data2):
(12)                        return op(data1.get(attr, default), data2.get(attr, default))
(4)                 else:
(8)                     attrs = list(zip(attr, default, op))  # Python 3
(8)                     def match(data1, data2):
(12)                        for attr, d, operator in attrs:
(16)                            if not operator(data1.get(attr, d), data2.get(attr, d)):
(20)                                return False
(12)                        else:
(16)                            return True
(4)                 return match
(0)             generic_edge_match = copyfunc(generic_node_match, "generic_edge_match")
(0)             def generic_multiedge_match(attr, default, op):
(4)                 """Returns a comparison function for a generic attribute.
(4)                 The value(s) of the attr(s) are compared using the specified
(4)                 operators. If all the attributes are equal, then the constructed
(4)                 function returns True. Potentially, the constructed edge_match
(4)                 function can be slow since it must verify that no isomorphism
(4)                 exists between the multiedges before it returns False.
(4)                 Parameters
(4)                 ----------
(4)                 attr : string | list
(8)                     The edge attribute to compare, or a list of node attributes
(8)                     to compare.
(4)                 default : value | list
(8)                     The default value for the edge attribute, or a list of
```

```
221: (8)                    default values for the edgeattributes.
222: (4)                op : callable | list
223: (8)                    The operator to use when comparing attribute values, or a list
224: (8)                    of operators to use when comparing values for each attribute.
225: (4)                Returns
226: (4)                -------
227: (4)                match : function
228: (8)                    The customized, generic `edge_match` function.
229: (4)                Examples
230: (4)                --------
231: (4)                >>> from operator import eq
232: (4)                >>> from math import isclose
233: (4)                >>> from networkx.algorithms.isomorphism import generic_node_match
234: (4)                >>> nm = generic_node_match("weight", 1.0, isclose)
235: (4)                >>> nm = generic_node_match("color", "red", eq)
236: (4)                >>> nm = generic_node_match(["weight", "color"], [1.0, "red"], [isclose,
eq])
237: (4)                """
238: (4)                if isinstance(attr, str):
239: (8)                    attr = [attr]
240: (8)                    default = [default]
241: (8)                    op = [op]
242: (4)                attrs = list(zip(attr, default))  # Python 3
243: (4)                def match(datasets1, datasets2):
244: (8)                    values1 = []
245: (8)                    for data1 in datasets1.values():
246: (12)                       x = tuple(data1.get(attr, d) for attr, d in attrs)
247: (12)                       values1.append(x)
248: (8)                    values2 = []
249: (8)                    for data2 in datasets2.values():
250: (12)                       x = tuple(data2.get(attr, d) for attr, d in attrs)
251: (12)                       values2.append(x)
252: (8)                    for vals2 in permutations(values2):
253: (12)                       for xi, yi in zip(values1, vals2):
254: (16)                           if not all(map(lambda x, y, z: z(x, y), xi, yi, op)):
255: (20)                               break
256: (12)                       else:
257: (16)                           return True
258: (8)                    else:
259: (12)                       return False
260: (4)                return match
261: (0)            generic_node_match.__doc__ = generic_doc
262: (0)            generic_edge_match.__doc__ = generic_doc.replace("node", "edge")
```

File 259 - temporalisomorphvf2.py:

```
1: (0)              """
2: (0)              ****************************
3: (0)              Time-respecting VF2 Algorithm
4: (0)              ****************************
5: (0)              An extension of the VF2 algorithm for time-respecting graph isomorphism
6: (0)              testing in temporal graphs.
7: (0)              A temporal graph is one in which edges contain a datetime attribute,
8: (0)              denoting when interaction occurred between the incident nodes. A
9: (0)              time-respecting subgraph of a temporal graph is a subgraph such that
10: (0)             all interactions incident to a node occurred within a time threshold,
11: (0)             delta, of each other. A directed time-respecting subgraph has the
12: (0)             added constraint that incoming interactions to a node must precede
13: (0)             outgoing interactions from the same node - this enforces a sense of
14: (0)             directed flow.
15: (0)             Introduction
16: (0)             ------------
17: (0)             The TimeRespectingGraphMatcher and TimeRespectingDiGraphMatcher
18: (0)             extend the GraphMatcher and DiGraphMatcher classes, respectively,
19: (0)             to include temporal constraints on matches. This is achieved through
20: (0)             a semantic check, via the semantic_feasibility() function.
21: (0)             As well as including G1 (the graph in which to seek embeddings) and
```

```
22: (0)             G2 (the subgraph structure of interest), the name of the temporal
23: (0)             attribute on the edges and the time threshold, delta, must be supplied
24: (0)             as arguments to the matching constructors.
25: (0)             A delta of zero is the strictest temporal constraint on the match -
26: (0)             only embeddings in which all interactions occur at the same time will
27: (0)             be returned. A delta of one day will allow embeddings in which
28: (0)             adjacent interactions occur up to a day apart.
29: (0)             Examples
30: (0)             --------
31: (0)             Examples will be provided when the datetime type has been incorporated.
32: (0)             Temporal Subgraph Isomorphism
33: (0)             ----------------------------
34: (0)             A brief discussion of the somewhat diverse current literature will be
35: (0)             included here.
36: (0)             References
37: (0)             ----------
38: (0)             [1] Redmond, U. and Cunningham, P. Temporal subgraph isomorphism. In:
39: (0)             The 2013 IEEE/ACM International Conference on Advances in Social
40: (0)             Networks Analysis and Mining (ASONAM). Niagara Falls, Canada; 2013:
41: (0)             pages 1451 - 1452. [65]
42: (0)             For a discussion of the literature on temporal networks:
43: (0)             [3] P. Holme and J. Saramaki. Temporal networks. Physics Reports,
44: (0)             519(3):97–125, 2012.
45: (0)             Notes
46: (0)             -----
47: (0)             Handles directed and undirected graphs and graphs with parallel edges.
48: (0)             """
49: (0)             import networkx as nx
50: (0)             from .isomorphvf2 import DiGraphMatcher, GraphMatcher
51: (0)             __all__ = ["TimeRespectingGraphMatcher", "TimeRespectingDiGraphMatcher"]
52: (0)             class TimeRespectingGraphMatcher(GraphMatcher):
53: (4)                 def __init__(self, G1, G2, temporal_attribute_name, delta):
54: (8)                     """Initialize TimeRespectingGraphMatcher.
55: (8)                     G1 and G2 should be nx.Graph or nx.MultiGraph instances.
56: (8)                     Examples
57: (8)                     --------
58: (8)                     To create a TimeRespectingGraphMatcher which checks for
59: (8)                     syntactic and semantic feasibility:
60: (8)                     >>> from networkx.algorithms import isomorphism
61: (8)                     >>> from datetime import timedelta
62: (8)                     >>> G1 = nx.Graph(nx.path_graph(4, create_using=nx.Graph()))
63: (8)                     >>> G2 = nx.Graph(nx.path_graph(4, create_using=nx.Graph()))
64: (8)                     >>> GM = isomorphism.TimeRespectingGraphMatcher(G1, G2, "date",
timedelta(days=1))
65: (8)                     """
66: (8)                     self.temporal_attribute_name = temporal_attribute_name
67: (8)                     self.delta = delta
68: (8)                     super().__init__(G1, G2)
69: (4)                 def one_hop(self, Gx, Gx_node, neighbors):
70: (8)                     """
71: (8)                     Edges one hop out from a node in the mapping should be
72: (8)                     time-respecting with respect to each other.
73: (8)                     """
74: (8)                     dates = []
75: (8)                     for n in neighbors:
76: (12)                        if isinstance(Gx, nx.Graph):  # Graph G[u][v] returns the data
dictionary.
77: (16)                            dates.append(Gx[Gx_node][n][self.temporal_attribute_name])
78: (12)                        else:  # MultiGraph G[u][v] returns a dictionary of key -> data
dictionary.
79: (16)                            for edge in Gx[Gx_node][
80: (20)                                n
81: (16)                            ].values():  # Iterates all edges between node pair.
82: (20)                                dates.append(edge[self.temporal_attribute_name])
83: (8)                     if any(x is None for x in dates):
84: (12)                        raise ValueError("Datetime not supplied for at least one edge.")
85: (8)                     return not dates or max(dates) - min(dates) <= self.delta
86: (4)                 def two_hop(self, Gx, core_x, Gx_node, neighbors):
87: (8)                     """
```

```
88: (8)                     Paths of length 2 from Gx_node should be time-respecting.
89: (8)                     """
90: (8)                     return all(
91: (12)                        self.one_hop(Gx, v, [n for n in Gx[v] if n in core_x] + [Gx_node])
92: (12)                        for v in neighbors
93: (8)                     )
94: (4)                 def semantic_feasibility(self, G1_node, G2_node):
95: (8)                     """Returns True if adding (G1_node, G2_node) is semantically
96: (8)                     feasible.
97: (8)                     Any subclass which redefines semantic_feasibility() must
98: (8)                     maintain the self.tests if needed, to keep the match() method
99: (8)                     functional. Implementations should consider multigraphs.
100: (8)                    """
101: (8)                    neighbors = [n for n in self.G1[G1_node] if n in self.core_1]
102: (8)                    if not self.one_hop(self.G1, G1_node, neighbors):  # Fail fast on
first node.
103: (12)                       return False
104: (8)                    if not self.two_hop(self.G1, self.core_1, G1_node, neighbors):
105: (12)                       return False
106: (8)                    return True
107: (0)            class TimeRespectingDiGraphMatcher(DiGraphMatcher):
108: (4)                def __init__(self, G1, G2, temporal_attribute_name, delta):
109: (8)                    """Initialize TimeRespectingDiGraphMatcher.
110: (8)                    G1 and G2 should be nx.DiGraph or nx.MultiDiGraph instances.
111: (8)                    Examples
112: (8)                    --------
113: (8)                    To create a TimeRespectingDiGraphMatcher which checks for
114: (8)                    syntactic and semantic feasibility:
115: (8)                    >>> from networkx.algorithms import isomorphism
116: (8)                    >>> from datetime import timedelta
117: (8)                    >>> G1 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
118: (8)                    >>> G2 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
119: (8)                    >>> GM = isomorphism.TimeRespectingDiGraphMatcher(G1, G2, "date",
timedelta(days=1))
120: (8)                    """
121: (8)                    self.temporal_attribute_name = temporal_attribute_name
122: (8)                    self.delta = delta
123: (8)                    super().__init__(G1, G2)
124: (4)                def get_pred_dates(self, Gx, Gx_node, core_x, pred):
125: (8)                    """
126: (8)                    Get the dates of edges from predecessors.
127: (8)                    """
128: (8)                    pred_dates = []
129: (8)                    if isinstance(Gx, nx.DiGraph):  # Graph G[u][v] returns the data
dictionary.
130: (12)                       for n in pred:
131: (16)                           pred_dates.append(Gx[n][Gx_node]
[self.temporal_attribute_name])
132: (8)                    else:  # MultiGraph G[u][v] returns a dictionary of key -> data
dictionary.
133: (12)                       for n in pred:
134: (16)                           for edge in Gx[n][
135: (20)                               Gx_node
136: (16)                           ].values():  # Iterates all edge data between node pair.
137: (20)                               pred_dates.append(edge[self.temporal_attribute_name])
138: (8)                    return pred_dates
139: (4)                def get_succ_dates(self, Gx, Gx_node, core_x, succ):
140: (8)                    """
141: (8)                    Get the dates of edges to successors.
142: (8)                    """
143: (8)                    succ_dates = []
144: (8)                    if isinstance(Gx, nx.DiGraph):  # Graph G[u][v] returns the data
dictionary.
145: (12)                       for n in succ:
146: (16)                           succ_dates.append(Gx[Gx_node][n]
[self.temporal_attribute_name])
147: (8)                    else:  # MultiGraph G[u][v] returns a dictionary of key -> data
dictionary.
148: (12)                       for n in succ:
```

```
149: (16)                            for edge in Gx[Gx_node][
150: (20)                                n
151: (16)                            ].values():  # Iterates all edge data between node pair.
152: (20)                                succ_dates.append(edge[self.temporal_attribute_name])
153: (8)                    return succ_dates
154: (4)                def one_hop(self, Gx, Gx_node, core_x, pred, succ):
155: (8)                    """
156: (8)                    The ego node.
157: (8)                    """
158: (8)                    pred_dates = self.get_pred_dates(Gx, Gx_node, core_x, pred)
159: (8)                    succ_dates = self.get_succ_dates(Gx, Gx_node, core_x, succ)
160: (8)                    return self.test_one(pred_dates, succ_dates) and self.test_two(
161: (12)                        pred_dates, succ_dates
162: (8)                    )
163: (4)                def two_hop_pred(self, Gx, Gx_node, core_x, pred):
164: (8)                    """
165: (8)                    The predecessors of the ego node.
166: (8)                    """
167: (8)                    return all(
168: (12)                        self.one_hop(
169: (16)                            Gx,
170: (16)                            p,
171: (16)                            core_x,
172: (16)                            self.preds(Gx, core_x, p),
173: (16)                            self.succs(Gx, core_x, p, Gx_node),
174: (12)                        )
175: (12)                        for p in pred
176: (8)                    )
177: (4)                def two_hop_succ(self, Gx, Gx_node, core_x, succ):
178: (8)                    """
179: (8)                    The successors of the ego node.
180: (8)                    """
181: (8)                    return all(
182: (12)                        self.one_hop(
183: (16)                            Gx,
184: (16)                            s,
185: (16)                            core_x,
186: (16)                            self.preds(Gx, core_x, s, Gx_node),
187: (16)                            self.succs(Gx, core_x, s),
188: (12)                        )
189: (12)                        for s in succ
190: (8)                    )
191: (4)                def preds(self, Gx, core_x, v, Gx_node=None):
192: (8)                    pred = [n for n in Gx.predecessors(v) if n in core_x]
193: (8)                    if Gx_node:
194: (12)                        pred.append(Gx_node)
195: (8)                    return pred
196: (4)                def succs(self, Gx, core_x, v, Gx_node=None):
197: (8)                    succ = [n for n in Gx.successors(v) if n in core_x]
198: (8)                    if Gx_node:
199: (12)                        succ.append(Gx_node)
200: (8)                    return succ
201: (4)                def test_one(self, pred_dates, succ_dates):
202: (8)                    """
203: (8)                    Edges one hop out from Gx_node in the mapping should be
204: (8)                    time-respecting with respect to each other, regardless of
205: (8)                    direction.
206: (8)                    """
207: (8)                    time_respecting = True
208: (8)                    dates = pred_dates + succ_dates
209: (8)                    if any(x is None for x in dates):
210: (12)                        raise ValueError("Date or datetime not supplied for at least one
edge.")
211: (8)                    dates.sort()  # Small to large.
212: (8)                    if 0 < len(dates) and not (dates[-1] - dates[0] <= self.delta):
213: (12)                        time_respecting = False
214: (8)                    return time_respecting
215: (4)                def test_two(self, pred_dates, succ_dates):
216: (8)                    """
```

```
217: (8)                    Edges from a dual Gx_node in the mapping should be ordered in
218: (8)                    a time-respecting manner.
219: (8)                    """
220: (8)                    time_respecting = True
221: (8)                    pred_dates.sort()
222: (8)                    succ_dates.sort()
223: (8)                    if (
224: (12)                       0 < len(succ_dates)
225: (12)                       and 0 < len(pred_dates)
226: (12)                       and succ_dates[0] < pred_dates[-1]
227: (8)                    ):
228: (12)                       time_respecting = False
229: (8)                    return time_respecting
230: (4)                def semantic_feasibility(self, G1_node, G2_node):
231: (8)                    """Returns True if adding (G1_node, G2_node) is semantically
232: (8)                    feasible.
233: (8)                    Any subclass which redefines semantic_feasibility() must
234: (8)                    maintain the self.tests if needed, to keep the match() method
235: (8)                    functional. Implementations should consider multigraphs.
236: (8)                    """
237: (8)                    pred, succ = (
238: (12)                       [n for n in self.G1.predecessors(G1_node) if n in self.core_1],
239: (12)                       [n for n in self.G1.successors(G1_node) if n in self.core_1],
240: (8)                    )
241: (8)                    if not self.one_hop(
242: (12)                       self.G1, G1_node, self.core_1, pred, succ
243: (8)                    ):  # Fail fast on first node.
244: (12)                       return False
245: (8)                    if not self.two_hop_pred(self.G1, G1_node, self.core_1, pred):
246: (12)                       return False
247: (8)                    if not self.two_hop_succ(self.G1, G1_node, self.core_1, succ):
248: (12)                       return False
249: (8)                    return True
```

File 260 - vf2pp.py:

```
1: (0)              """
2: (0)              ***************
3: (0)              VF2++ Algorithm
4: (0)              ***************
5: (0)              An implementation of the VF2++ algorithm [1]_ for Graph Isomorphism testing.
6: (0)              The simplest interface to use this module is to call:
7: (0)              `vf2pp_is_isomorphic`: to check whether two graphs are isomorphic.
8: (0)              `vf2pp_isomorphism`: to obtain the node mapping between two graphs,
9: (0)              in case they are isomorphic.
10: (0)             `vf2pp_all_isomorphisms`: to generate all possible mappings between two
graphs,
11: (0)             if isomorphic.
12: (0)             Introduction
13: (0)             ------------
14: (0)             The VF2++ algorithm, follows a similar logic to that of VF2, while also
15: (0)             introducing new easy-to-check cutting rules and determining the optimal access
16: (0)             order of nodes. It is also implemented in a non-recursive manner, which saves
17: (0)             both time and space, when compared to its previous counterpart.
18: (0)             The optimal node ordering is obtained after taking into consideration both the
19: (0)             degree but also the label rarity of each node.
20: (0)             This way we place the nodes that are more likely to match, first in the order,
21: (0)             thus examining the most promising branches in the beginning.
22: (0)             The rules also consider node labels, making it easier to prune unfruitful
23: (0)             branches early in the process.
24: (0)             Examples
25: (0)             --------
26: (0)             Suppose G1 and G2 are Isomorphic Graphs. Verification is as follows:
27: (0)             Without node labels:
28: (0)             >>> import networkx as nx
29: (0)             >>> G1 = nx.path_graph(4)
30: (0)             >>> G2 = nx.path_graph(4)
```

```
31: (0)             >>> nx.vf2pp_is_isomorphic(G1, G2, node_label=None)
32: (0)             True
33: (0)             >>> nx.vf2pp_isomorphism(G1, G2, node_label=None)
34: (0)             {1: 1, 2: 2, 0: 0, 3: 3}
35: (0)             With node labels:
36: (0)             >>> G1 = nx.path_graph(4)
37: (0)             >>> G2 = nx.path_graph(4)
38: (0)             >>> mapped = {1: 1, 2: 2, 3: 3, 0: 0}
39: (0)             >>> nx.set_node_attributes(G1, dict(zip(G1, ["blue", "red", "green",
"yellow"])), "label")
40: (0)             >>> nx.set_node_attributes(G2, dict(zip([mapped[u] for u in G1], ["blue",
"red", "green", "yellow"])), "label")
41: (0)             >>> nx.vf2pp_is_isomorphic(G1, G2, node_label="label")
42: (0)             True
43: (0)             >>> nx.vf2pp_isomorphism(G1, G2, node_label="label")
44: (0)             {1: 1, 2: 2, 0: 0, 3: 3}
45: (0)             References
46: (0)             ----------
47: (0)             .. [1] Jüttner, Alpár & Madarasi, Péter. (2018). "VF2++—An improved subgraph
48: (3)                isomorphism algorithm". Discrete Applied Mathematics. 242.
49: (3)                https://doi.org/10.1016/j.dam.2018.02.018
50: (0)             """
51: (0)             import collections
52: (0)             import networkx as nx
53: (0)             __all__ = ["vf2pp_isomorphism", "vf2pp_is_isomorphic",
"vf2pp_all_isomorphisms"]
54: (0)             _GraphParameters = collections.namedtuple(
55: (4)                 "_GraphParameters",
56: (4)                 [
57: (8)                     "G1",
58: (8)                     "G2",
59: (8)                     "G1_labels",
60: (8)                     "G2_labels",
61: (8)                     "nodes_of_G1Labels",
62: (8)                     "nodes_of_G2Labels",
63: (8)                     "G2_nodes_of_degree",
64: (4)                 ],
65: (0)             )
66: (0)             _StateParameters = collections.namedtuple(
67: (4)                 "_StateParameters",
68: (4)                 [
69: (8)                     "mapping",
70: (8)                     "reverse_mapping",
71: (8)                     "T1",
72: (8)                     "T1_in",
73: (8)                     "T1_tilde",
74: (8)                     "T1_tilde_in",
75: (8)                     "T2",
76: (8)                     "T2_in",
77: (8)                     "T2_tilde",
78: (8)                     "T2_tilde_in",
79: (4)                 ],
80: (0)             )
81: (0)             @nx._dispatchable(graphs={"G1": 0, "G2": 1}, node_attrs={"node_label":
"default_label"})
82: (0)             def vf2pp_isomorphism(G1, G2, node_label=None, default_label=None):
83: (4)                 """Return an isomorphic mapping between `G1` and `G2` if it exists.
84: (4)                 Parameters
85: (4)                 ----------
86: (4)                 G1, G2 : NetworkX Graph or MultiGraph instances.
87: (8)                     The two graphs to check for isomorphism.
88: (4)                 node_label : str, optional
89: (8)                     The name of the node attribute to be used when comparing nodes.
90: (8)                     The default is `None`, meaning node attributes are not considered
91: (8)                     in the comparison. Any node that doesn't have the `node_label`
92: (8)                     attribute uses `default_label` instead.
93: (4)                 default_label : scalar
94: (8)                     Default value to use when a node doesn't have an attribute
95: (8)                     named `node_label`. Default is `None`.
```

```
96: (4)                 Returns
97: (4)                 -------
98: (4)                 dict or None
99: (8)                     Node mapping if the two graphs are isomorphic. None otherwise.
100: (4)                """
101: (4)                try:
102: (8)                    mapping = next(vf2pp_all_isomorphisms(G1, G2, node_label,
default_label))
103: (8)                    return mapping
104: (4)                except StopIteration:
105: (8)                    return None
106: (0)            @nx._dispatchable(graphs={"G1": 0, "G2": 1}, node_attrs={"node_label":
"default_label"})
107: (0)            def vf2pp_is_isomorphic(G1, G2, node_label=None, default_label=None):
108: (4)                """Examines whether G1 and G2 are isomorphic.
109: (4)                Parameters
110: (4)                ----------
111: (4)                G1, G2 : NetworkX Graph or MultiGraph instances.
112: (8)                    The two graphs to check for isomorphism.
113: (4)                node_label : str, optional
114: (8)                    The name of the node attribute to be used when comparing nodes.
115: (8)                    The default is `None`, meaning node attributes are not considered
116: (8)                    in the comparison. Any node that doesn't have the `node_label`
117: (8)                    attribute uses `default_label` instead.
118: (4)                default_label : scalar
119: (8)                    Default value to use when a node doesn't have an attribute
120: (8)                    named `node_label`. Default is `None`.
121: (4)                Returns
122: (4)                -------
123: (4)                bool
124: (8)                    True if the two graphs are isomorphic, False otherwise.
125: (4)                """
126: (4)                if vf2pp_isomorphism(G1, G2, node_label, default_label) is not None:
127: (8)                    return True
128: (4)                return False
129: (0)            @nx._dispatchable(graphs={"G1": 0, "G2": 1}, node_attrs={"node_label":
"default_label"})
130: (0)            def vf2pp_all_isomorphisms(G1, G2, node_label=None, default_label=None):
131: (4)                """Yields all the possible mappings between G1 and G2.
132: (4)                Parameters
133: (4)                ----------
134: (4)                G1, G2 : NetworkX Graph or MultiGraph instances.
135: (8)                    The two graphs to check for isomorphism.
136: (4)                node_label : str, optional
137: (8)                    The name of the node attribute to be used when comparing nodes.
138: (8)                    The default is `None`, meaning node attributes are not considered
139: (8)                    in the comparison. Any node that doesn't have the `node_label`
140: (8)                    attribute uses `default_label` instead.
141: (4)                default_label : scalar
142: (8)                    Default value to use when a node doesn't have an attribute
143: (8)                    named `node_label`. Default is `None`.
144: (4)                Yields
145: (4)                ------
146: (4)                dict
147: (8)                    Isomorphic mapping between the nodes in `G1` and `G2`.
148: (4)                """
149: (4)                if G1.number_of_nodes() == 0 or G2.number_of_nodes() == 0:
150: (8)                    return False
151: (4)                if G1.is_directed():
152: (8)                    G1_degree = {
153: (12)                       n: (in_degree, out_degree)
154: (12)                       for (n, in_degree), (_, out_degree) in zip(G1.in_degree,
G1.out_degree)
155: (8)                    }
156: (8)                    G2_degree = {
157: (12)                       n: (in_degree, out_degree)
158: (12)                       for (n, in_degree), (_, out_degree) in zip(G2.in_degree,
G2.out_degree)
159: (8)                    }
```

```
160: (4)                else:
161: (8)                    G1_degree = dict(G1.degree)
162: (8)                    G2_degree = dict(G2.degree)
163: (4)                if not G1.is_directed():
164: (8)                    find_candidates = _find_candidates
165: (8)                    restore_Tinout = _restore_Tinout
166: (4)                else:
167: (8)                    find_candidates = _find_candidates_Di
168: (8)                    restore_Tinout = _restore_Tinout_Di
169: (4)                if G1.order() != G2.order():
170: (8)                    return False
171: (4)                if sorted(G1_degree.values()) != sorted(G2_degree.values()):
172: (8)                    return False
173: (4)                graph_params, state_params = _initialize_parameters(
174: (8)                    G1, G2, G2_degree, node_label, default_label
175: (4)                )
176: (4)                if not _precheck_label_properties(graph_params):
177: (8)                    return False
178: (4)                node_order = _matching_order(graph_params)
179: (4)                stack = []
180: (4)                candidates = iter(
181: (8)                    find_candidates(node_order[0], graph_params, state_params, G1_degree)
182: (4)                )
183: (4)                stack.append((node_order[0], candidates))
184: (4)                mapping = state_params.mapping
185: (4)                reverse_mapping = state_params.reverse_mapping
186: (4)                matching_node = 1
187: (4)                while stack:
188: (8)                    current_node, candidate_nodes = stack[-1]
189: (8)                    try:
190: (12)                       candidate = next(candidate_nodes)
191: (8)                    except StopIteration:
192: (12)                       stack.pop()
193: (12)                       matching_node -= 1
194: (12)                       if stack:
195: (16)                           popped_node1, _ = stack[-1]
196: (16)                           popped_node2 = mapping[popped_node1]
197: (16)                           mapping.pop(popped_node1)
198: (16)                           reverse_mapping.pop(popped_node2)
199: (16)                           restore_Tinout(popped_node1, popped_node2, graph_params,
state_params)
200: (12)                       continue
201: (8)                    if _feasibility(current_node, candidate, graph_params, state_params):
202: (12)                       if len(mapping) == G2.number_of_nodes() - 1:
203: (16)                           cp_mapping = mapping.copy()
204: (16)                           cp_mapping[current_node] = candidate
205: (16)                           yield cp_mapping
206: (16)                           continue
207: (12)                       mapping[current_node] = candidate
208: (12)                       reverse_mapping[candidate] = current_node
209: (12)                       _update_Tinout(current_node, candidate, graph_params,
state_params)
210: (12)                       candidates = iter(
211: (16)                           find_candidates(
212: (20)                               node_order[matching_node], graph_params, state_params,
G1_degree
213: (16)                           )
214: (12)                       )
215: (12)                       stack.append((node_order[matching_node], candidates))
216: (12)                       matching_node += 1
217: (0)            def _precheck_label_properties(graph_params):
218: (4)                G1, G2, G1_labels, G2_labels, nodes_of_G1Labels, nodes_of_G2Labels, _ =
graph_params
219: (4)                if any(
220: (8)                    label not in nodes_of_G1Labels or len(nodes_of_G1Labels[label]) !=
len(nodes)
221: (8)                    for label, nodes in nodes_of_G2Labels.items()
222: (4)                ):
223: (8)                    return False
```

```
224: (4)                return True
225: (0)            def _initialize_parameters(G1, G2, G2_degree, node_label=None,
default_label=-1):
226: (4)                """Initializes all the necessary parameters for VF2++
227: (4)                Parameters
228: (4)                ----------
229: (4)                G1,G2: NetworkX Graph or MultiGraph instances.
230: (8)                    The two graphs to check for isomorphism or monomorphism
231: (4)                G1_labels,G2_labels: dict
232: (8)                    The label of every node in G1 and G2 respectively
233: (4)                Returns
234: (4)                -------
235: (4)                graph_params: namedtuple
236: (8)                    Contains all the Graph-related parameters:
237: (8)                    G1,G2
238: (8)                    G1_labels,G2_labels: dict
239: (4)                state_params: namedtuple
240: (8)                    Contains all the State-related parameters:
241: (8)                    mapping: dict
242: (12)                       The mapping as extended so far. Maps nodes of G1 to nodes of G2
243: (8)                    reverse_mapping: dict
244: (12)                       The reverse mapping as extended so far. Maps nodes from G2 to
nodes of G1. It's basically "mapping" reversed
245: (8)                    T1, T2: set
246: (12)                       Ti contains uncovered neighbors of covered nodes from Gi, i.e.
nodes that are not in the mapping, but are
247: (12)                       neighbors of nodes that are.
248: (8)                    T1_out, T2_out: set
249: (12)                       Ti_out contains all the nodes from Gi, that are neither in the
mapping nor in Ti
250: (4)                """
251: (4)                G1_labels = dict(G1.nodes(data=node_label, default=default_label))
252: (4)                G2_labels = dict(G2.nodes(data=node_label, default=default_label))
253: (4)                graph_params = _GraphParameters(
254: (8)                    G1,
255: (8)                    G2,
256: (8)                    G1_labels,
257: (8)                    G2_labels,
258: (8)                    nx.utils.groups(G1_labels),
259: (8)                    nx.utils.groups(G2_labels),
260: (8)                    nx.utils.groups(G2_degree),
261: (4)                )
262: (4)                T1, T1_in = set(), set()
263: (4)                T2, T2_in = set(), set()
264: (4)                if G1.is_directed():
265: (8)                    T1_tilde, T1_tilde_in = (
266: (12)                       set(G1.nodes()),
267: (12)                       set(),
268: (8)                    )  # todo: do we need Ti_tilde_in? What nodes does it have?
269: (8)                    T2_tilde, T2_tilde_in = set(G2.nodes()), set()
270: (4)                else:
271: (8)                    T1_tilde, T1_tilde_in = set(G1.nodes()), set()
272: (8)                    T2_tilde, T2_tilde_in = set(G2.nodes()), set()
273: (4)                state_params = _StateParameters(
274: (8)                    {},
275: (8)                    {},
276: (8)                    T1,
277: (8)                    T1_in,
278: (8)                    T1_tilde,
279: (8)                    T1_tilde_in,
280: (8)                    T2,
281: (8)                    T2_in,
282: (8)                    T2_tilde,
283: (8)                    T2_tilde_in,
284: (4)                )
285: (4)                return graph_params, state_params
286: (0)            def _matching_order(graph_params):
287: (4)                """The node ordering as introduced in VF2++.
288: (4)                Notes
```

```
(4)                 -----
(4)                 Taking into account the structure of the Graph and the node labeling, the nodes are placed in an order such that,
(4)                 most of the unfruitful/infeasible branches of the search space can be pruned on high levels, significantly
(4)                 decreasing the number of visited states. The premise is that, the algorithm will be able to recognize
(4)                 inconsistencies early, proceeding to go deep into the search tree only if it's needed.
(4)                 Parameters
(4)                 ----------
(4)                 graph_params: namedtuple
(8)                     Contains:
(12)                        G1,G2: NetworkX Graph or MultiGraph instances.
(16)                            The two graphs to check for isomorphism or monomorphism.
(12)                        G1_labels,G2_labels: dict
(16)                            The label of every node in G1 and G2 respectively.
(4)                 Returns
(4)                 -------
(4)                 node_order: list
(8)                     The ordering of the nodes.
(4)                 """
(4)                 G1, G2, G1_labels, _, _, nodes_of_G2Labels, _ = graph_params
(4)                 if not G1 and not G2:
(8)                     return {}
(4)                 if G1.is_directed():
(8)                     G1 = G1.to_undirected(as_view=True)
(4)                 V1_unordered = set(G1.nodes())
(4)                 label_rarity = {label: len(nodes) for label, nodes in nodes_of_G2Labels.items()}
(4)                 used_degrees = {node: 0 for node in G1}
(4)                 node_order = []
(4)                 while V1_unordered:
(8)                     max_rarity = min(label_rarity[G1_labels[x]] for x in V1_unordered)
(8)                     rarest_nodes = [
(12)                        n for n in V1_unordered if label_rarity[G1_labels[n]] == max_rarity
(8)                     ]
(8)                     max_node = max(rarest_nodes, key=G1.degree)
(8)                     for dlevel_nodes in nx.bfs_layers(G1, max_node):
(12)                        nodes_to_add = dlevel_nodes.copy()
(12)                        while nodes_to_add:
(16)                            max_used_degree = max(used_degrees[n] for n in nodes_to_add)
(16)                            max_used_degree_nodes = [
(20)                                n for n in nodes_to_add if used_degrees[n] == max_used_degree
(16)                            ]
(16)                            max_degree = max(G1.degree[n] for n in max_used_degree_nodes)
(16)                            max_degree_nodes = [
(20)                                n for n in max_used_degree_nodes if G1.degree[n] == max_degree
(16)                            ]
(16)                            next_node = min(
(20)                                max_degree_nodes, key=lambda x: label_rarity[G1_labels[x]]
(16)                            )
(16)                            node_order.append(next_node)
(16)                            for node in G1.neighbors(next_node):
(20)                                used_degrees[node] += 1
(16)                            nodes_to_add.remove(next_node)
(16)                            label_rarity[G1_labels[next_node]] -= 1
(16)                            V1_unordered.discard(next_node)
(4)                 return node_order
(0)             def _find_candidates(
(4)                 u, graph_params, state_params, G1_degree
(0)             ):  # todo: make the 4th argument the degree of u
(4)                 """Given node u of G1, finds the candidates of u from G2.
(4)                 Parameters
(4)                 ----------
(4)                 u: Graph node
```

```
(8)                     The node from G1 for which to find the candidates from G2.
(4)                 graph_params: namedtuple
(8)                     Contains all the Graph-related parameters:
(8)                     G1,G2: NetworkX Graph or MultiGraph instances.
(12)                        The two graphs to check for isomorphism or monomorphism
(8)                     G1_labels,G2_labels: dict
(12)                        The label of every node in G1 and G2 respectively
(4)                 state_params: namedtuple
(8)                     Contains all the State-related parameters:
(8)                     mapping: dict
(12)                        The mapping as extended so far. Maps nodes of G1 to nodes of G2
(8)                     reverse_mapping: dict
(12)                        The reverse mapping as extended so far. Maps nodes from G2 to
nodes of G1. It's basically "mapping" reversed
(8)                     T1, T2: set
(12)                        Ti contains uncovered neighbors of covered nodes from Gi, i.e.
nodes that are not in the mapping, but are
(12)                        neighbors of nodes that are.
(8)                     T1_tilde, T2_tilde: set
(12)                        Ti_tilde contains all the nodes from Gi, that are neither in the
mapping nor in Ti
(4)                 Returns
(4)                 -------
(4)                 candidates: set
(8)                     The nodes from G2 which are candidates for u.
(4)                 """
(4)                 G1, G2, G1_labels, _, _, nodes_of_G2Labels, G2_nodes_of_degree =
graph_params
(4)                 mapping, reverse_mapping, _, _, _, _, _, _, T2_tilde, _ = state_params
(4)                 covered_nbrs = [nbr for nbr in G1[u] if nbr in mapping]
(4)                 if not covered_nbrs:
(8)                     candidates = set(nodes_of_G2Labels[G1_labels[u]])
(8)                     candidates.intersection_update(G2_nodes_of_degree[G1_degree[u]])
(8)                     candidates.intersection_update(T2_tilde)
(8)                     candidates.difference_update(reverse_mapping)
(8)                     if G1.is_multigraph():
(12)                        candidates.difference_update(
(16)                            {
(20)                                node
(20)                                for node in candidates
(20)                                if G1.number_of_edges(u, u) != G2.number_of_edges(node,
node)
(16)                            }
(12)                        )
(8)                     return candidates
(4)                 nbr1 = covered_nbrs[0]
(4)                 common_nodes = set(G2[mapping[nbr1]])
(4)                 for nbr1 in covered_nbrs[1:]:
(8)                     common_nodes.intersection_update(G2[mapping[nbr1]])
(4)                 common_nodes.difference_update(reverse_mapping)
(4)                 common_nodes.intersection_update(G2_nodes_of_degree[G1_degree[u]])
(4)                 common_nodes.intersection_update(nodes_of_G2Labels[G1_labels[u]])
(4)                 if G1.is_multigraph():
(8)                     common_nodes.difference_update(
(12)                        {
(16)                            node
(16)                            for node in common_nodes
(16)                            if G1.number_of_edges(u, u) != G2.number_of_edges(node, node)
(12)                        }
(8)                     )
(4)                 return common_nodes
(0)             def _find_candidates_Di(u, graph_params, state_params, G1_degree):
(4)                 G1, G2, G1_labels, _, _, nodes_of_G2Labels, G2_nodes_of_degree =
graph_params
(4)                 mapping, reverse_mapping, _, _, _, _, _, _, T2_tilde, _ = state_params
(4)                 covered_successors = [succ for succ in G1[u] if succ in mapping]
(4)                 covered_predecessors = [pred for pred in G1.pred[u] if pred in mapping]
(4)                 if not (covered_successors or covered_predecessors):
(8)                     candidates = set(nodes_of_G2Labels[G1_labels[u]])
```

```
413: (8)                    candidates.intersection_update(G2_nodes_of_degree[G1_degree[u]])
414: (8)                    candidates.intersection_update(T2_tilde)
415: (8)                    candidates.difference_update(reverse_mapping)
416: (8)                    if G1.is_multigraph():
417: (12)                       candidates.difference_update(
418: (16)                           {
419: (20)                               node
420: (20)                               for node in candidates
421: (20)                               if G1.number_of_edges(u, u) != G2.number_of_edges(node,
node)
422: (16)                           }
423: (12)                       )
424: (8)                    return candidates
425: (4)                if covered_successors:
426: (8)                    succ1 = covered_successors[0]
427: (8)                    common_nodes = set(G2.pred[mapping[succ1]])
428: (8)                    for succ1 in covered_successors[1:]:
429: (12)                       common_nodes.intersection_update(G2.pred[mapping[succ1]])
430: (4)                else:
431: (8)                    pred1 = covered_predecessors.pop()
432: (8)                    common_nodes = set(G2[mapping[pred1]])
433: (4)                for pred1 in covered_predecessors:
434: (8)                    common_nodes.intersection_update(G2[mapping[pred1]])
435: (4)                common_nodes.difference_update(reverse_mapping)
436: (4)                common_nodes.intersection_update(G2_nodes_of_degree[G1_degree[u]])
437: (4)                common_nodes.intersection_update(nodes_of_G2Labels[G1_labels[u]])
438: (4)                if G1.is_multigraph():
439: (8)                    common_nodes.difference_update(
440: (12)                       {
441: (16)                           node
442: (16)                           for node in common_nodes
443: (16)                           if G1.number_of_edges(u, u) != G2.number_of_edges(node, node)
444: (12)                       }
445: (8)                    )
446: (4)                return common_nodes
447: (0)            def _feasibility(node1, node2, graph_params, state_params):
448: (4)                """Given a candidate pair of nodes u and v from G1 and G2 respectively,
checks if it's feasible to extend the
449: (4)                mapping, i.e. if u and v can be matched.
450: (4)                Notes
451: (4)                -----
452: (4)                This function performs all the necessary checking by applying both
consistency and cutting rules.
453: (4)                Parameters
454: (4)                ----------
455: (4)                node1, node2: Graph node
456: (8)                    The candidate pair of nodes being checked for matching
457: (4)                graph_params: namedtuple
458: (8)                    Contains all the Graph-related parameters:
459: (8)                    G1,G2: NetworkX Graph or MultiGraph instances.
460: (12)                       The two graphs to check for isomorphism or monomorphism
461: (8)                    G1_labels,G2_labels: dict
462: (12)                       The label of every node in G1 and G2 respectively
463: (4)                state_params: namedtuple
464: (8)                    Contains all the State-related parameters:
465: (8)                    mapping: dict
466: (12)                       The mapping as extended so far. Maps nodes of G1 to nodes of G2
467: (8)                    reverse_mapping: dict
468: (12)                       The reverse mapping as extended so far. Maps nodes from G2 to
nodes of G1. It's basically "mapping" reversed
469: (8)                    T1, T2: set
470: (12)                       Ti contains uncovered neighbors of covered nodes from Gi, i.e.
nodes that are not in the mapping, but are
471: (12)                       neighbors of nodes that are.
472: (8)                    T1_out, T2_out: set
473: (12)                       Ti_out contains all the nodes from Gi, that are neither in the
mapping nor in Ti
474: (4)                Returns
475: (4)                -------
```

```
(4)                 True if all checks are successful, False otherwise.
(4)                 """
(4)                 G1 = graph_params.G1
(4)                 if _cut_PT(node1, node2, graph_params, state_params):
(8)                     return False
(4)                 if G1.is_multigraph():
(8)                     if not _consistent_PT(node1, node2, graph_params, state_params):
(12)                        return False
(4)                 return True
(0)             def _cut_PT(u, v, graph_params, state_params):
(4)                 """Implements the cutting rules for the ISO problem.
(4)                 Parameters
(4)                 ----------
(4)                 u, v: Graph node
(8)                     The two candidate nodes being examined.
(4)                 graph_params: namedtuple
(8)                     Contains all the Graph-related parameters:
(8)                     G1,G2: NetworkX Graph or MultiGraph instances.
(12)                        The two graphs to check for isomorphism or monomorphism
(8)                     G1_labels,G2_labels: dict
(12)                        The label of every node in G1 and G2 respectively
(4)                 state_params: namedtuple
(8)                     Contains all the State-related parameters:
(8)                     mapping: dict
(12)                        The mapping as extended so far. Maps nodes of G1 to nodes of G2
(8)                     reverse_mapping: dict
(12)                        The reverse mapping as extended so far. Maps nodes from G2 to
nodes of G1. It's basically "mapping" reversed
(8)                     T1, T2: set
(12)                        Ti contains uncovered neighbors of covered nodes from Gi, i.e.
nodes that are not in the mapping, but are
(12)                        neighbors of nodes that are.
(8)                     T1_tilde, T2_tilde: set
(12)                        Ti_out contains all the nodes from Gi, that are neither in the
mapping nor in Ti
(4)                 Returns
(4)                 -------
(4)                 True if we should prune this branch, i.e. the node pair failed the cutting
checks. False otherwise.
(4)                 """
(4)                 G1, G2, G1_labels, G2_labels, _, _, _ = graph_params
(4)                 (
(8)                     _,
(8)                     _,
(8)                     T1,
(8)                     T1_in,
(8)                     T1_tilde,
(8)                     _,
(8)                     T2,
(8)                     T2_in,
(8)                     T2_tilde,
(8)                     _,
(4)                 ) = state_params
(4)                 u_labels_predecessors, v_labels_predecessors = {}, {}
(4)                 if G1.is_directed():
(8)                     u_labels_predecessors = nx.utils.groups(
(12)                        {n1: G1_labels[n1] for n1 in G1.pred[u]}
(8)                     )
(8)                     v_labels_predecessors = nx.utils.groups(
(12)                        {n2: G2_labels[n2] for n2 in G2.pred[v]}
(8)                     )
(8)                     if set(u_labels_predecessors.keys()) !=
set(v_labels_predecessors.keys()):
(12)                        return True
(4)                 u_labels_successors = nx.utils.groups({n1: G1_labels[n1] for n1 in G1[u]})
(4)                 v_labels_successors = nx.utils.groups({n2: G2_labels[n2] for n2 in G2[v]})
(4)                 if set(u_labels_successors.keys()) != set(v_labels_successors.keys()):
(8)                     return True
(4)                 for label, G1_nbh in u_labels_successors.items():
```

```
540: (8)                    G2_nbh = v_labels_successors[label]
541: (8)                    if G1.is_multigraph():
542: (12)                       u_nbrs_edges = sorted(G1.number_of_edges(u, x) for x in G1_nbh)
543: (12)                       v_nbrs_edges = sorted(G2.number_of_edges(v, x) for x in G2_nbh)
544: (12)                       if any(
545: (16)                           u_nbr_edges != v_nbr_edges
546: (16)                           for u_nbr_edges, v_nbr_edges in zip(u_nbrs_edges,
v_nbrs_edges)
547: (12)                       ):
548: (16)                           return True
549: (8)                    if len(T1.intersection(G1_nbh)) != len(T2.intersection(G2_nbh)):
550: (12)                       return True
551: (8)                    if len(T1_tilde.intersection(G1_nbh)) !=
len(T2_tilde.intersection(G2_nbh)):
552: (12)                       return True
553: (8)                    if G1.is_directed() and len(T1_in.intersection(G1_nbh)) != len(
554: (12)                       T2_in.intersection(G2_nbh)
555: (8)                    ):
556: (12)                       return True
557: (4)                if not G1.is_directed():
558: (8)                    return False
559: (4)                for label, G1_pred in u_labels_predecessors.items():
560: (8)                    G2_pred = v_labels_predecessors[label]
561: (8)                    if G1.is_multigraph():
562: (12)                       u_pred_edges = sorted(G1.number_of_edges(u, x) for x in G1_pred)
563: (12)                       v_pred_edges = sorted(G2.number_of_edges(v, x) for x in G2_pred)
564: (12)                       if any(
565: (16)                           u_nbr_edges != v_nbr_edges
566: (16)                           for u_nbr_edges, v_nbr_edges in zip(u_pred_edges,
v_pred_edges)
567: (12)                       ):
568: (16)                           return True
569: (8)                    if len(T1.intersection(G1_pred)) != len(T2.intersection(G2_pred)):
570: (12)                       return True
571: (8)                    if len(T1_tilde.intersection(G1_pred)) !=
len(T2_tilde.intersection(G2_pred)):
572: (12)                       return True
573: (8)                    if len(T1_in.intersection(G1_pred)) !=
len(T2_in.intersection(G2_pred)):
574: (12)                       return True
575: (4)                return False
576: (0)            def _consistent_PT(u, v, graph_params, state_params):
577: (4)                """Checks the consistency of extending the mapping using the current node
pair.
578: (4)                Parameters
579: (4)                ----------
580: (4)                u, v: Graph node
581: (8)                    The two candidate nodes being examined.
582: (4)                graph_params: namedtuple
583: (8)                    Contains all the Graph-related parameters:
584: (8)                    G1,G2: NetworkX Graph or MultiGraph instances.
585: (12)                       The two graphs to check for isomorphism or monomorphism
586: (8)                    G1_labels,G2_labels: dict
587: (12)                       The label of every node in G1 and G2 respectively
588: (4)                state_params: namedtuple
589: (8)                    Contains all the State-related parameters:
590: (8)                    mapping: dict
591: (12)                       The mapping as extended so far. Maps nodes of G1 to nodes of G2
592: (8)                    reverse_mapping: dict
593: (12)                       The reverse mapping as extended so far. Maps nodes from G2 to
nodes of G1. It's basically "mapping" reversed
594: (8)                    T1, T2: set
595: (12)                       Ti contains uncovered neighbors of covered nodes from Gi, i.e.
nodes that are not in the mapping, but are
596: (12)                       neighbors of nodes that are.
597: (8)                    T1_out, T2_out: set
598: (12)                       Ti_out contains all the nodes from Gi, that are neither in the
mapping nor in Ti
599: (4)                Returns
```

```
    -------
    True if the pair passes all the consistency checks successfully. False otherwise.
    """
    G1, G2 = graph_params.G1, graph_params.G2
    mapping, reverse_mapping = state_params.mapping, state_params.reverse_mapping
    for neighbor in G1[u]:
        if neighbor in mapping:
            if G1.number_of_edges(u, neighbor) != G2.number_of_edges(
                v, mapping[neighbor]
            ):
                return False
    for neighbor in G2[v]:
        if neighbor in reverse_mapping:
            if G1.number_of_edges(u, reverse_mapping[neighbor]) != G2.number_of_edges(
                v, neighbor
            ):
                return False
    if not G1.is_directed():
        return True
    for predecessor in G1.pred[u]:
        if predecessor in mapping:
            if G1.number_of_edges(predecessor, u) != G2.number_of_edges(
                mapping[predecessor], v
            ):
                return False
    for predecessor in G2.pred[v]:
        if predecessor in reverse_mapping:
            if G1.number_of_edges(
                reverse_mapping[predecessor], u
            ) != G2.number_of_edges(predecessor, v):
                return False
    return True
def _update_Tinout(new_node1, new_node2, graph_params, state_params):
    """Updates the Ti/Ti_out (i=1,2) when a new node pair u-v is added to the mapping.
    Notes
    -----
    This function should be called right after the feasibility checks are passed, and node1 is mapped to node2. The
    purpose of this function is to avoid brute force computing of Ti/Ti_out by iterating over all nodes of the graph
    and checking which nodes satisfy the necessary conditions. Instead, in every step of the algorithm we focus
    exclusively on the two nodes that are being added to the mapping, incrementally updating Ti/Ti_out.
    Parameters
    ----------
    new_node1, new_node2: Graph node
        The two new nodes, added to the mapping.
    graph_params: namedtuple
        Contains all the Graph-related parameters:
        G1,G2: NetworkX Graph or MultiGraph instances.
            The two graphs to check for isomorphism or monomorphism
        G1_labels,G2_labels: dict
            The label of every node in G1 and G2 respectively
    state_params: namedtuple
        Contains all the State-related parameters:
        mapping: dict
            The mapping as extended so far. Maps nodes of G1 to nodes of G2
        reverse_mapping: dict
            The reverse mapping as extended so far. Maps nodes from G2 to nodes of G1. It's basically "mapping" reversed
        T1, T2: set
            Ti contains uncovered neighbors of covered nodes from Gi, i.e. nodes that are not in the mapping, but are
            neighbors of nodes that are.
```

```
659: (8)                    T1_tilde, T2_tilde: set
660: (12)                       Ti_out contains all the nodes from Gi, that are neither in the
mapping nor in Ti
661: (4)                """
662: (4)                G1, G2, _, _, _, _, _ = graph_params
663: (4)                (
664: (8)                    mapping,
665: (8)                    reverse_mapping,
666: (8)                    T1,
667: (8)                    T1_in,
668: (8)                    T1_tilde,
669: (8)                    T1_tilde_in,
670: (8)                    T2,
671: (8)                    T2_in,
672: (8)                    T2_tilde,
673: (8)                    T2_tilde_in,
674: (4)                ) = state_params
675: (4)                uncovered_successors_G1 = {succ for succ in G1[new_node1] if succ not in
mapping}
676: (4)                uncovered_successors_G2 = {
677: (8)                    succ for succ in G2[new_node2] if succ not in reverse_mapping
678: (4)                }
679: (4)                T1.update(uncovered_successors_G1)
680: (4)                T2.update(uncovered_successors_G2)
681: (4)                T1.discard(new_node1)
682: (4)                T2.discard(new_node2)
683: (4)                T1_tilde.difference_update(uncovered_successors_G1)
684: (4)                T2_tilde.difference_update(uncovered_successors_G2)
685: (4)                T1_tilde.discard(new_node1)
686: (4)                T2_tilde.discard(new_node2)
687: (4)                if not G1.is_directed():
688: (8)                    return
689: (4)                uncovered_predecessors_G1 = {
690: (8)                    pred for pred in G1.pred[new_node1] if pred not in mapping
691: (4)                }
692: (4)                uncovered_predecessors_G2 = {
693: (8)                    pred for pred in G2.pred[new_node2] if pred not in reverse_mapping
694: (4)                }
695: (4)                T1_in.update(uncovered_predecessors_G1)
696: (4)                T2_in.update(uncovered_predecessors_G2)
697: (4)                T1_in.discard(new_node1)
698: (4)                T2_in.discard(new_node2)
699: (4)                T1_tilde.difference_update(uncovered_predecessors_G1)
700: (4)                T2_tilde.difference_update(uncovered_predecessors_G2)
701: (4)                T1_tilde.discard(new_node1)
702: (4)                T2_tilde.discard(new_node2)
703: (0)            def _restore_Tinout(popped_node1, popped_node2, graph_params, state_params):
704: (4)                """Restores the previous version of Ti/Ti_out when a node pair is deleted
from the mapping.
705: (4)                Parameters
706: (4)                ----------
707: (4)                popped_node1, popped_node2: Graph node
708: (8)                    The two nodes deleted from the mapping.
709: (4)                graph_params: namedtuple
710: (8)                    Contains all the Graph-related parameters:
711: (8)                    G1,G2: NetworkX Graph or MultiGraph instances.
712: (12)                       The two graphs to check for isomorphism or monomorphism
713: (8)                    G1_labels,G2_labels: dict
714: (12)                       The label of every node in G1 and G2 respectively
715: (4)                state_params: namedtuple
716: (8)                    Contains all the State-related parameters:
717: (8)                    mapping: dict
718: (12)                       The mapping as extended so far. Maps nodes of G1 to nodes of G2
719: (8)                    reverse_mapping: dict
720: (12)                       The reverse mapping as extended so far. Maps nodes from G2 to
nodes of G1. It's basically "mapping" reversed
721: (8)                    T1, T2: set
722: (12)                       Ti contains uncovered neighbors of covered nodes from Gi, i.e.
nodes that are not in the mapping, but are
```

```
723: (12)                        neighbors of nodes that are.
724: (8)                    T1_tilde, T2_tilde: set
725: (12)                        Ti_out contains all the nodes from Gi, that are neither in the
mapping nor in Ti
726: (4)                """
727: (4)                G1, G2, _, _, _, _, _ = graph_params
728: (4)                (
729: (8)                    mapping,
730: (8)                    reverse_mapping,
731: (8)                    T1,
732: (8)                    T1_in,
733: (8)                    T1_tilde,
734: (8)                    T1_tilde_in,
735: (8)                    T2,
736: (8)                    T2_in,
737: (8)                    T2_tilde,
738: (8)                    T2_tilde_in,
739: (4)                ) = state_params
740: (4)                is_added = False
741: (4)                for neighbor in G1[popped_node1]:
742: (8)                    if neighbor in mapping:
743: (12)                       is_added = True
744: (12)                       T1.add(popped_node1)
745: (8)                    else:
746: (12)                       if any(nbr in mapping for nbr in G1[neighbor]):
747: (16)                           continue
748: (12)                       T1.discard(neighbor)
749: (12)                       T1_tilde.add(neighbor)
750: (4)                if not is_added:
751: (8)                    T1_tilde.add(popped_node1)
752: (4)                is_added = False
753: (4)                for neighbor in G2[popped_node2]:
754: (8)                    if neighbor in reverse_mapping:
755: (12)                       is_added = True
756: (12)                       T2.add(popped_node2)
757: (8)                    else:
758: (12)                       if any(nbr in reverse_mapping for nbr in G2[neighbor]):
759: (16)                           continue
760: (12)                       T2.discard(neighbor)
761: (12)                       T2_tilde.add(neighbor)
762: (4)                if not is_added:
763: (8)                    T2_tilde.add(popped_node2)
764: (0)            def _restore_Tinout_Di(popped_node1, popped_node2, graph_params,
state_params):
765: (4)                G1, G2, _, _, _, _, _ = graph_params
766: (4)                (
767: (8)                    mapping,
768: (8)                    reverse_mapping,
769: (8)                    T1,
770: (8)                    T1_in,
771: (8)                    T1_tilde,
772: (8)                    T1_tilde_in,
773: (8)                    T2,
774: (8)                    T2_in,
775: (8)                    T2_tilde,
776: (8)                    T2_tilde_in,
777: (4)                ) = state_params
778: (4)                is_added = False
779: (4)                for successor in G1[popped_node1]:
780: (8)                    if successor in mapping:
781: (12)                       is_added = True
782: (12)                       T1_in.add(popped_node1)
783: (8)                    else:
784: (12)                       if not any(pred in mapping for pred in G1.pred[successor]):
785: (16)                           T1.discard(successor)
786: (12)                       if not any(succ in mapping for succ in G1[successor]):
787: (16)                           T1_in.discard(successor)
788: (12)                       if successor not in T1:
789: (16)                           if successor not in T1_in:
```

```
790: (20)                                T1_tilde.add(successor)
791: (4)                for predecessor in G1.pred[popped_node1]:
792: (8)                    if predecessor in mapping:
793: (12)                       is_added = True
794: (12)                       T1.add(popped_node1)
795: (8)                    else:
796: (12)                       if not any(pred in mapping for pred in G1.pred[predecessor]):
797: (16)                           T1.discard(predecessor)
798: (12)                       if not any(succ in mapping for succ in G1[predecessor]):
799: (16)                           T1_in.discard(predecessor)
800: (12)                       if not (predecessor in T1 or predecessor in T1_in):
801: (16)                           T1_tilde.add(predecessor)
802: (4)                if not is_added:
803: (8)                    T1_tilde.add(popped_node1)
804: (4)                is_added = False
805: (4)                for successor in G2[popped_node2]:
806: (8)                    if successor in reverse_mapping:
807: (12)                       is_added = True
808: (12)                       T2_in.add(popped_node2)
809: (8)                    else:
810: (12)                       if not any(pred in reverse_mapping for pred in
G2.pred[successor]):
811: (16)                           T2.discard(successor)
812: (12)                       if not any(succ in reverse_mapping for succ in G2[successor]):
813: (16)                           T2_in.discard(successor)
814: (12)                       if successor not in T2:
815: (16)                           if successor not in T2_in:
816: (20)                               T2_tilde.add(successor)
817: (4)                for predecessor in G2.pred[popped_node2]:
818: (8)                    if predecessor in reverse_mapping:
819: (12)                       is_added = True
820: (12)                       T2.add(popped_node2)
821: (8)                    else:
822: (12)                       if not any(pred in reverse_mapping for pred in
G2.pred[predecessor]):
823: (16)                           T2.discard(predecessor)
824: (12)                       if not any(succ in reverse_mapping for succ in G2[predecessor]):
825: (16)                           T2_in.discard(predecessor)
826: (12)                       if not (predecessor in T2 or predecessor in T2_in):
827: (16)                           T2_tilde.add(predecessor)
828: (4)                if not is_added:
829: (8)                    T2_tilde.add(popped_node2)
```

File 261 - tree_isomorphism.py:

```
1: (0)              """
2: (0)              An algorithm for finding if two undirected trees are isomorphic,
3: (0)              and if so returns an isomorphism between the two sets of nodes.
4: (0)              This algorithm uses a routine to tell if two rooted trees (trees with a
5: (0)              specified root node) are isomorphic, which may be independently useful.
6: (0)              This implements an algorithm from:
7: (0)              The Design and Analysis of Computer Algorithms
8: (0)              by Aho, Hopcroft, and Ullman
9: (0)              Addison-Wesley Publishing 1974
10: (0)             Example 3.2 pp. 84-86.
11: (0)             A more understandable version of this algorithm is described in:
12: (0)             Homework Assignment 5
13: (0)             McGill University SOCS 308-250B, Winter 2002
14: (0)             by Matthew Suderman
15: (0)             http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/HW5+.pdf
16: (0)             """
17: (0)             import networkx as nx
18: (0)             from networkx.utils.decorators import not_implemented_for
19: (0)             __all__ = ["rooted_tree_isomorphism", "tree_isomorphism"]
20: (0)             @nx._dispatchable(graphs={"t1": 0, "t2": 2}, returns_graph=True)
21: (0)             def root_trees(t1, root1, t2, root2):
22: (4)                 """Create a single digraph dT of free trees t1 and t2
```

```
(4)                 """
(4)                 dT = nx.DiGraph()
(4)                 newroot1 = 1  # left root will be 1
(4)                 newroot2 = nx.number_of_nodes(t1) + 1  # right will be n+1
(4)                 namemap1 = {root1: newroot1}
(4)                 namemap2 = {root2: newroot2}
(4)                 dT.add_edge(0, namemap1[root1])
(4)                 dT.add_edge(0, namemap2[root2])
(4)                 for i, (v1, v2) in enumerate(nx.bfs_edges(t1, root1)):
(8)                     namemap1[v2] = i + namemap1[root1] + 1
(8)                     dT.add_edge(namemap1[v1], namemap1[v2])
(4)                 for i, (v1, v2) in enumerate(nx.bfs_edges(t2, root2)):
(8)                     namemap2[v2] = i + namemap2[root2] + 1
(8)                     dT.add_edge(namemap2[v1], namemap2[v2])
(4)                 namemap = {}
(4)                 for old, new in namemap1.items():
(8)                     namemap[new] = old
(4)                 for old, new in namemap2.items():
(8)                     namemap[new] = old
(4)                 return (dT, namemap, newroot1, newroot2)
(0)             @nx._dispatchable
(0)             def assign_levels(G, root):
(4)                 level = {}
(4)                 level[root] = 0
(4)                 for v1, v2 in nx.bfs_edges(G, root):
(8)                     level[v2] = level[v1] + 1
(4)                 return level
(0)             def group_by_levels(levels):
(4)                 L = {}
(4)                 for n, lev in levels.items():
(8)                     if lev not in L:
(12)                        L[lev] = []
(8)                     L[lev].append(n)
(4)                 return L
(0)             def generate_isomorphism(v, w, M, ordered_children):
(4)                 assert v < w
(4)                 M.append((v, w))
(4)                 for i, (x, y) in enumerate(zip(ordered_children[v], ordered_children[w])):
(8)                     generate_isomorphism(x, y, M, ordered_children)
(0)             @nx._dispatchable(graphs={"t1": 0, "t2": 2})
(0)             def rooted_tree_isomorphism(t1, root1, t2, root2):
(4)                 """
(4)                 Given two rooted trees `t1` and `t2`,
(4)                 with roots `root1` and `root2` respectively
(4)                 this routine will determine if they are isomorphic.
(4)                 These trees may be either directed or undirected,
(4)                 but if they are directed, all edges should flow from the root.
(4)                 It returns the isomorphism, a mapping of the nodes of `t1` onto the nodes
(4)                 of `t2`, such that two trees are then identical.
(4)                 Note that two trees may have more than one isomorphism, and this
(4)                 routine just returns one valid mapping.
(4)                 Parameters
(4)                 ----------
(4)                 `t1` :  NetworkX graph
(8)                     One of the trees being compared
(4)                 `root1` : a node of `t1` which is the root of the tree
(4)                 `t2` : undirected NetworkX graph
(8)                     The other tree being compared
(4)                 `root2` : a node of `t2` which is the root of the tree
(4)                 This is a subroutine used to implement `tree_isomorphism`, but will
(4)                 be somewhat faster if you already have rooted trees.
(4)                 Returns
(4)                 -------
(4)                 isomorphism : list
(8)                     A list of pairs in which the left element is a node in `t1`
(8)                     and the right element is a node in `t2`.  The pairs are in
(8)                     arbitrary order.  If the nodes in one tree is mapped to the names in
(8)                     the other, then trees will be identical. Note that an isomorphism
(8)                     will not necessarily be unique.
```

```
 92: (8)                     If `t1` and `t2` are not isomorphic, then it returns the empty list.
 93: (4)                 """
 94: (4)                 assert nx.is_tree(t1)
 95: (4)                 assert nx.is_tree(t2)
 96: (4)                 (dT, namemap, newroot1, newroot2) = root_trees(t1, root1, t2, root2)
 97: (4)                 levels = assign_levels(dT, 0)
 98: (4)                 h = max(levels.values())
 99: (4)                 L = group_by_levels(levels)
 100: (4)                label = {v: 0 for v in dT}
 101: (4)                ordered_labels = {v: () for v in dT}
 102: (4)                ordered_children = {v: () for v in dT}
 103: (4)                for i in range(h - 1, 0, -1):
 104: (8)                    for v in L[i]:
 105: (12)                       if dT.out_degree(v) > 0:
 106: (16)                           s = sorted((label[u], u) for u in dT.successors(v))
 107: (16)                           ordered_labels[v], ordered_children[v] = list(zip(*s))
 108: (8)                    forlabel = sorted((ordered_labels[v], v) for v in L[i])
 109: (8)                    current = 0
 110: (8)                    for i, (ol, v) in enumerate(forlabel):
 111: (12)                       if (i != 0) and (ol != forlabel[i - 1][0]):
 112: (16)                           current += 1
 113: (12)                       label[v] = current
 114: (4)                isomorphism = []
 115: (4)                if label[newroot1] == 0 and label[newroot2] == 0:
 116: (8)                    generate_isomorphism(newroot1, newroot2, isomorphism,
ordered_children)
 117: (8)                    isomorphism = [(namemap[u], namemap[v]) for (u, v) in isomorphism]
 118: (4)                return isomorphism
 119: (0)            @not_implemented_for("directed")
 120: (0)            @not_implemented_for("multigraph")
 121: (0)            @nx._dispatchable(graphs={"t1": 0, "t2": 1})
 122: (0)            def tree_isomorphism(t1, t2):
 123: (4)                """
 124: (4)                Given two undirected (or free) trees `t1` and `t2`,
 125: (4)                this routine will determine if they are isomorphic.
 126: (4)                It returns the isomorphism, a mapping of the nodes of `t1` onto the nodes
 127: (4)                of `t2`, such that two trees are then identical.
 128: (4)                Note that two trees may have more than one isomorphism, and this
 129: (4)                routine just returns one valid mapping.
 130: (4)                Parameters
 131: (4)                ----------
 132: (4)                t1 : undirected NetworkX graph
 133: (8)                    One of the trees being compared
 134: (4)                t2 : undirected NetworkX graph
 135: (8)                    The other tree being compared
 136: (4)                Returns
 137: (4)                -------
 138: (4)                isomorphism : list
 139: (8)                    A list of pairs in which the left element is a node in `t1`
 140: (8)                    and the right element is a node in `t2`.  The pairs are in
 141: (8)                    arbitrary order.  If the nodes in one tree is mapped to the names in
 142: (8)                    the other, then trees will be identical. Note that an isomorphism
 143: (8)                    will not necessarily be unique.
 144: (8)                    If `t1` and `t2` are not isomorphic, then it returns the empty list.
 145: (4)                Notes
 146: (4)                -----
 147: (4)                This runs in O(n*log(n)) time for trees with n nodes.
 148: (4)                """
 149: (4)                assert nx.is_tree(t1)
 150: (4)                assert nx.is_tree(t2)
 151: (4)                if nx.number_of_nodes(t1) != nx.number_of_nodes(t2):
 152: (8)                    return []
 153: (4)                degree_sequence1 = sorted(d for (n, d) in t1.degree())
 154: (4)                degree_sequence2 = sorted(d for (n, d) in t2.degree())
 155: (4)                if degree_sequence1 != degree_sequence2:
 156: (8)                    return []
 157: (4)                center1 = nx.center(t1)
 158: (4)                center2 = nx.center(t2)
 159: (4)                if len(center1) != len(center2):
```

```
160: (8)                    return []
161: (4)                if len(center1) == 1:
162: (8)                    return rooted_tree_isomorphism(t1, center1[0], t2, center2[0])
163: (4)                attempts = rooted_tree_isomorphism(t1, center1[0], t2, center2[0])
164: (4)                if len(attempts) > 0:
165: (8)                    return attempts
166: (4)                return rooted_tree_isomorphism(t1, center1[0], t2, center2[1])
```

File 262 - __init__.py:

```
1: (0)
```

File 263 - vf2userfunc.py:

```
1: (0)              """
2: (4)                  Module to simplify the specification of user-defined equality functions
for
3: (4)                  node and edge attributes during isomorphism checks.
4: (4)                  During the construction of an isomorphism, the algorithm considers two
5: (4)                  candidate nodes n1 in G1 and n2 in G2.  The graphs G1 and G2 are then
6: (4)                  compared with respect to properties involving n1 and n2, and if the
outcome
7: (4)                  is good, then the candidate nodes are considered isomorphic. NetworkX
8: (4)                  provides a simple mechanism for users to extend the comparisons to include
9: (4)                  node and edge attributes.
10: (4)                 Node attributes are handled by the node_match keyword. When considering
11: (4)                 n1 and n2, the algorithm passes their node attribute dictionaries to
12: (4)                 node_match, and if it returns False, then n1 and n2 cannot be
13: (4)                 considered to be isomorphic.
14: (4)                 Edge attributes are handled by the edge_match keyword. When considering
15: (4)                 n1 and n2, the algorithm must verify that outgoing edges from n1 are
16: (4)                 commensurate with the outgoing edges for n2. If the graph is directed,
17: (4)                 then a similar check is also performed for incoming edges.
18: (4)                 Focusing only on outgoing edges, we consider pairs of nodes (n1, v1) from
19: (4)                 G1 and (n2, v2) from G2. For graphs and digraphs, there is only one edge
20: (4)                 between (n1, v1) and only one edge between (n2, v2). Those edge attribute
21: (4)                 dictionaries are passed to edge_match, and if it returns False, then
22: (4)                 n1 and n2 cannot be considered isomorphic. For multigraphs and
23: (4)                 multidigraphs, there can be multiple edges between (n1, v1) and also
24: (4)                 multiple edges between (n2, v2).  Now, there must exist an isomorphism
25: (4)                 from "all the edges between (n1, v1)" to "all the edges between (n2, v2)".
26: (4)                 So, all of the edge attribute dictionaries are passed to edge_match, and
27: (4)                 it must determine if there is an isomorphism between the two sets of
edges.
28: (0)             """
29: (0)             from . import isomorphvf2 as vf2
30: (0)             __all__ = ["GraphMatcher", "DiGraphMatcher", "MultiGraphMatcher",
"MultiDiGraphMatcher"]
31: (0)             def _semantic_feasibility(self, G1_node, G2_node):
32: (4)                 """Returns True if mapping G1_node to G2_node is semantically feasible."""
33: (4)                 if self.node_match is not None:
34: (8)                     nm = self.node_match(self.G1.nodes[G1_node], self.G2.nodes[G2_node])
35: (8)                     if not nm:
36: (12)                        return False
37: (4)                 if self.edge_match is not None:
38: (8)                     G1nbrs = self.G1_adj[G1_node]
39: (8)                     G2nbrs = self.G2_adj[G2_node]
40: (8)                     core_1 = self.core_1
41: (8)                     edge_match = self.edge_match
42: (8)                     for neighbor in G1nbrs:
43: (12)                        if neighbor == G1_node:
44: (16)                            if G2_node in G2nbrs and not edge_match(
45: (20)                                G1nbrs[G1_node], G2nbrs[G2_node]
46: (16)                            ):
47: (20)                                return False
```

```
(12)                        elif neighbor in core_1:
(16)                            G2_nbr = core_1[neighbor]
(16)                            if G2_nbr in G2nbrs and not edge_match(
(20)                                G1nbrs[neighbor], G2nbrs[G2_nbr]
(16)                            ):
(20)                                return False
(4)                 return True
(0)             class GraphMatcher(vf2.GraphMatcher):
(4)                 """VF2 isomorphism checker for undirected graphs."""
(4)                 def __init__(self, G1, G2, node_match=None, edge_match=None):
(8)                     """Initialize graph matcher.
(8)                     Parameters
(8)                     ----------
(8)                     G1, G2: graph
(12)                        The graphs to be tested.
(8)                     node_match: callable
(12)                        A function that returns True iff node n1 in G1 and n2 in G2
(12)                        should be considered equal during the isomorphism test. The
(12)                        function will be called like::
(15)                           node_match(G1.nodes[n1], G2.nodes[n2])
(12)                        That is, the function will receive the node attribute dictionaries
(12)                        of the nodes under consideration. If None, then no attributes are
(12)                        considered when testing for an isomorphism.
(8)                     edge_match: callable
(12)                        A function that returns True iff the edge attribute dictionary for
(12)                        the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be
(12)                        considered equal during the isomorphism test. The function will be
(12)                        called like::
(15)                           edge_match(G1[u1][v1], G2[u2][v2])
(12)                        That is, the function will receive the edge attribute dictionaries
(12)                        of the edges under consideration. If None, then no attributes are
(12)                        considered when testing for an isomorphism.
(8)                     """
(8)                     vf2.GraphMatcher.__init__(self, G1, G2)
(8)                     self.node_match = node_match
(8)                     self.edge_match = edge_match
(8)                     self.G1_adj = self.G1.adj
(8)                     self.G2_adj = self.G2.adj
(4)                 semantic_feasibility = _semantic_feasibility
(0)             class DiGraphMatcher(vf2.DiGraphMatcher):
(4)                 """VF2 isomorphism checker for directed graphs."""
(4)                 def __init__(self, G1, G2, node_match=None, edge_match=None):
(8)                     """Initialize graph matcher.
(8)                     Parameters
(8)                     ----------
(8)                     G1, G2 : graph
(12)                        The graphs to be tested.
(8)                     node_match : callable
(12)                        A function that returns True iff node n1 in G1 and n2 in G2
(12)                        should be considered equal during the isomorphism test. The
(12)                        function will be called like::
(15)                           node_match(G1.nodes[n1], G2.nodes[n2])
(12)                        That is, the function will receive the node attribute dictionaries
(12)                        of the nodes under consideration. If None, then no attributes are
(12)                        considered when testing for an isomorphism.
(8)                     edge_match : callable
(12)                        A function that returns True iff the edge attribute dictionary for
(12)                        the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be
(12)                        considered equal during the isomorphism test. The function will be
(12)                        called like::
(15)                           edge_match(G1[u1][v1], G2[u2][v2])
(12)                        That is, the function will receive the edge attribute dictionaries
(12)                        of the edges under consideration. If None, then no attributes are
(12)                        considered when testing for an isomorphism.
(8)                     """
(8)                     vf2.DiGraphMatcher.__init__(self, G1, G2)
(8)                     self.node_match = node_match
(8)                     self.edge_match = edge_match
(8)                     self.G1_adj = self.G1.adj
```

```
117: (8)                    self.G2_adj = self.G2.adj
118: (4)                def semantic_feasibility(self, G1_node, G2_node):
119: (8)                    """Returns True if mapping G1_node to G2_node is semantically
feasible."""
120: (8)                    feasible = _semantic_feasibility(self, G1_node, G2_node)
121: (8)                    if not feasible:
122: (12)                       return False
123: (8)                    self.G1_adj = self.G1.pred
124: (8)                    self.G2_adj = self.G2.pred
125: (8)                    feasible = _semantic_feasibility(self, G1_node, G2_node)
126: (8)                    self.G1_adj = self.G1.adj
127: (8)                    self.G2_adj = self.G2.adj
128: (8)                    return feasible
129: (0)            class MultiGraphMatcher(GraphMatcher):
130: (4)                """VF2 isomorphism checker for undirected multigraphs."""
131: (0)            class MultiDiGraphMatcher(DiGraphMatcher):
132: (4)                """VF2 isomorphism checker for directed multigraphs."""
```

File 264 - test_ismags.py:

```
1: (0)              """
2: (4)                  Tests for ISMAGS isomorphism algorithm.
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms import isomorphism as iso
7: (0)              def _matches_to_sets(matches):
8: (4)                  """
9: (4)                  Helper function to facilitate comparing collections of dictionaries in
10: (4)                 which order does not matter.
11: (4)                 """
12: (4)                 return {frozenset(m.items()) for m in matches}
13: (0)             class TestSelfIsomorphism:
14: (4)                 data = [
15: (8)                     (
16: (12)                        [
17: (16)                            (0, {"name": "a"}),
18: (16)                            (1, {"name": "a"}),
19: (16)                            (2, {"name": "b"}),
20: (16)                            (3, {"name": "b"}),
21: (16)                            (4, {"name": "a"}),
22: (16)                            (5, {"name": "a"}),
23: (12)                        ],
24: (12)                        [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)],
25: (8)                     ),
26: (8)                     (range(1, 5), [(1, 2), (2, 4), (4, 3), (3, 1)]),
27: (8)                     (
28: (12)                        [],
29: (12)                        [
30: (16)                            (0, 1),
31: (16)                            (1, 2),
32: (16)                            (2, 3),
33: (16)                            (3, 4),
34: (16)                            (4, 5),
35: (16)                            (5, 0),
36: (16)                            (0, 6),
37: (16)                            (6, 7),
38: (16)                            (2, 8),
39: (16)                            (8, 9),
40: (16)                            (4, 10),
41: (16)                            (10, 11),
42: (12)                        ],
43: (8)                     ),
44: (8)                     ([], [(0, 1), (1, 2), (1, 4), (2, 3), (3, 5), (3, 6)]),
45: (4)                 ]
46: (4)                 def test_self_isomorphism(self):
47: (8)                     """
```

```
48: (8)                     For some small, symmetric graphs, make sure that 1) they are
isomorphic
49: (8)                     to themselves, and 2) that only the identity mapping is found.
50: (8)                     """
51: (8)                     for node_data, edge_data in self.data:
52: (12)                        graph = nx.Graph()
53: (12)                        graph.add_nodes_from(node_data)
54: (12)                        graph.add_edges_from(edge_data)
55: (12)                        ismags = iso.ISMAGS(
56: (16)                            graph, graph, node_match=iso.categorical_node_match("name",
None)
57: (12)                        )
58: (12)                        assert ismags.is_isomorphic()
59: (12)                        assert ismags.subgraph_is_isomorphic()
60: (12)                        assert list(ismags.subgraph_isomorphisms_iter(symmetry=True)) == [
61: (16)                            {n: n for n in graph.nodes}
62: (12)                        ]
63: (4)                 def test_edgecase_self_isomorphism(self):
64: (8)                     """
65: (8)                     This edgecase is one of the cases in which it is hard to find all
66: (8)                     symmetry elements.
67: (8)                     """
68: (8)                     graph = nx.Graph()
69: (8)                     nx.add_path(graph, range(5))
70: (8)                     graph.add_edges_from([(2, 5), (5, 6)])
71: (8)                     ismags = iso.ISMAGS(graph, graph)
72: (8)                     ismags_answer = list(ismags.find_isomorphisms(True))
73: (8)                     assert ismags_answer == [{n: n for n in graph.nodes}]
74: (8)                     graph = nx.relabel_nodes(graph, {0: 0, 1: 1, 2: 2, 3: 3, 4: 6, 5: 4,
6: 5})
75: (8)                     ismags = iso.ISMAGS(graph, graph)
76: (8)                     ismags_answer = list(ismags.find_isomorphisms(True))
77: (8)                     assert ismags_answer == [{n: n for n in graph.nodes}]
78: (4)                 def test_directed_self_isomorphism(self):
79: (8)                     """
80: (8)                     For some small, directed, symmetric graphs, make sure that 1) they are
81: (8)                     isomorphic to themselves, and 2) that only the identity mapping is
82: (8)                     found.
83: (8)                     """
84: (8)                     for node_data, edge_data in self.data:
85: (12)                        graph = nx.Graph()
86: (12)                        graph.add_nodes_from(node_data)
87: (12)                        graph.add_edges_from(edge_data)
88: (12)                        ismags = iso.ISMAGS(
89: (16)                            graph, graph, node_match=iso.categorical_node_match("name",
None)
90: (12)                        )
91: (12)                        assert ismags.is_isomorphic()
92: (12)                        assert ismags.subgraph_is_isomorphic()
93: (12)                        assert list(ismags.subgraph_isomorphisms_iter(symmetry=True)) == [
94: (16)                            {n: n for n in graph.nodes}
95: (12)                        ]
96: (0)             class TestSubgraphIsomorphism:
97: (4)                 def test_isomorphism(self):
98: (8)                     g1 = nx.Graph()
99: (8)                     nx.add_cycle(g1, range(4))
100: (8)                    g2 = nx.Graph()
101: (8)                    nx.add_cycle(g2, range(4))
102: (8)                    g2.add_edges_from(list(zip(g2, range(4, 8))))
103: (8)                    ismags = iso.ISMAGS(g2, g1)
104: (8)                    assert list(ismags.subgraph_isomorphisms_iter(symmetry=True)) == [
105: (12)                       {n: n for n in g1.nodes}
106: (8)                    ]
107: (4)                def test_isomorphism2(self):
108: (8)                    g1 = nx.Graph()
109: (8)                    nx.add_path(g1, range(3))
110: (8)                    g2 = g1.copy()
111: (8)                    g2.add_edge(1, 3)
112: (8)                    ismags = iso.ISMAGS(g2, g1)
```

```
113: (8)                    matches = ismags.subgraph_isomorphisms_iter(symmetry=True)
114: (8)                    expected_symmetric = [
115: (12)                       {0: 0, 1: 1, 2: 2},
116: (12)                       {0: 0, 1: 1, 3: 2},
117: (12)                       {2: 0, 1: 1, 3: 2},
118: (8)                    ]
119: (8)                    assert _matches_to_sets(matches) ==
_matches_to_sets(expected_symmetric)
120: (8)                    matches = ismags.subgraph_isomorphisms_iter(symmetry=False)
121: (8)                    expected_asymmetric = [
122: (12)                       {0: 2, 1: 1, 2: 0},
123: (12)                       {0: 2, 1: 1, 3: 0},
124: (12)                       {2: 2, 1: 1, 3: 0},
125: (8)                    ]
126: (8)                    assert _matches_to_sets(matches) == _matches_to_sets(
127: (12)                       expected_symmetric + expected_asymmetric
128: (8)                    )
129: (4)                def test_labeled_nodes(self):
130: (8)                    g1 = nx.Graph()
131: (8)                    nx.add_cycle(g1, range(3))
132: (8)                    g1.nodes[1]["attr"] = True
133: (8)                    g2 = g1.copy()
134: (8)                    g2.add_edge(1, 3)
135: (8)                    ismags = iso.ISMAGS(g2, g1, node_match=lambda x, y: x == y)
136: (8)                    matches = ismags.subgraph_isomorphisms_iter(symmetry=True)
137: (8)                    expected_symmetric = [{0: 0, 1: 1, 2: 2}]
138: (8)                    assert _matches_to_sets(matches) ==
_matches_to_sets(expected_symmetric)
139: (8)                    matches = ismags.subgraph_isomorphisms_iter(symmetry=False)
140: (8)                    expected_asymmetric = [{0: 2, 1: 1, 2: 0}]
141: (8)                    assert _matches_to_sets(matches) == _matches_to_sets(
142: (12)                       expected_symmetric + expected_asymmetric
143: (8)                    )
144: (4)                def test_labeled_edges(self):
145: (8)                    g1 = nx.Graph()
146: (8)                    nx.add_cycle(g1, range(3))
147: (8)                    g1.edges[1, 2]["attr"] = True
148: (8)                    g2 = g1.copy()
149: (8)                    g2.add_edge(1, 3)
150: (8)                    ismags = iso.ISMAGS(g2, g1, edge_match=lambda x, y: x == y)
151: (8)                    matches = ismags.subgraph_isomorphisms_iter(symmetry=True)
152: (8)                    expected_symmetric = [{0: 0, 1: 1, 2: 2}]
153: (8)                    assert _matches_to_sets(matches) ==
_matches_to_sets(expected_symmetric)
154: (8)                    matches = ismags.subgraph_isomorphisms_iter(symmetry=False)
155: (8)                    expected_asymmetric = [{1: 2, 0: 0, 2: 1}]
156: (8)                    assert _matches_to_sets(matches) == _matches_to_sets(
157: (12)                       expected_symmetric + expected_asymmetric
158: (8)                    )
159: (0)            class TestWikipediaExample:
160: (4)                g1edges = [
161: (8)                    ["a", "g"],
162: (8)                    ["a", "h"],
163: (8)                    ["a", "i"],
164: (8)                    ["b", "g"],
165: (8)                    ["b", "h"],
166: (8)                    ["b", "j"],
167: (8)                    ["c", "g"],
168: (8)                    ["c", "i"],
169: (8)                    ["c", "j"],
170: (8)                    ["d", "h"],
171: (8)                    ["d", "i"],
172: (8)                    ["d", "j"],
173: (4)                ]
174: (4)                g2edges = [
175: (8)                    [1, 2],
176: (8)                    [2, 3],
177: (8)                    [3, 4],
178: (8)                    [4, 1],
```

```
179: (8)                    [5, 6],
180: (8)                    [6, 7],
181: (8)                    [7, 8],
182: (8)                    [8, 5],
183: (8)                    [1, 5],
184: (8)                    [2, 6],
185: (8)                    [3, 7],
186: (8)                    [4, 8],
187: (4)                ]
188: (4)                def test_graph(self):
189: (8)                    g1 = nx.Graph()
190: (8)                    g2 = nx.Graph()
191: (8)                    g1.add_edges_from(self.g1edges)
192: (8)                    g2.add_edges_from(self.g2edges)
193: (8)                    gm = iso.ISMAGS(g1, g2)
194: (8)                    assert gm.is_isomorphic()
195: (0)            class TestLargestCommonSubgraph:
196: (4)                def test_mcis(self):
197: (8)                    graph1 = nx.Graph()
198: (8)                    graph1.add_edges_from([(1, 2), (2, 3), (2, 4), (3, 4), (4, 5)])
199: (8)                    graph1.nodes[1]["color"] = 0
200: (8)                    graph2 = nx.Graph()
201: (8)                    graph2.add_edges_from(
202: (12)                       [(1, 2), (2, 3), (2, 4), (3, 4), (3, 5), (5, 6), (5, 7), (6, 7)]
203: (8)                    )
204: (8)                    graph2.nodes[1]["color"] = 1
205: (8)                    graph2.nodes[6]["color"] = 2
206: (8)                    graph2.nodes[7]["color"] = 2
207: (8)                    ismags = iso.ISMAGS(
208: (12)                       graph1, graph2, node_match=iso.categorical_node_match("color",
None)
209: (8)                    )
210: (8)                    assert list(ismags.subgraph_isomorphisms_iter(True)) == []
211: (8)                    assert list(ismags.subgraph_isomorphisms_iter(False)) == []
212: (8)                    found_mcis = _matches_to_sets(ismags.largest_common_subgraph())
213: (8)                    expected = _matches_to_sets(
214: (12)                       [{2: 2, 3: 4, 4: 3, 5: 5}, {2: 4, 3: 2, 4: 3, 5: 5}]
215: (8)                    )
216: (8)                    assert expected == found_mcis
217: (8)                    ismags = iso.ISMAGS(
218: (12)                       graph2, graph1, node_match=iso.categorical_node_match("color",
None)
219: (8)                    )
220: (8)                    assert list(ismags.subgraph_isomorphisms_iter(True)) == []
221: (8)                    assert list(ismags.subgraph_isomorphisms_iter(False)) == []
222: (8)                    found_mcis = _matches_to_sets(ismags.largest_common_subgraph())
223: (8)                    expected = _matches_to_sets(
224: (12)                       [{2: 2, 3: 4, 4: 3, 5: 5}, {4: 2, 2: 3, 3: 4, 5: 5}]
225: (8)                    )
226: (8)                    assert expected == found_mcis
227: (4)                def test_symmetry_mcis(self):
228: (8)                    graph1 = nx.Graph()
229: (8)                    nx.add_path(graph1, range(4))
230: (8)                    graph2 = nx.Graph()
231: (8)                    nx.add_path(graph2, range(3))
232: (8)                    graph2.add_edge(1, 3)
233: (8)                    ismags1 = iso.ISMAGS(
234: (12)                       graph1, graph2, node_match=iso.categorical_node_match("color",
None)
235: (8)                    )
236: (8)                    assert list(ismags1.subgraph_isomorphisms_iter(True)) == []
237: (8)                    found_mcis = _matches_to_sets(ismags1.largest_common_subgraph())
238: (8)                    expected = _matches_to_sets([{0: 0, 1: 1, 2: 2}, {1: 0, 3: 2, 2: 1}])
239: (8)                    assert expected == found_mcis
240: (8)                    ismags2 = iso.ISMAGS(
241: (12)                       graph2, graph1, node_match=iso.categorical_node_match("color",
None)
242: (8)                    )
243: (8)                    assert list(ismags2.subgraph_isomorphisms_iter(True)) == []
```

```
244: (8)                    found_mcis = _matches_to_sets(ismags2.largest_common_subgraph())
245: (8)                    expected = _matches_to_sets(
246: (12)                       [
247: (16)                           {3: 2, 0: 0, 1: 1},
248: (16)                           {2: 0, 0: 2, 1: 1},
249: (16)                           {3: 0, 0: 2, 1: 1},
250: (16)                           {3: 0, 1: 1, 2: 2},
251: (16)                           {0: 0, 1: 1, 2: 2},
252: (16)                           {2: 0, 3: 2, 1: 1},
253: (12)                       ]
254: (8)                    )
255: (8)                    assert expected == found_mcis
256: (8)                    found_mcis1 = _matches_to_sets(ismags1.largest_common_subgraph(False))
257: (8)                    found_mcis2 = ismags2.largest_common_subgraph(False)
258: (8)                    found_mcis2 = [{v: k for k, v in d.items()} for d in found_mcis2]
259: (8)                    found_mcis2 = _matches_to_sets(found_mcis2)
260: (8)                    expected = _matches_to_sets(
261: (12)                       [
262: (16)                           {3: 2, 1: 3, 2: 1},
263: (16)                           {2: 0, 0: 2, 1: 1},
264: (16)                           {1: 2, 3: 3, 2: 1},
265: (16)                           {3: 0, 1: 3, 2: 1},
266: (16)                           {0: 2, 2: 3, 1: 1},
267: (16)                           {3: 0, 1: 2, 2: 1},
268: (16)                           {2: 0, 0: 3, 1: 1},
269: (16)                           {0: 0, 2: 3, 1: 1},
270: (16)                           {1: 0, 3: 3, 2: 1},
271: (16)                           {1: 0, 3: 2, 2: 1},
272: (16)                           {0: 3, 1: 1, 2: 2},
273: (16)                           {0: 0, 1: 1, 2: 2},
274: (12)                       ]
275: (8)                    )
276: (8)                    assert expected == found_mcis1
277: (8)                    assert expected == found_mcis2
```

File 265 - test_isomorphism.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import isomorphism as iso
4: (0)              class TestIsomorph:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      cls.G1 = nx.Graph()
8: (8)                      cls.G2 = nx.Graph()
9: (8)                      cls.G3 = nx.Graph()
10: (8)                     cls.G4 = nx.Graph()
11: (8)                     cls.G5 = nx.Graph()
12: (8)                     cls.G6 = nx.Graph()
13: (8)                     cls.G1.add_edges_from([[1, 2], [1, 3], [1, 5], [2, 3]])
14: (8)                     cls.G2.add_edges_from([[10, 20], [20, 30], [10, 30], [10, 50]])
15: (8)                     cls.G3.add_edges_from([[1, 2], [1, 3], [1, 5], [2, 5]])
16: (8)                     cls.G4.add_edges_from([[1, 2], [1, 3], [1, 5], [2, 4]])
17: (8)                     cls.G5.add_edges_from([[1, 2], [1, 3]])
18: (8)                     cls.G6.add_edges_from([[10, 20], [20, 30], [10, 30], [10, 50], [20,
50]])
19: (4)                 def test_could_be_isomorphic(self):
20: (8)                     assert iso.could_be_isomorphic(self.G1, self.G2)
21: (8)                     assert iso.could_be_isomorphic(self.G1, self.G3)
22: (8)                     assert not iso.could_be_isomorphic(self.G1, self.G4)
23: (8)                     assert iso.could_be_isomorphic(self.G3, self.G2)
24: (8)                     assert not iso.could_be_isomorphic(self.G1, self.G6)
25: (4)                 def test_fast_could_be_isomorphic(self):
26: (8)                     assert iso.fast_could_be_isomorphic(self.G3, self.G2)
27: (8)                     assert not iso.fast_could_be_isomorphic(self.G3, self.G5)
28: (8)                     assert not iso.fast_could_be_isomorphic(self.G1, self.G6)
29: (4)                 def test_faster_could_be_isomorphic(self):
```

```
30: (8)                     assert iso.faster_could_be_isomorphic(self.G3, self.G2)
31: (8)                     assert not iso.faster_could_be_isomorphic(self.G3, self.G5)
32: (8)                     assert not iso.faster_could_be_isomorphic(self.G1, self.G6)
33: (4)                 def test_is_isomorphic(self):
34: (8)                     assert iso.is_isomorphic(self.G1, self.G2)
35: (8)                     assert not iso.is_isomorphic(self.G1, self.G4)
36: (8)                     assert iso.is_isomorphic(self.G1.to_directed(), self.G2.to_directed())
37: (8)                     assert not iso.is_isomorphic(self.G1.to_directed(),
self.G4.to_directed())
38: (8)                     with pytest.raises(
39: (12)                        nx.NetworkXError, match="Graphs G1 and G2 are not of the same
type."
40: (8)                     ):
41: (12)                        iso.is_isomorphic(self.G1.to_directed(), self.G1)
```

File 266 - test_isomorphvf2.py:

```
1: (0)              """
2: (4)                  Tests for VF2 isomorphism algorithm.
3: (0)              """
4: (0)              import importlib.resources
5: (0)              import os
6: (0)              import random
7: (0)              import struct
8: (0)              import networkx as nx
9: (0)              from networkx.algorithms import isomorphism as iso
10: (0)             class TestWikipediaExample:
11: (4)                 g1edges = [
12: (8)                     ["a", "g"],
13: (8)                     ["a", "h"],
14: (8)                     ["a", "i"],
15: (8)                     ["b", "g"],
16: (8)                     ["b", "h"],
17: (8)                     ["b", "j"],
18: (8)                     ["c", "g"],
19: (8)                     ["c", "i"],
20: (8)                     ["c", "j"],
21: (8)                     ["d", "h"],
22: (8)                     ["d", "i"],
23: (8)                     ["d", "j"],
24: (4)                 ]
25: (4)                 g2edges = [
26: (8)                     [1, 2],
27: (8)                     [2, 3],
28: (8)                     [3, 4],
29: (8)                     [4, 1],
30: (8)                     [5, 6],
31: (8)                     [6, 7],
32: (8)                     [7, 8],
33: (8)                     [8, 5],
34: (8)                     [1, 5],
35: (8)                     [2, 6],
36: (8)                     [3, 7],
37: (8)                     [4, 8],
38: (4)                 ]
39: (4)                 def test_graph(self):
40: (8)                     g1 = nx.Graph()
41: (8)                     g2 = nx.Graph()
42: (8)                     g1.add_edges_from(self.g1edges)
43: (8)                     g2.add_edges_from(self.g2edges)
44: (8)                     gm = iso.GraphMatcher(g1, g2)
45: (8)                     assert gm.is_isomorphic()
46: (8)                     assert gm.subgraph_is_monomorphic()
47: (8)                     mapping = sorted(gm.mapping.items())
48: (4)                 def test_subgraph(self):
49: (8)                     g1 = nx.Graph()
50: (8)                     g2 = nx.Graph()
```

```
51: (8)                     g1.add_edges_from(self.g1edges)
52: (8)                     g2.add_edges_from(self.g2edges)
53: (8)                     g3 = g2.subgraph([1, 2, 3, 4])
54: (8)                     gm = iso.GraphMatcher(g1, g3)
55: (8)                     assert gm.subgraph_is_isomorphic()
56: (4)                 def test_subgraph_mono(self):
57: (8)                     g1 = nx.Graph()
58: (8)                     g2 = nx.Graph()
59: (8)                     g1.add_edges_from(self.g1edges)
60: (8)                     g2.add_edges_from([[1, 2], [2, 3], [3, 4]])
61: (8)                     gm = iso.GraphMatcher(g1, g2)
62: (8)                     assert gm.subgraph_is_monomorphic()
63: (0)             class TestVF2GraphDB:
64: (4)                 @staticmethod
65: (4)                 def create_graph(filename):
66: (8)                     """Creates a Graph instance from the filename."""
67: (8)                     fh = open(filename, mode="rb")
68: (8)                     nodes = struct.unpack("<H", fh.read(2))[0]
69: (8)                     graph = nx.Graph()
70: (8)                     for from_node in range(nodes):
71: (12)                        edges = struct.unpack("<H", fh.read(2))[0]
72: (12)                        for edge in range(edges):
73: (16)                            to_node = struct.unpack("<H", fh.read(2))[0]
74: (16)                            graph.add_edge(from_node, to_node)
75: (8)                     fh.close()
76: (8)                     return graph
77: (4)                 def test_graph(self):
78: (8)                     head =
importlib.resources.files("networkx.algorithms.isomorphism.tests")
79: (8)                     g1 = self.create_graph(head / "iso_r01_s80.A99")
80: (8)                     g2 = self.create_graph(head / "iso_r01_s80.B99")
81: (8)                     gm = iso.GraphMatcher(g1, g2)
82: (8)                     assert gm.is_isomorphic()
83: (4)                 def test_subgraph(self):
84: (8)                     head =
importlib.resources.files("networkx.algorithms.isomorphism.tests")
85: (8)                     subgraph = self.create_graph(head / "si2_b06_m200.A99")
86: (8)                     graph = self.create_graph(head / "si2_b06_m200.B99")
87: (8)                     gm = iso.GraphMatcher(graph, subgraph)
88: (8)                     assert gm.subgraph_is_isomorphic()
89: (8)                     assert gm.subgraph_is_monomorphic()
90: (0)             class TestAtlas:
91: (4)                 @classmethod
92: (4)                 def setup_class(cls):
93: (8)                     global atlas
94: (8)                     from networkx.generators import atlas
95: (8)                     cls.GAG = atlas.graph_atlas_g()
96: (4)                 def test_graph_atlas(self):
97: (8)                     Atlas = self.GAG[0:100]
98: (8)                     alphabet = list(range(26))
99: (8)                     for graph in Atlas:
100: (12)                       nlist = list(graph)
101: (12)                       labels = alphabet[: len(nlist)]
102: (12)                       for s in range(10):
103: (16)                           random.shuffle(labels)
104: (16)                           d = dict(zip(nlist, labels))
105: (16)                           relabel = nx.relabel_nodes(graph, d)
106: (16)                           gm = iso.GraphMatcher(graph, relabel)
107: (16)                           assert gm.is_isomorphic()
108: (0)            def test_multiedge():
109: (4)                edges = [
110: (8)                    (0, 1),
111: (8)                    (1, 2),
112: (8)                    (2, 3),
113: (8)                    (3, 4),
114: (8)                    (4, 5),
115: (8)                    (5, 6),
116: (8)                    (6, 7),
117: (8)                    (7, 8),
```

```
(8)                     (8, 9),
(8)                     (9, 10),
(8)                     (10, 11),
(8)                     (10, 11),
(8)                     (11, 12),
(8)                     (11, 12),
(8)                     (12, 13),
(8)                     (12, 13),
(8)                     (13, 14),
(8)                     (13, 14),
(8)                     (14, 15),
(8)                     (14, 15),
(8)                     (15, 16),
(8)                     (15, 16),
(8)                     (16, 17),
(8)                     (16, 17),
(8)                     (17, 18),
(8)                     (17, 18),
(8)                     (18, 19),
(8)                     (18, 19),
(8)                     (19, 0),
(8)                     (19, 0),
(4)                 ]
(4)                 nodes = list(range(20))
(4)                 for g1 in [nx.MultiGraph(), nx.MultiDiGraph()]:
(8)                     g1.add_edges_from(edges)
(8)                     for _ in range(10):
(12)                        new_nodes = list(nodes)
(12)                        random.shuffle(new_nodes)
(12)                        d = dict(zip(nodes, new_nodes))
(12)                        g2 = nx.relabel_nodes(g1, d)
(12)                        if not g1.is_directed():
(16)                            gm = iso.GraphMatcher(g1, g2)
(12)                        else:
(16)                            gm = iso.DiGraphMatcher(g1, g2)
(12)                        assert gm.is_isomorphic()
(12)                        assert gm.subgraph_is_monomorphic()
(0)             def test_selfloop():
(4)                 edges = [
(8)                     (0, 1),
(8)                     (0, 2),
(8)                     (1, 2),
(8)                     (1, 3),
(8)                     (2, 2),
(8)                     (2, 4),
(8)                     (3, 1),
(8)                     (3, 2),
(8)                     (4, 2),
(8)                     (4, 5),
(8)                     (5, 4),
(4)                 ]
(4)                 nodes = list(range(6))
(4)                 for g1 in [nx.Graph(), nx.DiGraph()]:
(8)                     g1.add_edges_from(edges)
(8)                     for _ in range(100):
(12)                        new_nodes = list(nodes)
(12)                        random.shuffle(new_nodes)
(12)                        d = dict(zip(nodes, new_nodes))
(12)                        g2 = nx.relabel_nodes(g1, d)
(12)                        if not g1.is_directed():
(16)                            gm = iso.GraphMatcher(g1, g2)
(12)                        else:
(16)                            gm = iso.DiGraphMatcher(g1, g2)
(12)                        assert gm.is_isomorphic()
(0)             def test_selfloop_mono():
(4)                 edges0 = [
(8)                     (0, 1),
(8)                     (0, 2),
(8)                     (1, 2),
```

```
(8)                     (1, 3),
(8)                     (2, 4),
(8)                     (3, 1),
(8)                     (3, 2),
(8)                     (4, 2),
(8)                     (4, 5),
(8)                     (5, 4),
(4)                 ]
(4)                 edges = edges0 + [(2, 2)]
(4)                 nodes = list(range(6))
(4)                 for g1 in [nx.Graph(), nx.DiGraph()]:
(8)                     g1.add_edges_from(edges)
(8)                     for _ in range(100):
(12)                        new_nodes = list(nodes)
(12)                        random.shuffle(new_nodes)
(12)                        d = dict(zip(nodes, new_nodes))
(12)                        g2 = nx.relabel_nodes(g1, d)
(12)                        g2.remove_edges_from(nx.selfloop_edges(g2))
(12)                        if not g1.is_directed():
(16)                            gm = iso.GraphMatcher(g2, g1)
(12)                        else:
(16)                            gm = iso.DiGraphMatcher(g2, g1)
(12)                        assert not gm.subgraph_is_monomorphic()
(0)             def test_isomorphism_iter1():
(4)                 g1 = nx.DiGraph()
(4)                 g2 = nx.DiGraph()
(4)                 g3 = nx.DiGraph()
(4)                 g1.add_edge("A", "B")
(4)                 g1.add_edge("B", "C")
(4)                 g2.add_edge("Y", "Z")
(4)                 g3.add_edge("Z", "Y")
(4)                 gm12 = iso.DiGraphMatcher(g1, g2)
(4)                 gm13 = iso.DiGraphMatcher(g1, g3)
(4)                 x = list(gm12.subgraph_isomorphisms_iter())
(4)                 y = list(gm13.subgraph_isomorphisms_iter())
(4)                 assert {"A": "Y", "B": "Z"} in x
(4)                 assert {"B": "Y", "C": "Z"} in x
(4)                 assert {"A": "Z", "B": "Y"} in y
(4)                 assert {"B": "Z", "C": "Y"} in y
(4)                 assert len(x) == len(y)
(4)                 assert len(x) == 2
(0)             def test_monomorphism_iter1():
(4)                 g1 = nx.DiGraph()
(4)                 g2 = nx.DiGraph()
(4)                 g1.add_edge("A", "B")
(4)                 g1.add_edge("B", "C")
(4)                 g1.add_edge("C", "A")
(4)                 g2.add_edge("X", "Y")
(4)                 g2.add_edge("Y", "Z")
(4)                 gm12 = iso.DiGraphMatcher(g1, g2)
(4)                 x = list(gm12.subgraph_monomorphisms_iter())
(4)                 assert {"A": "X", "B": "Y", "C": "Z"} in x
(4)                 assert {"A": "Y", "B": "Z", "C": "X"} in x
(4)                 assert {"A": "Z", "B": "X", "C": "Y"} in x
(4)                 assert len(x) == 3
(4)                 gm21 = iso.DiGraphMatcher(g2, g1)
(4)                 assert not gm21.subgraph_is_monomorphic()
(0)             def test_isomorphism_iter2():
(4)                 for L in range(2, 10):
(8)                     g1 = nx.path_graph(L)
(8)                     gm = iso.GraphMatcher(g1, g1)
(8)                     s = len(list(gm.isomorphisms_iter()))
(8)                     assert s == 2
(4)                 for L in range(3, 10):
(8)                     g1 = nx.cycle_graph(L)
(8)                     gm = iso.GraphMatcher(g1, g1)
(8)                     s = len(list(gm.isomorphisms_iter()))
(8)                     assert s == 2 * L
(0)             def test_multiple():
```

```
256: (4)                edges = [("A", "B"), ("B", "A"), ("B", "C")]
257: (4)                for g1, g2 in [(nx.Graph(), nx.Graph()), (nx.DiGraph(), nx.DiGraph())]:
258: (8)                    g1.add_edges_from(edges)
259: (8)                    g2.add_edges_from(edges)
260: (8)                    g3 = nx.subgraph(g2, ["A", "B"])
261: (8)                    if not g1.is_directed():
262: (12)                       gmA = iso.GraphMatcher(g1, g2)
263: (12)                       gmB = iso.GraphMatcher(g1, g3)
264: (8)                    else:
265: (12)                       gmA = iso.DiGraphMatcher(g1, g2)
266: (12)                       gmB = iso.DiGraphMatcher(g1, g3)
267: (8)                    assert gmA.is_isomorphic()
268: (8)                    g2.remove_node("C")
269: (8)                    if not g1.is_directed():
270: (12)                       gmA = iso.GraphMatcher(g1, g2)
271: (8)                    else:
272: (12)                       gmA = iso.DiGraphMatcher(g1, g2)
273: (8)                    assert gmA.subgraph_is_isomorphic()
274: (8)                    assert gmB.subgraph_is_isomorphic()
275: (8)                    assert gmA.subgraph_is_monomorphic()
276: (8)                    assert gmB.subgraph_is_monomorphic()
277: (0)            def test_noncomparable_nodes():
278: (4)                node1 = object()
279: (4)                node2 = object()
280: (4)                node3 = object()
281: (4)                G = nx.path_graph([node1, node2, node3])
282: (4)                gm = iso.GraphMatcher(G, G)
283: (4)                assert gm.is_isomorphic()
284: (4)                assert gm.subgraph_is_monomorphic()
285: (4)                G = nx.path_graph([node1, node2, node3], create_using=nx.DiGraph)
286: (4)                H = nx.path_graph([node3, node2, node1], create_using=nx.DiGraph)
287: (4)                dgm = iso.DiGraphMatcher(G, H)
288: (4)                assert dgm.is_isomorphic()
289: (4)                assert gm.subgraph_is_monomorphic()
290: (0)            def test_monomorphism_edge_match():
291: (4)                G = nx.DiGraph()
292: (4)                G.add_node(1)
293: (4)                G.add_node(2)
294: (4)                G.add_edge(1, 2, label="A")
295: (4)                G.add_edge(2, 1, label="B")
296: (4)                G.add_edge(2, 2, label="C")
297: (4)                SG = nx.DiGraph()
298: (4)                SG.add_node(5)
299: (4)                SG.add_node(6)
300: (4)                SG.add_edge(5, 6, label="A")
301: (4)                gm = iso.DiGraphMatcher(G, SG,
edge_match=iso.categorical_edge_match("label", None))
302: (4)                assert gm.subgraph_is_monomorphic()
303: (0)            def test_isomorphvf2pp_multidigraphs():
304: (4)                g = nx.MultiDiGraph({0: [1, 1, 2, 2, 3], 1: [2, 3, 3], 2: [3]})
305: (4)                h = nx.MultiDiGraph({0: [1, 1, 2, 2, 3], 1: [2, 3, 3], 3: [2]})
306: (4)                assert not (nx.vf2pp_is_isomorphic(g, h))

----------------------------------------

File 267 - test_match_helpers.py:

1: (0)              from operator import eq
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import isomorphism as iso
4: (0)              def test_categorical_node_match():
5: (4)                  nm = iso.categorical_node_match(["x", "y", "z"], [None] * 3)
6: (4)                  assert nm({"x": 1, "y": 2, "z": 3}, {"x": 1, "y": 2, "z": 3})
7: (4)                  assert not nm({"x": 1, "y": 2, "z": 2}, {"x": 1, "y": 2, "z": 1})
8: (0)              class TestGenericMultiEdgeMatch:
9: (4)                  def setup_method(self):
10: (8)                     self.G1 = nx.MultiDiGraph()
11: (8)                     self.G2 = nx.MultiDiGraph()
12: (8)                     self.G3 = nx.MultiDiGraph()
```

```
13: (8)                     self.G4 = nx.MultiDiGraph()
14: (8)                     attr_dict1 = {"id": "edge1", "minFlow": 0, "maxFlow": 10}
15: (8)                     attr_dict2 = {"id": "edge2", "minFlow": -3, "maxFlow": 7}
16: (8)                     attr_dict3 = {"id": "edge3", "minFlow": 13, "maxFlow": 117}
17: (8)                     attr_dict4 = {"id": "edge4", "minFlow": 13, "maxFlow": 117}
18: (8)                     attr_dict5 = {"id": "edge5", "minFlow": 8, "maxFlow": 12}
19: (8)                     attr_dict6 = {"id": "edge6", "minFlow": 8, "maxFlow": 12}
20: (8)                     for attr_dict in [
21: (12)                        attr_dict1,
22: (12)                        attr_dict2,
23: (12)                        attr_dict3,
24: (12)                        attr_dict4,
25: (12)                        attr_dict5,
26: (12)                        attr_dict6,
27: (8)                     ]:
28: (12)                        self.G1.add_edge(1, 2, **attr_dict)
29: (8)                     for attr_dict in [
30: (12)                        attr_dict5,
31: (12)                        attr_dict3,
32: (12)                        attr_dict6,
33: (12)                        attr_dict1,
34: (12)                        attr_dict4,
35: (12)                        attr_dict2,
36: (8)                     ]:
37: (12)                        self.G2.add_edge(2, 3, **attr_dict)
38: (8)                     for attr_dict in [attr_dict3, attr_dict5]:
39: (12)                        self.G3.add_edge(3, 4, **attr_dict)
40: (8)                     for attr_dict in [attr_dict6, attr_dict4]:
41: (12)                        self.G4.add_edge(4, 5, **attr_dict)
42: (4)                 def test_generic_multiedge_match(self):
43: (8)                     full_match = iso.generic_multiedge_match(
44: (12)                        ["id", "flowMin", "flowMax"], [None] * 3, [eq] * 3
45: (8)                     )
46: (8)                     flow_match = iso.generic_multiedge_match(
47: (12)                        ["flowMin", "flowMax"], [None] * 2, [eq] * 2
48: (8)                     )
49: (8)                     min_flow_match = iso.generic_multiedge_match("flowMin", None, eq)
50: (8)                     id_match = iso.generic_multiedge_match("id", None, eq)
51: (8)                     assert flow_match(self.G1[1][2], self.G2[2][3])
52: (8)                     assert min_flow_match(self.G1[1][2], self.G2[2][3])
53: (8)                     assert id_match(self.G1[1][2], self.G2[2][3])
54: (8)                     assert full_match(self.G1[1][2], self.G2[2][3])
55: (8)                     assert flow_match(self.G3[3][4], self.G4[4][5])
56: (8)                     assert min_flow_match(self.G3[3][4], self.G4[4][5])
57: (8)                     assert not id_match(self.G3[3][4], self.G4[4][5])
58: (8)                     assert not full_match(self.G3[3][4], self.G4[4][5])

----------------------------------------

File 268 - test_tree_isomorphism.py:

1: (0)              import random
2: (0)              import time
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.isomorphism.tree_isomorphism import (
6: (4)                  rooted_tree_isomorphism,
7: (4)                  tree_isomorphism,
8: (0)              )
9: (0)              from networkx.classes.function import is_directed
10: (0)             @pytest.mark.parametrize("graph_constructor", (nx.DiGraph, nx.MultiGraph))
11: (0)             def
test_tree_isomorphism_raises_on_directed_and_multigraphs(graph_constructor):
12: (4)                 t1 = graph_constructor([(0, 1)])
13: (4)                 t2 = graph_constructor([(1, 2)])
14: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
15: (8)                     nx.isomorphism.tree_isomorphism(t1, t2)
16: (0)             def check_isomorphism(t1, t2, isomorphism):
17: (4)                 mapping = {v2: v1 for (v1, v2) in isomorphism}
```

```
(4)                 d1 = is_directed(t1)
(4)                 d2 = is_directed(t2)
(4)                 assert d1 == d2
(4)                 edges_1 = []
(4)                 for u, v in t1.edges():
(8)                     if d1:
(12)                        edges_1.append((u, v))
(8)                     else:
(12)                        if u < v:
(16)                            edges_1.append((u, v))
(12)                        else:
(16)                            edges_1.append((v, u))
(4)                 edges_2 = []
(4)                 for u, v in t2.edges():
(8)                     u = mapping[u]
(8)                     v = mapping[v]
(8)                     if d2:
(12)                        edges_2.append((u, v))
(8)                     else:
(12)                        if u < v:
(16)                            edges_2.append((u, v))
(12)                        else:
(16)                            edges_2.append((v, u))
(4)                 return sorted(edges_1) == sorted(edges_2)
(0)             def test_hardcoded():
(4)                 print("hardcoded test")
(4)                 edges_1 = [
(8)                     ("a", "b"),
(8)                     ("a", "c"),
(8)                     ("a", "d"),
(8)                     ("b", "e"),
(8)                     ("b", "f"),
(8)                     ("e", "j"),
(8)                     ("e", "k"),
(8)                     ("c", "g"),
(8)                     ("c", "h"),
(8)                     ("g", "m"),
(8)                     ("d", "i"),
(8)                     ("f", "l"),
(4)                 ]
(4)                 edges_2 = [
(8)                     ("v", "y"),
(8)                     ("v", "z"),
(8)                     ("u", "x"),
(8)                     ("q", "u"),
(8)                     ("q", "v"),
(8)                     ("p", "t"),
(8)                     ("n", "p"),
(8)                     ("n", "q"),
(8)                     ("n", "o"),
(8)                     ("o", "r"),
(8)                     ("o", "s"),
(8)                     ("s", "w"),
(4)                 ]
(4)                 isomorphism1 = [
(8)                     ("a", "n"),
(8)                     ("b", "q"),
(8)                     ("c", "o"),
(8)                     ("d", "p"),
(8)                     ("e", "v"),
(8)                     ("f", "u"),
(8)                     ("g", "s"),
(8)                     ("h", "r"),
(8)                     ("i", "t"),
(8)                     ("j", "y"),
(8)                     ("k", "z"),
(8)                     ("l", "x"),
(8)                     ("m", "w"),
(4)                 ]
```

```
(4)                 isomorphism2 = [
(8)                     ("a", "n"),
(8)                     ("b", "q"),
(8)                     ("c", "o"),
(8)                     ("d", "p"),
(8)                     ("e", "v"),
(8)                     ("f", "u"),
(8)                     ("g", "s"),
(8)                     ("h", "r"),
(8)                     ("i", "t"),
(8)                     ("j", "z"),
(8)                     ("k", "y"),
(8)                     ("l", "x"),
(8)                     ("m", "w"),
(4)                 ]
(4)                 t1 = nx.Graph()
(4)                 t1.add_edges_from(edges_1)
(4)                 root1 = "a"
(4)                 t2 = nx.Graph()
(4)                 t2.add_edges_from(edges_2)
(4)                 root2 = "n"
(4)                 isomorphism = sorted(rooted_tree_isomorphism(t1, root1, t2, root2))
(4)                 assert isomorphism in (isomorphism1, isomorphism2)
(4)                 assert check_isomorphism(t1, t2, isomorphism)
(4)                 t1 = nx.DiGraph()
(4)                 t1.add_edges_from(edges_1)
(4)                 root1 = "a"
(4)                 t2 = nx.DiGraph()
(4)                 t2.add_edges_from(edges_2)
(4)                 root2 = "n"
(4)                 isomorphism = sorted(rooted_tree_isomorphism(t1, root1, t2, root2))
(4)                 assert isomorphism in (isomorphism1, isomorphism2)
(4)                 assert check_isomorphism(t1, t2, isomorphism)
(0)             def random_swap(t):
(4)                 (a, b) = t
(4)                 if random.randint(0, 1) == 1:
(8)                     return (a, b)
(4)                 else:
(8)                     return (b, a)
(0)             def positive_single_tree(t1):
(4)                 assert nx.is_tree(t1)
(4)                 nodes1 = list(t1.nodes())
(4)                 nodes2 = nodes1.copy()
(4)                 random.shuffle(nodes2)
(4)                 someisomorphism = list(zip(nodes1, nodes2))
(4)                 map1to2 = dict(someisomorphism)
(4)                 edges2 = [random_swap((map1to2[u], map1to2[v])) for (u, v) in t1.edges()]
(4)                 random.shuffle(edges2)
(4)                 t2 = nx.Graph()
(4)                 t2.add_edges_from(edges2)
(4)                 isomorphism = tree_isomorphism(t1, t2)
(4)                 assert len(isomorphism) > 0
(4)                 assert check_isomorphism(t1, t2, isomorphism)
(0)             def test_positive(maxk=14):
(4)                 print("positive test")
(4)                 for k in range(2, maxk + 1):
(8)                     start_time = time.time()
(8)                     trial = 0
(8)                     for t in nx.nonisomorphic_trees(k):
(12)                        positive_single_tree(t)
(12)                        trial += 1
(8)                     print(k, trial, time.time() - start_time)
(0)             def test_trivial():
(4)                 print("trivial test")
(4)                 t1 = nx.Graph()
(4)                 t1.add_node("a")
(4)                 root1 = "a"
(4)                 t2 = nx.Graph()
(4)                 t2.add_node("n")
```

```
156: (4)                root2 = "n"
157: (4)                isomorphism = rooted_tree_isomorphism(t1, root1, t2, root2)
158: (4)                assert isomorphism == [("a", "n")]
159: (4)                assert check_isomorphism(t1, t2, isomorphism)
160: (0)            def test_trivial_2():
161: (4)                print("trivial test 2")
162: (4)                edges_1 = [("a", "b"), ("a", "c")]
163: (4)                edges_2 = [("v", "y")]
164: (4)                t1 = nx.Graph()
165: (4)                t1.add_edges_from(edges_1)
166: (4)                t2 = nx.Graph()
167: (4)                t2.add_edges_from(edges_2)
168: (4)                isomorphism = tree_isomorphism(t1, t2)
169: (4)                assert isomorphism == []
170: (0)            def test_negative(maxk=11):
171: (4)                print("negative test")
172: (4)                for k in range(4, maxk + 1):
173: (8)                    test_trees = list(nx.nonisomorphic_trees(k))
174: (8)                    start_time = time.time()
175: (8)                    trial = 0
176: (8)                    for i in range(len(test_trees) - 1):
177: (12)                       for j in range(i + 1, len(test_trees)):
178: (16)                           trial += 1
179: (16)                           assert tree_isomorphism(test_trees[i], test_trees[j]) == []
180: (8)                    print(k, trial, time.time() - start_time)

----------------------------------------

File 269 - test_temporalisomorphvf2.py:

1: (0)              """
2: (4)                  Tests for the temporal aspect of the Temporal VF2 isomorphism algorithm.
3: (0)              """
4: (0)              from datetime import date, datetime, timedelta
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms import isomorphism as iso
7: (0)              def provide_g1_edgelist():
8: (4)                  return [(0, 1), (0, 2), (1, 2), (2, 4), (1, 3), (3, 4), (4, 5)]
9: (0)              def put_same_time(G, att_name):
10: (4)                 for e in G.edges(data=True):
11: (8)                     e[2][att_name] = date(2015, 1, 1)
12: (4)                 return G
13: (0)             def put_same_datetime(G, att_name):
14: (4)                 for e in G.edges(data=True):
15: (8)                     e[2][att_name] = datetime(2015, 1, 1)
16: (4)                 return G
17: (0)             def put_sequence_time(G, att_name):
18: (4)                 current_date = date(2015, 1, 1)
19: (4)                 for e in G.edges(data=True):
20: (8)                     current_date += timedelta(days=1)
21: (8)                     e[2][att_name] = current_date
22: (4)                 return G
23: (0)             def put_time_config_0(G, att_name):
24: (4)                 G[0][1][att_name] = date(2015, 1, 2)
25: (4)                 G[0][2][att_name] = date(2015, 1, 2)
26: (4)                 G[1][2][att_name] = date(2015, 1, 3)
27: (4)                 G[1][3][att_name] = date(2015, 1, 1)
28: (4)                 G[2][4][att_name] = date(2015, 1, 1)
29: (4)                 G[3][4][att_name] = date(2015, 1, 3)
30: (4)                 G[4][5][att_name] = date(2015, 1, 3)
31: (4)                 return G
32: (0)             def put_time_config_1(G, att_name):
33: (4)                 G[0][1][att_name] = date(2015, 1, 2)
34: (4)                 G[0][2][att_name] = date(2015, 1, 1)
35: (4)                 G[1][2][att_name] = date(2015, 1, 3)
36: (4)                 G[1][3][att_name] = date(2015, 1, 1)
37: (4)                 G[2][4][att_name] = date(2015, 1, 2)
38: (4)                 G[3][4][att_name] = date(2015, 1, 4)
39: (4)                 G[4][5][att_name] = date(2015, 1, 3)
```

```
(4)                 return G
(0)             def put_time_config_2(G, att_name):
(4)                 G[0][1][att_name] = date(2015, 1, 1)
(4)                 G[0][2][att_name] = date(2015, 1, 1)
(4)                 G[1][2][att_name] = date(2015, 1, 3)
(4)                 G[1][3][att_name] = date(2015, 1, 2)
(4)                 G[2][4][att_name] = date(2015, 1, 2)
(4)                 G[3][4][att_name] = date(2015, 1, 3)
(4)                 G[4][5][att_name] = date(2015, 1, 2)
(4)                 return G
(0)             class TestTimeRespectingGraphMatcher:
(4)                 """
(4)                 A test class for the undirected temporal graph matcher.
(4)                 """
(4)                 def provide_g1_topology(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_edges_from(provide_g1_edgelist())
(8)                     return G1
(4)                 def provide_g2_path_3edges(self):
(8)                     G2 = nx.Graph()
(8)                     G2.add_edges_from([(0, 1), (1, 2), (2, 3)])
(8)                     return G2
(4)                 def test_timdelta_zero_timeRespecting_returnsTrue(self):
(8)                     G1 = self.provide_g1_topology()
(8)                     temporal_name = "date"
(8)                     G1 = put_same_time(G1, temporal_name)
(8)                     G2 = self.provide_g2_path_3edges()
(8)                     d = timedelta()
(8)                     gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
(8)                     assert gm.subgraph_is_isomorphic()
(4)                 def test_timdelta_zero_datetime_timeRespecting_returnsTrue(self):
(8)                     G1 = self.provide_g1_topology()
(8)                     temporal_name = "date"
(8)                     G1 = put_same_datetime(G1, temporal_name)
(8)                     G2 = self.provide_g2_path_3edges()
(8)                     d = timedelta()
(8)                     gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
(8)                     assert gm.subgraph_is_isomorphic()
(4)                 def test_attNameStrange_timdelta_zero_timeRespecting_returnsTrue(self):
(8)                     G1 = self.provide_g1_topology()
(8)                     temporal_name = "strange_name"
(8)                     G1 = put_same_time(G1, temporal_name)
(8)                     G2 = self.provide_g2_path_3edges()
(8)                     d = timedelta()
(8)                     gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
(8)                     assert gm.subgraph_is_isomorphic()
(4)                 def test_notTimeRespecting_returnsFalse(self):
(8)                     G1 = self.provide_g1_topology()
(8)                     temporal_name = "date"
(8)                     G1 = put_sequence_time(G1, temporal_name)
(8)                     G2 = self.provide_g2_path_3edges()
(8)                     d = timedelta()
(8)                     gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
(8)                     assert not gm.subgraph_is_isomorphic()
(4)                 def test_timdelta_one_config0_returns_no_embeddings(self):
(8)                     G1 = self.provide_g1_topology()
(8)                     temporal_name = "date"
(8)                     G1 = put_time_config_0(G1, temporal_name)
(8)                     G2 = self.provide_g2_path_3edges()
(8)                     d = timedelta(days=1)
(8)                     gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
(8)                     count_match = len(list(gm.subgraph_isomorphisms_iter()))
(8)                     assert count_match == 0
(4)                 def test_timdelta_one_config1_returns_four_embedding(self):
(8)                     G1 = self.provide_g1_topology()
(8)                     temporal_name = "date"
(8)                     G1 = put_time_config_1(G1, temporal_name)
(8)                     G2 = self.provide_g2_path_3edges()
(8)                     d = timedelta(days=1)
```

```
109: (8)                    gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
110: (8)                    count_match = len(list(gm.subgraph_isomorphisms_iter()))
111: (8)                    assert count_match == 4
112: (4)                def test_timdelta_one_config2_returns_ten_embeddings(self):
113: (8)                    G1 = self.provide_g1_topology()
114: (8)                    temporal_name = "date"
115: (8)                    G1 = put_time_config_2(G1, temporal_name)
116: (8)                    G2 = self.provide_g2_path_3edges()
117: (8)                    d = timedelta(days=1)
118: (8)                    gm = iso.TimeRespectingGraphMatcher(G1, G2, temporal_name, d)
119: (8)                    L = list(gm.subgraph_isomorphisms_iter())
120: (8)                    count_match = len(list(gm.subgraph_isomorphisms_iter()))
121: (8)                    assert count_match == 10
122: (0)            class TestDiTimeRespectingGraphMatcher:
123: (4)                """
124: (4)                A test class for the directed time-respecting graph matcher.
125: (4)                """
126: (4)                def provide_g1_topology(self):
127: (8)                    G1 = nx.DiGraph()
128: (8)                    G1.add_edges_from(provide_g1_edgelist())
129: (8)                    return G1
130: (4)                def provide_g2_path_3edges(self):
131: (8)                    G2 = nx.DiGraph()
132: (8)                    G2.add_edges_from([(0, 1), (1, 2), (2, 3)])
133: (8)                    return G2
134: (4)                def test_timdelta_zero_same_dates_returns_true(self):
135: (8)                    G1 = self.provide_g1_topology()
136: (8)                    temporal_name = "date"
137: (8)                    G1 = put_same_time(G1, temporal_name)
138: (8)                    G2 = self.provide_g2_path_3edges()
139: (8)                    d = timedelta()
140: (8)                    gm = iso.TimeRespectingDiGraphMatcher(G1, G2, temporal_name, d)
141: (8)                    assert gm.subgraph_is_isomorphic()
142: (4)                def test_attNameStrange_timdelta_zero_same_dates_returns_true(self):
143: (8)                    G1 = self.provide_g1_topology()
144: (8)                    temporal_name = "strange"
145: (8)                    G1 = put_same_time(G1, temporal_name)
146: (8)                    G2 = self.provide_g2_path_3edges()
147: (8)                    d = timedelta()
148: (8)                    gm = iso.TimeRespectingDiGraphMatcher(G1, G2, temporal_name, d)
149: (8)                    assert gm.subgraph_is_isomorphic()
150: (4)                def test_timdelta_one_config0_returns_no_embeddings(self):
151: (8)                    G1 = self.provide_g1_topology()
152: (8)                    temporal_name = "date"
153: (8)                    G1 = put_time_config_0(G1, temporal_name)
154: (8)                    G2 = self.provide_g2_path_3edges()
155: (8)                    d = timedelta(days=1)
156: (8)                    gm = iso.TimeRespectingDiGraphMatcher(G1, G2, temporal_name, d)
157: (8)                    count_match = len(list(gm.subgraph_isomorphisms_iter()))
158: (8)                    assert count_match == 0
159: (4)                def test_timdelta_one_config1_returns_one_embedding(self):
160: (8)                    G1 = self.provide_g1_topology()
161: (8)                    temporal_name = "date"
162: (8)                    G1 = put_time_config_1(G1, temporal_name)
163: (8)                    G2 = self.provide_g2_path_3edges()
164: (8)                    d = timedelta(days=1)
165: (8)                    gm = iso.TimeRespectingDiGraphMatcher(G1, G2, temporal_name, d)
166: (8)                    count_match = len(list(gm.subgraph_isomorphisms_iter()))
167: (8)                    assert count_match == 1
168: (4)                def test_timdelta_one_config2_returns_two_embeddings(self):
169: (8)                    G1 = self.provide_g1_topology()
170: (8)                    temporal_name = "date"
171: (8)                    G1 = put_time_config_2(G1, temporal_name)
172: (8)                    G2 = self.provide_g2_path_3edges()
173: (8)                    d = timedelta(days=1)
174: (8)                    gm = iso.TimeRespectingDiGraphMatcher(G1, G2, temporal_name, d)
175: (8)                    count_match = len(list(gm.subgraph_isomorphisms_iter()))
176: (8)                    assert count_match == 2
```

```
----------------------------------------

File 270 - test_vf2pp.py:

import itertools as it
import pytest
import networkx as nx
from networkx import vf2pp_is_isomorphic, vf2pp_isomorphism
labels_same = ["blue"]
labels_many = [
    "white",
    "red",
    "blue",
    "green",
    "orange",
    "black",
    "purple",
    "yellow",
    "brown",
    "cyan",
    "solarized",
    "pink",
    "none",
]
class TestPreCheck:
    def test_first_graph_empty(self):
        G1 = nx.Graph()
        G2 = nx.Graph([(0, 1), (1, 2)])
        assert not vf2pp_is_isomorphic(G1, G2)
    def test_second_graph_empty(self):
        G1 = nx.Graph([(0, 1), (1, 2)])
        G2 = nx.Graph()
        assert not vf2pp_is_isomorphic(G1, G2)
    def test_different_order1(self):
        G1 = nx.path_graph(5)
        G2 = nx.path_graph(6)
        assert not vf2pp_is_isomorphic(G1, G2)
    def test_different_order2(self):
        G1 = nx.barbell_graph(100, 20)
        G2 = nx.barbell_graph(101, 20)
        assert not vf2pp_is_isomorphic(G1, G2)
    def test_different_order3(self):
        G1 = nx.complete_graph(7)
        G2 = nx.complete_graph(8)
        assert not vf2pp_is_isomorphic(G1, G2)
    def test_different_degree_sequences1(self):
        G1 = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (0, 4)])
        G2 = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (0, 4), (2, 5)])
        assert not vf2pp_is_isomorphic(G1, G2)
        G2.remove_node(3)
        nx.set_node_attributes(G1, dict(zip(G1, it.cycle(["a"]))), "label")
        nx.set_node_attributes(G2, dict(zip(G2, it.cycle("a"))), "label")
        assert vf2pp_is_isomorphic(G1, G2)
    def test_different_degree_sequences2(self):
        G1 = nx.Graph(
            [
                (0, 1),
                (1, 2),
                (0, 2),
                (2, 3),
                (3, 4),
                (4, 5),
                (5, 6),
                (6, 3),
                (4, 7),
                (7, 8),
                (8, 3),
            ]
        )
```

```
66: (8)                     G2 = G1.copy()
67: (8)                     G2.add_edge(8, 0)
68: (8)                     assert not vf2pp_is_isomorphic(G1, G2)
69: (8)                     G1.add_edge(6, 1)
70: (8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(["a"]))), "label")
71: (8)                     nx.set_node_attributes(G2, dict(zip(G2, it.cycle("a"))), "label")
72: (8)                     assert vf2pp_is_isomorphic(G1, G2)
73: (4)                 def test_different_degree_sequences3(self):
74: (8)                     G1 = nx.Graph([(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 4), (2, 5),
(2, 6)])
75: (8)                     G2 = nx.Graph(
76: (12)                        [(0, 1), (0, 6), (0, 2), (1, 2), (2, 3), (2, 4), (3, 4), (2, 5),
(2, 6)]
77: (8)                     )
78: (8)                     assert not vf2pp_is_isomorphic(G1, G2)
79: (8)                     G1.add_edge(3, 5)
80: (8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(["a"]))), "label")
81: (8)                     nx.set_node_attributes(G2, dict(zip(G2, it.cycle("a"))), "label")
82: (8)                     assert vf2pp_is_isomorphic(G1, G2)
83: (4)                 def test_label_distribution(self):
84: (8)                     G1 = nx.Graph([(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 4), (2, 5),
(2, 6)])
85: (8)                     G2 = nx.Graph([(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 4), (2, 5),
(2, 6)])
86: (8)                     colors1 = ["blue", "blue", "blue", "yellow", "black", "purple",
"purple"]
87: (8)                     colors2 = ["blue", "blue", "yellow", "yellow", "black", "purple",
"purple"]
88: (8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(colors1[::-1]))),
"label")
89: (8)                     nx.set_node_attributes(G2, dict(zip(G2, it.cycle(colors2[::-1]))),
"label")
90: (8)                     assert not vf2pp_is_isomorphic(G1, G2, node_label="label")
91: (8)                     G2.nodes[3]["label"] = "blue"
92: (8)                     assert vf2pp_is_isomorphic(G1, G2, node_label="label")
93: (0)             class TestAllGraphTypesEdgeCases:
94: (4)                 @pytest.mark.parametrize("graph_type", (nx.Graph, nx.MultiGraph,
nx.DiGraph))
95: (4)                 def test_both_graphs_empty(self, graph_type):
96: (8)                     G = graph_type()
97: (8)                     H = graph_type()
98: (8)                     assert vf2pp_isomorphism(G, H) is None
99: (8)                     G.add_node(0)
100: (8)                    assert vf2pp_isomorphism(G, H) is None
101: (8)                    assert vf2pp_isomorphism(H, G) is None
102: (8)                    H.add_node(0)
103: (8)                    assert vf2pp_isomorphism(G, H) == {0: 0}
104: (4)                @pytest.mark.parametrize("graph_type", (nx.Graph, nx.MultiGraph,
nx.DiGraph))
105: (4)                def test_first_graph_empty(self, graph_type):
106: (8)                    G = graph_type()
107: (8)                    H = graph_type([(0, 1)])
108: (8)                    assert vf2pp_isomorphism(G, H) is None
109: (4)                @pytest.mark.parametrize("graph_type", (nx.Graph, nx.MultiGraph,
nx.DiGraph))
110: (4)                def test_second_graph_empty(self, graph_type):
111: (8)                    G = graph_type([(0, 1)])
112: (8)                    H = graph_type()
113: (8)                    assert vf2pp_isomorphism(G, H) is None
114: (0)            class TestGraphISOVF2pp:
115: (4)                def test_custom_graph1_same_labels(self):
116: (8)                    G1 = nx.Graph()
117: (8)                    mapped = {1: "A", 2: "B", 3: "C", 4: "D", 5: "Z", 6: "E"}
118: (8)                    edges1 = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 6), (3, 4), (5, 1), (5,
2)]
119: (8)                    G1.add_edges_from(edges1)
120: (8)                    G2 = nx.relabel_nodes(G1, mapped)
121: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
```

```
122: (8)                    nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
123: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
124: (8)                    G1.add_edge(3, 7)
125: (8)                    G1.nodes[7]["label"] = "blue"
126: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
127: (8)                    G2.add_edges_from([(mapped[3], "X"), (mapped[6], mapped[5])])
128: (8)                    G1.add_edge(4, 7)
129: (8)                    G2.nodes["X"]["label"] = "blue"
130: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
131: (8)                    G1.remove_edges_from([(1, 4), (1, 3)])
132: (8)                    G2.remove_edges_from([(mapped[1], mapped[5]), (mapped[1], mapped[2])])
133: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
134: (4)                def test_custom_graph1_different_labels(self):
135: (8)                    G1 = nx.Graph()
136: (8)                    mapped = {1: "A", 2: "B", 3: "C", 4: "D", 5: "Z", 6: "E"}
137: (8)                    edges1 = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 6), (3, 4), (5, 1), (5,
2)]
138: (8)                    G1.add_edges_from(edges1)
139: (8)                    G2 = nx.relabel_nodes(G1, mapped)
140: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
141: (8)                    nx.set_node_attributes(
142: (12)                       G2,
143: (12)                       dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
144: (12)                       "label",
145: (8)                    )
146: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
147: (4)                def test_custom_graph2_same_labels(self):
148: (8)                    G1 = nx.Graph()
149: (8)                    mapped = {1: "A", 2: "C", 3: "D", 4: "E", 5: "G", 7: "B", 6: "F"}
150: (8)                    edges1 = [(1, 2), (1, 5), (5, 6), (2, 3), (2, 4), (3, 4), (4, 5), (2,
7)]
151: (8)                    G1.add_edges_from(edges1)
152: (8)                    G2 = nx.relabel_nodes(G1, mapped)
153: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
154: (8)                    nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
155: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
156: (8)                    G2.remove_edge(mapped[1], mapped[2])
157: (8)                    G2.add_edge(mapped[1], mapped[4])
158: (8)                    H1 = nx.Graph(G1.subgraph([2, 3, 4, 7]))
159: (8)                    H2 = nx.Graph(G2.subgraph([mapped[1], mapped[4], mapped[5],
mapped[6]]))
160: (8)                    assert vf2pp_isomorphism(H1, H2, node_label="label")
161: (8)                    H1.add_edges_from([(3, 7), (4, 7)])
162: (8)                    H2.add_edges_from([(mapped[1], mapped[6]), (mapped[4], mapped[6])])
163: (8)                    assert vf2pp_isomorphism(H1, H2, node_label="label")
164: (4)                def test_custom_graph2_different_labels(self):
165: (8)                    G1 = nx.Graph()
166: (8)                    mapped = {1: "A", 2: "C", 3: "D", 4: "E", 5: "G", 7: "B", 6: "F"}
167: (8)                    edges1 = [(1, 2), (1, 5), (5, 6), (2, 3), (2, 4), (3, 4), (4, 5), (2,
7)]
168: (8)                    G1.add_edges_from(edges1)
169: (8)                    G2 = nx.relabel_nodes(G1, mapped)
170: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
171: (8)                    nx.set_node_attributes(
172: (12)                       G2,
173: (12)                       dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
174: (12)                       "label",
175: (8)                    )
176: (8)                    G1.add_node(0)
177: (8)                    G2.add_node("Z")
178: (8)                    G1.nodes[0]["label"] = G1.nodes[1]["label"]
179: (8)                    G2.nodes["Z"]["label"] = G1.nodes[1]["label"]
180: (8)                    mapped.update({0: "Z"})
181: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
```

```
182: (8)                    G2.nodes["Z"]["label"] = G1.nodes[2]["label"]
183: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
184: (8)                    G1.nodes[0]["label"] = "blue"
185: (8)                    G2.nodes["Z"]["label"] = "blue"
186: (8)                    G1.add_edge(0, 1)
187: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
188: (8)                    G2.add_edge("Z", "A")
189: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
190: (4)                def test_custom_graph3_same_labels(self):
191: (8)                    G1 = nx.Graph()
192: (8)                    mapped = {1: 9, 2: 8, 3: 7, 4: 6, 5: 3, 8: 5, 9: 4, 7: 1, 6: 2}
193: (8)                    edges1 = [
194: (12)                       (1, 2),
195: (12)                       (1, 3),
196: (12)                       (2, 3),
197: (12)                       (3, 4),
198: (12)                       (4, 5),
199: (12)                       (4, 7),
200: (12)                       (4, 9),
201: (12)                       (5, 8),
202: (12)                       (8, 9),
203: (12)                       (5, 6),
204: (12)                       (6, 7),
205: (12)                       (5, 2),
206: (8)                    ]
207: (8)                    G1.add_edges_from(edges1)
208: (8)                    G2 = nx.relabel_nodes(G1, mapped)
209: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
210: (8)                    nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
211: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
212: (8)                    G1.add_edges_from([(6, 9), (7, 8)])
213: (8)                    G2.add_edges_from([(mapped[6], mapped[8]), (mapped[7], mapped[9])])
214: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
215: (8)                    G1.add_edges_from([(6, 8), (7, 9)])
216: (8)                    G2.add_edges_from([(mapped[6], mapped[9]), (mapped[7], mapped[8])])
217: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
218: (8)                    G1.add_edges_from([(2, 7), (3, 6)])
219: (8)                    G2.add_edges_from([(mapped[2], mapped[7]), (mapped[3], mapped[6])])
220: (8)                    G1.add_node(10)
221: (8)                    G2.add_node("Z")
222: (8)                    G1.nodes[10]["label"] = "blue"
223: (8)                    G2.nodes["Z"]["label"] = "blue"
224: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
225: (8)                    G1.add_edges_from([(10, 1), (10, 5), (10, 8)])
226: (8)                    G2.add_edges_from([("Z", mapped[1]), ("Z", mapped[4]), ("Z",
mapped[9])])
227: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
228: (8)                    H1 = nx.Graph(G1.subgraph([2, 3, 4, 5, 6, 7, 10]))
229: (8)                    H2 = nx.Graph(
230: (12)                       G2.subgraph(
231: (16)                           [mapped[4], mapped[5], mapped[6], mapped[7], mapped[8],
mapped[9], "Z"]
232: (12)                       )
233: (8)                    )
234: (8)                    assert vf2pp_isomorphism(H1, H2, node_label="label") is None
235: (8)                    H1.add_edges_from([(10, 2), (10, 6), (3, 6), (2, 7), (2, 6), (3, 7)])
236: (8)                    H2.add_edges_from(
237: (12)                       [("Z", mapped[7]), (mapped[6], mapped[9]), (mapped[7], mapped[8])]
238: (8)                    )
239: (8)                    assert vf2pp_isomorphism(H1, H2, node_label="label")
240: (8)                    H1.add_edge(3, 5)
241: (8)                    H2.add_edge(mapped[5], mapped[7])
242: (8)                    assert vf2pp_isomorphism(H1, H2, node_label="label") is None
243: (4)                def test_custom_graph3_different_labels(self):
244: (8)                    G1 = nx.Graph()
245: (8)                    mapped = {1: 9, 2: 8, 3: 7, 4: 6, 5: 3, 8: 5, 9: 4, 7: 1, 6: 2}
246: (8)                    edges1 = [
```

```
247: (12)                        (1, 2),
248: (12)                        (1, 3),
249: (12)                        (2, 3),
250: (12)                        (3, 4),
251: (12)                        (4, 5),
252: (12)                        (4, 7),
253: (12)                        (4, 9),
254: (12)                        (5, 8),
255: (12)                        (8, 9),
256: (12)                        (5, 6),
257: (12)                        (6, 7),
258: (12)                        (5, 2),
259: (8)                    ]
260: (8)                    G1.add_edges_from(edges1)
261: (8)                    G2 = nx.relabel_nodes(G1, mapped)
262: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
263: (8)                    nx.set_node_attributes(
264: (12)                        G2,
265: (12)                        dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
266: (12)                        "label",
267: (8)                    )
268: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
269: (8)                    G1.add_edge(1, 7)
270: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
271: (8)                    G2.add_edge(9, 1)
272: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
273: (8)                    G1.add_node("A")
274: (8)                    G2.add_node("K")
275: (8)                    G1.nodes["A"]["label"] = "green"
276: (8)                    G2.nodes["K"]["label"] = "green"
277: (8)                    mapped.update({"A": "K"})
278: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
279: (8)                    G1.add_edge("A", 6)
280: (8)                    G2.add_edge("K", 5)
281: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
282: (8)                    G1.add_edge(1, 5)
283: (8)                    G1.add_edge(2, 9)
284: (8)                    G2.add_edge(9, 3)
285: (8)                    G2.add_edge(8, 4)
286: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
287: (8)                    for node in G1.nodes():
288: (12)                        color = "red"
289: (12)                        G1.nodes[node]["label"] = color
290: (12)                        G2.nodes[mapped[node]]["label"] = color
291: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
292: (4)                def test_custom_graph4_different_labels(self):
293: (8)                    G1 = nx.Graph()
294: (8)                    edges1 = [
295: (12)                        (1, 2),
296: (12)                        (2, 3),
297: (12)                        (3, 8),
298: (12)                        (3, 4),
299: (12)                        (4, 5),
300: (12)                        (4, 6),
301: (12)                        (3, 6),
302: (12)                        (8, 7),
303: (12)                        (8, 9),
304: (12)                        (5, 9),
305: (12)                        (10, 11),
306: (12)                        (11, 12),
307: (12)                        (12, 13),
308: (12)                        (11, 13),
309: (8)                    ]
310: (8)                    mapped = {
311: (12)                        1: "n",
312: (12)                        2: "m",
313: (12)                        3: "l",
314: (12)                        4: "j",
```

```
(12)                        5: "k",
(12)                        6: "i",
(12)                        7: "g",
(12)                        8: "h",
(12)                        9: "f",
(12)                        10: "b",
(12)                        11: "a",
(12)                        12: "d",
(12)                        13: "e",
(8)                     }
(8)                     G1.add_edges_from(edges1)
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
(8)                     nx.set_node_attributes(
(12)                        G2,
(12)                        dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
(12)                        "label",
(8)                     )
(8)                     assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
(4)                 def test_custom_graph4_same_labels(self):
(8)                     G1 = nx.Graph()
(8)                     edges1 = [
(12)                        (1, 2),
(12)                        (2, 3),
(12)                        (3, 8),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (3, 6),
(12)                        (8, 7),
(12)                        (8, 9),
(12)                        (5, 9),
(12)                        (10, 11),
(12)                        (11, 12),
(12)                        (12, 13),
(12)                        (11, 13),
(8)                     ]
(8)                     mapped = {
(12)                        1: "n",
(12)                        2: "m",
(12)                        3: "l",
(12)                        4: "j",
(12)                        5: "k",
(12)                        6: "i",
(12)                        7: "g",
(12)                        8: "h",
(12)                        9: "f",
(12)                        10: "b",
(12)                        11: "a",
(12)                        12: "d",
(12)                        13: "e",
(8)                     }
(8)                     G1.add_edges_from(edges1)
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
(8)                     nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
(8)                     assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     G1.add_node(0)
(8)                     G2.add_node("z")
(8)                     G1.nodes[0]["label"] = "green"
(8)                     G2.nodes["z"]["label"] = "blue"
(8)                     assert vf2pp_isomorphism(G1, G2, node_label="label") is None
(8)                     G2.nodes["z"]["label"] = "green"
(8)                     assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     G1.add_edge(2, 5)
(8)                     G2.remove_edge("i", "l")
```

```
(8)                 G2.add_edge("g", "l")
(8)                 G2.add_edge("m", "f")
(8)                 assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 G1.remove_node(13)
(8)                 G2.remove_node("d")
(8)                 assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 G1.add_edge(0, 10)
(8)                 G2.add_edge("e", "z")
(8)                 assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 G1.add_edge(11, 3)
(8)                 G1.add_edge(0, 8)
(8)                 G2.add_edge("a", "l")
(8)                 G2.add_edge("z", "j")
(8)                 assert vf2pp_isomorphism(G1, G2, node_label="label")
(4)             def test_custom_graph5_same_labels(self):
(8)                 G1 = nx.Graph()
(8)                 edges1 = [
(12)                    (1, 5),
(12)                    (1, 2),
(12)                    (1, 4),
(12)                    (2, 3),
(12)                    (2, 6),
(12)                    (3, 4),
(12)                    (3, 7),
(12)                    (4, 8),
(12)                    (5, 8),
(12)                    (5, 6),
(12)                    (6, 7),
(12)                    (7, 8),
(8)                 ]
(8)                 mapped = {1: "a", 2: "h", 3: "d", 4: "i", 5: "g", 6: "b", 7: "j", 8:
"c"}
(8)                 G1.add_edges_from(edges1)
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
(8)                 nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
(8)                 assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 G1.add_edges_from([(3, 6), (2, 7), (2, 5), (1, 3), (4, 7), (6, 8)])
(8)                 G2.add_edges_from(
(12)                    [
(16)                        (mapped[6], mapped[3]),
(16)                        (mapped[2], mapped[7]),
(16)                        (mapped[1], mapped[6]),
(16)                        (mapped[5], mapped[7]),
(16)                        (mapped[3], mapped[8]),
(16)                        (mapped[2], mapped[4]),
(12)                    ]
(8)                 )
(8)                 assert vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 H1 = nx.Graph(G1.subgraph([1, 5, 8, 6, 7, 3]))
(8)                 H2 = nx.Graph(
(12)                    G2.subgraph(
(16)                        [mapped[1], mapped[4], mapped[8], mapped[7], mapped[3],
mapped[5]]
(12)                    )
(8)                 )
(8)                 assert vf2pp_isomorphism(H1, H2, node_label="label")
(8)                 H1.remove_node(8)
(8)                 H2.remove_node(mapped[7])
(8)                 assert vf2pp_isomorphism(H1, H2, node_label="label")
(8)                 H1.add_edge(1, 6)
(8)                 H1.remove_edge(3, 6)
(8)                 assert vf2pp_isomorphism(H1, H2, node_label="label")
(4)             def test_custom_graph5_different_labels(self):
(8)                 G1 = nx.Graph()
(8)                 edges1 = [
(12)                    (1, 5),
```

```
446: (12)                        (1, 2),
447: (12)                        (1, 4),
448: (12)                        (2, 3),
449: (12)                        (2, 6),
450: (12)                        (3, 4),
451: (12)                        (3, 7),
452: (12)                        (4, 8),
453: (12)                        (5, 8),
454: (12)                        (5, 6),
455: (12)                        (6, 7),
456: (12)                        (7, 8),
457: (8)                    ]
458: (8)                    mapped = {1: "a", 2: "h", 3: "d", 4: "i", 5: "g", 6: "b", 7: "j", 8:
"c"}
459: (8)                    G1.add_edges_from(edges1)
460: (8)                    G2 = nx.relabel_nodes(G1, mapped)
461: (8)                    colors = ["red", "blue", "grey", "none", "brown", "solarized",
"yellow", "pink"]
462: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
463: (8)                    nx.set_node_attributes(
464: (12)                       G2,
465: (12)                       dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
466: (12)                       "label",
467: (8)                    )
468: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
469: (8)                    c = 0
470: (8)                    for node in G1.nodes():
471: (12)                       color1 = colors[c]
472: (12)                       color2 = colors[(c + 3) % len(colors)]
473: (12)                       G1.nodes[node]["label"] = color1
474: (12)                       G2.nodes[mapped[node]]["label"] = color2
475: (12)                       c += 1
476: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") is None
477: (8)                    H1 = G1.subgraph([1, 5])
478: (8)                    H2 = G2.subgraph(["i", "c"])
479: (8)                    c = 0
480: (8)                    for node1, node2 in zip(H1.nodes(), H2.nodes()):
481: (12)                       H1.nodes[node1]["label"] = "red"
482: (12)                       H2.nodes[node2]["label"] = "red"
483: (12)                       c += 1
484: (8)                    assert vf2pp_isomorphism(H1, H2, node_label="label")
485: (4)                def test_disconnected_graph_all_same_labels(self):
486: (8)                    G1 = nx.Graph()
487: (8)                    G1.add_nodes_from(list(range(10)))
488: (8)                    mapped = {0: 9, 1: 8, 2: 7, 3: 6, 4: 5, 5: 4, 6: 3, 7: 2, 8: 1, 9: 0}
489: (8)                    G2 = nx.relabel_nodes(G1, mapped)
490: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
491: (8)                    nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
492: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
493: (4)                def test_disconnected_graph_all_different_labels(self):
494: (8)                    G1 = nx.Graph()
495: (8)                    G1.add_nodes_from(list(range(10)))
496: (8)                    mapped = {0: 9, 1: 8, 2: 7, 3: 6, 4: 5, 5: 4, 6: 3, 7: 2, 8: 1, 9: 0}
497: (8)                    G2 = nx.relabel_nodes(G1, mapped)
498: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
499: (8)                    nx.set_node_attributes(
500: (12)                       G2,
501: (12)                       dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
502: (12)                       "label",
503: (8)                    )
504: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label") == mapped
505: (4)                def test_disconnected_graph_some_same_labels(self):
506: (8)                    G1 = nx.Graph()
507: (8)                    G1.add_nodes_from(list(range(10)))
508: (8)                    mapped = {0: 9, 1: 8, 2: 7, 3: 6, 4: 5, 5: 4, 6: 3, 7: 2, 8: 1, 9: 0}
```

```
509: (8)                    G2 = nx.relabel_nodes(G1, mapped)
510: (8)                    colors = [
511: (12)                       "white",
512: (12)                       "white",
513: (12)                       "white",
514: (12)                       "purple",
515: (12)                       "purple",
516: (12)                       "red",
517: (12)                       "red",
518: (12)                       "pink",
519: (12)                       "pink",
520: (12)                       "pink",
521: (8)                    ]
522: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(colors))), "label")
523: (8)                    nx.set_node_attributes(
524: (12)                       G2, dict(zip([mapped[n] for n in G1], it.cycle(colors))), "label"
525: (8)                    )
526: (8)                    assert vf2pp_isomorphism(G1, G2, node_label="label")
527: (0)            class TestMultiGraphISOVF2pp:
528: (4)                def test_custom_multigraph1_same_labels(self):
529: (8)                    G1 = nx.MultiGraph()
530: (8)                    mapped = {1: "A", 2: "B", 3: "C", 4: "D", 5: "Z", 6: "E"}
531: (8)                    edges1 = [
532: (12)                       (1, 2),
533: (12)                       (1, 3),
534: (12)                       (1, 4),
535: (12)                       (1, 4),
536: (12)                       (1, 4),
537: (12)                       (2, 3),
538: (12)                       (2, 6),
539: (12)                       (2, 6),
540: (12)                       (3, 4),
541: (12)                       (3, 4),
542: (12)                       (5, 1),
543: (12)                       (5, 1),
544: (12)                       (5, 2),
545: (12)                       (5, 2),
546: (8)                    ]
547: (8)                    G1.add_edges_from(edges1)
548: (8)                    G2 = nx.relabel_nodes(G1, mapped)
549: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
550: (8)                    nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
551: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
552: (8)                    assert m
553: (8)                    G1.remove_edges_from([(2, 6), (2, 6)])
554: (8)                    G1.add_edges_from([(3, 6), (3, 6)])
555: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
556: (8)                    assert not m
557: (8)                    G2.remove_edge(mapped[1], mapped[4])
558: (8)                    G1.remove_edge(1, 4)
559: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
560: (8)                    assert m
561: (8)                    G1.add_edges_from([(5, 5), (5, 5), (1, 1)])
562: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
563: (8)                    assert not m
564: (8)                    G2.add_edges_from(
565: (12)                       [(mapped[1], mapped[1]), (mapped[4], mapped[4]), (mapped[4],
mapped[4])]
566: (8)                    )
567: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
568: (8)                    assert m
569: (4)                def test_custom_multigraph1_different_labels(self):
570: (8)                    G1 = nx.MultiGraph()
571: (8)                    mapped = {1: "A", 2: "B", 3: "C", 4: "D", 5: "Z", 6: "E"}
572: (8)                    edges1 = [
573: (12)                       (1, 2),
574: (12)                       (1, 3),
```

```
575: (12)                        (1, 4),
576: (12)                        (1, 4),
577: (12)                        (1, 4),
578: (12)                        (2, 3),
579: (12)                        (2, 6),
580: (12)                        (2, 6),
581: (12)                        (3, 4),
582: (12)                        (3, 4),
583: (12)                        (5, 1),
584: (12)                        (5, 1),
585: (12)                        (5, 2),
586: (12)                        (5, 2),
587: (8)                    ]
588: (8)                    G1.add_edges_from(edges1)
589: (8)                    G2 = nx.relabel_nodes(G1, mapped)
590: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
591: (8)                    nx.set_node_attributes(
592: (12)                        G2,
593: (12)                        dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
594: (12)                        "label",
595: (8)                    )
596: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
597: (8)                    assert m
598: (8)                    assert m == mapped
599: (8)                    G1.remove_edge(1, 4)
600: (8)                    G1.add_edge(1, 5)
601: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
602: (8)                    assert not m
603: (8)                    G2.remove_edge("A", "D")
604: (8)                    G2.add_edge("A", "Z")
605: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
606: (8)                    assert m
607: (8)                    assert m == mapped
608: (8)                    G1.add_edges_from([(6, 6), (6, 6), (6, 6)])
609: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
610: (8)                    assert not m
611: (8)                    G2.add_edges_from([("E", "E"), ("E", "E"), ("E", "E")])
612: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
613: (8)                    assert m
614: (8)                    assert m == mapped
615: (4)                def test_custom_multigraph2_same_labels(self):
616: (8)                    G1 = nx.MultiGraph()
617: (8)                    mapped = {1: "A", 2: "C", 3: "D", 4: "E", 5: "G", 7: "B", 6: "F"}
618: (8)                    edges1 = [
619: (12)                        (1, 2),
620: (12)                        (1, 2),
621: (12)                        (1, 5),
622: (12)                        (1, 5),
623: (12)                        (1, 5),
624: (12)                        (5, 6),
625: (12)                        (2, 3),
626: (12)                        (2, 3),
627: (12)                        (2, 4),
628: (12)                        (3, 4),
629: (12)                        (3, 4),
630: (12)                        (4, 5),
631: (12)                        (4, 5),
632: (12)                        (4, 5),
633: (12)                        (2, 7),
634: (12)                        (2, 7),
635: (12)                        (2, 7),
636: (8)                    ]
637: (8)                    G1.add_edges_from(edges1)
638: (8)                    G2 = nx.relabel_nodes(G1, mapped)
639: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
640: (8)                    nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
```

```
641: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
642: (8)                    assert m
643: (8)                    G2.remove_edges_from([(mapped[1], mapped[2]), (mapped[1], mapped[2])])
644: (8)                    G2.add_edge(mapped[1], mapped[4])
645: (8)                    H1 = nx.MultiGraph(G1.subgraph([2, 3, 4, 7]))
646: (8)                    H2 = nx.MultiGraph(G2.subgraph([mapped[1], mapped[4], mapped[5],
mapped[6]]))
647: (8)                    m = vf2pp_isomorphism(H1, H2, node_label="label")
648: (8)                    assert not m
649: (8)                    H1.remove_edge(3, 4)
650: (8)                    H1.add_edges_from([(2, 3), (2, 4), (2, 4)])
651: (8)                    H2.add_edges_from([(mapped[5], mapped[6]), (mapped[5], mapped[6])])
652: (8)                    m = vf2pp_isomorphism(H1, H2, node_label="label")
653: (8)                    assert m
654: (8)                    H1.remove_edges_from([(2, 3), (2, 3), (2, 3)])
655: (8)                    H2.remove_edges_from([(mapped[5], mapped[4])] * 3)
656: (8)                    m = vf2pp_isomorphism(H1, H2, node_label="label")
657: (8)                    assert m
658: (8)                    H1.remove_edges_from([(2, 7), (2, 7)])
659: (8)                    H1.add_edges_from([(3, 4), (3, 4)])
660: (8)                    m = vf2pp_isomorphism(H1, H2, node_label="label")
661: (8)                    assert m
662: (8)                    H2.add_edge(mapped[5], mapped[1])
663: (8)                    H1.add_edge(3, 4)
664: (8)                    m = vf2pp_isomorphism(H1, H2, node_label="label")
665: (8)                    assert not m
666: (4)                def test_custom_multigraph2_different_labels(self):
667: (8)                    G1 = nx.MultiGraph()
668: (8)                    mapped = {1: "A", 2: "C", 3: "D", 4: "E", 5: "G", 7: "B", 6: "F"}
669: (8)                    edges1 = [
670: (12)                       (1, 2),
671: (12)                       (1, 2),
672: (12)                       (1, 5),
673: (12)                       (1, 5),
674: (12)                       (1, 5),
675: (12)                       (5, 6),
676: (12)                       (2, 3),
677: (12)                       (2, 3),
678: (12)                       (2, 4),
679: (12)                       (3, 4),
680: (12)                       (3, 4),
681: (12)                       (4, 5),
682: (12)                       (4, 5),
683: (12)                       (4, 5),
684: (12)                       (2, 7),
685: (12)                       (2, 7),
686: (12)                       (2, 7),
687: (8)                    ]
688: (8)                    G1.add_edges_from(edges1)
689: (8)                    G2 = nx.relabel_nodes(G1, mapped)
690: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
691: (8)                    nx.set_node_attributes(
692: (12)                       G2,
693: (12)                       dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
694: (12)                       "label",
695: (8)                    )
696: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
697: (8)                    assert m
698: (8)                    assert m == mapped
699: (8)                    G1.remove_edge(2, 7)
700: (8)                    G1.add_edge(5, 6)
701: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
702: (8)                    assert not m
703: (8)                    G2.remove_edge("B", "C")
704: (8)                    G2.add_edge("G", "F")
705: (8)                    m = vf2pp_isomorphism(G1, G2, node_label="label")
706: (8)                    assert m
707: (8)                    assert m == mapped
```

```
(8)                     G1.remove_node(3)
(8)                     G2.remove_node("D")
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.remove_edge(1, 2)
(8)                     G1.remove_edge(2, 7)
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert not m
(8)                     G2.add_edges_from([("A", "C"), ("C", "E"), ("C", "E")])
(8)                     G2.remove_edges_from(
(12)                        [("A", "G"), ("A", "G"), ("F", "G"), ("E", "G"), ("E", "G")]
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert not m
(8)                     for n1, n2 in zip(G1.nodes(), G2.nodes()):
(12)                        G1.nodes[n1]["label"] = "blue"
(12)                        G2.nodes[n2]["label"] = "blue"
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(4)                 def test_custom_multigraph3_same_labels(self):
(8)                     G1 = nx.MultiGraph()
(8)                     mapped = {1: 9, 2: 8, 3: 7, 4: 6, 5: 3, 8: 5, 9: 4, 7: 1, 6: 2}
(8)                     edges1 = [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (1, 3),
(12)                        (2, 3),
(12)                        (2, 3),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 7),
(12)                        (4, 9),
(12)                        (4, 9),
(12)                        (4, 9),
(12)                        (5, 8),
(12)                        (5, 8),
(12)                        (8, 9),
(12)                        (8, 9),
(12)                        (5, 6),
(12)                        (6, 7),
(12)                        (6, 7),
(12)                        (6, 7),
(12)                        (5, 2),
(8)                     ]
(8)                     G1.add_edges_from(edges1)
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
(8)                     nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.add_edges_from([(6, 9), (7, 8), (5, 8), (4, 9), (4, 9)])
(8)                     G2.add_edges_from(
(12)                        [
(16)                            (mapped[6], mapped[8]),
(16)                            (mapped[7], mapped[9]),
(16)                            (mapped[5], mapped[8]),
(16)                            (mapped[4], mapped[9]),
(16)                            (mapped[4], mapped[9]),
(12)                        ]
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert not m
(8)                     G1.add_edges_from([(6, 8), (6, 8), (7, 9), (7, 9), (7, 9)])
(8)                     G2.add_edges_from(
(12)                        [
(16)                            (mapped[6], mapped[8]),
(16)                            (mapped[6], mapped[9]),
```

```
(16)                            (mapped[7], mapped[8]),
(16)                            (mapped[7], mapped[9]),
(16)                            (mapped[7], mapped[9]),
(12)                        ]
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.add_edges_from([(2, 7), (2, 7), (3, 6), (3, 6)])
(8)                     G2.add_edges_from(
(12)                        [
(16)                            (mapped[2], mapped[7]),
(16)                            (mapped[2], mapped[7]),
(16)                            (mapped[3], mapped[6]),
(16)                            (mapped[3], mapped[6]),
(12)                        ]
(8)                     )
(8)                     G1.add_node(10)
(8)                     G2.add_node("Z")
(8)                     G1.nodes[10]["label"] = "blue"
(8)                     G2.nodes["Z"]["label"] = "blue"
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.add_edges_from([(10, 1), (10, 5), (10, 8), (10, 10), (10, 10)])
(8)                     G2.add_edges_from(
(12)                        [
(16)                            ("Z", mapped[1]),
(16)                            ("Z", mapped[4]),
(16)                            ("Z", mapped[9]),
(16)                            ("Z", "Z"),
(16)                            ("Z", "Z"),
(12)                        ]
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert not m
(8)                     G1.remove_edges_from([(1, 3), (4, 9), (4, 9), (7, 9)])
(8)                     G2.remove_edges_from(
(12)                        [
(16)                            (mapped[1], mapped[3]),
(16)                            (mapped[4], mapped[9]),
(16)                            (mapped[4], mapped[9]),
(16)                            (mapped[7], mapped[9]),
(12)                        ]
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     H1 = nx.Graph(G1.subgraph([2, 3, 4, 5, 6, 7, 10]))
(8)                     H2 = nx.Graph(
(12)                        G2.subgraph(
(16)                            [mapped[4], mapped[5], mapped[6], mapped[7], mapped[8],
mapped[9], "Z"]
(12)                        )
(8)                     )
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert not m
(8)                     H1.add_edges_from([(10, 2), (10, 6), (3, 6), (2, 7), (2, 6), (3, 7)])
(8)                     H2.add_edges_from(
(12)                        [("Z", mapped[7]), (mapped[6], mapped[9]), (mapped[7], mapped[8])]
(8)                     )
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     H2.remove_edge("Z", "Z")
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert not m
(8)                     H1.remove_edge(10, 10)
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(4)                 def test_custom_multigraph3_different_labels(self):
(8)                     G1 = nx.MultiGraph()
(8)                     mapped = {1: 9, 2: 8, 3: 7, 4: 6, 5: 3, 8: 5, 9: 4, 7: 1, 6: 2}
```

```
(8)                     edges1 = [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (1, 3),
(12)                        (2, 3),
(12)                        (2, 3),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 7),
(12)                        (4, 9),
(12)                        (4, 9),
(12)                        (4, 9),
(12)                        (5, 8),
(12)                        (5, 8),
(12)                        (8, 9),
(12)                        (8, 9),
(12)                        (5, 6),
(12)                        (6, 7),
(12)                        (6, 7),
(12)                        (6, 7),
(12)                        (5, 2),
(8)                     ]
(8)                     G1.add_edges_from(edges1)
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
(8)                     nx.set_node_attributes(
(12)                        G2,
(12)                        dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
(12)                        "label",
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     assert m == mapped
(8)                     G1.remove_edge(4, 9)
(8)                     G2.remove_edge(4, 6)
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     assert m == mapped
(8)                     G1.add_edges_from([(4, 9), (1, 2), (1, 2)])
(8)                     G1.remove_edges_from([(1, 3), (1, 3)])
(8)                     G2.add_edges_from([(3, 5), (7, 9)])
(8)                     G2.remove_edge(8, 9)
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert not m
(8)                     for n1, n2 in zip(G1.nodes(), G2.nodes()):
(12)                        G1.nodes[n1]["label"] = "blue"
(12)                        G2.nodes[n2]["label"] = "blue"
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.add_node(10)
(8)                     G2.add_node("Z")
(8)                     G1.nodes[10]["label"] = "green"
(8)                     G2.nodes["Z"]["label"] = "green"
(8)                     G1.add_edges_from([(10, 10), (10, 10)])
(8)                     G2.add_edges_from([("Z", "Z")])
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert not m
(8)                     G1.remove_edge(10, 10)
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.add_edges_from([(10, 3), (10, 4)])
(8)                     G2.add_edges_from([("Z", 8), ("Z", 3)])
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.remove_node(4)
(8)                     G2.remove_node(3)
(8)                     G1.add_edges_from([(5, 6), (5, 6), (5, 7)])
(8)                     G2.add_edges_from([(1, 6), (1, 6), (6, 2)])
```

```
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(4)                 def test_custom_multigraph4_same_labels(self):
(8)                     G1 = nx.MultiGraph()
(8)                     edges1 = [
(12)                        (1, 2),
(12)                        (1, 2),
(12)                        (2, 2),
(12)                        (2, 3),
(12)                        (3, 8),
(12)                        (3, 8),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 5),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (3, 6),
(12)                        (3, 6),
(12)                        (6, 6),
(12)                        (8, 7),
(12)                        (7, 7),
(12)                        (8, 9),
(12)                        (9, 9),
(12)                        (8, 9),
(12)                        (8, 9),
(12)                        (5, 9),
(12)                        (10, 11),
(12)                        (11, 12),
(12)                        (12, 13),
(12)                        (11, 13),
(12)                        (10, 10),
(12)                        (10, 11),
(12)                        (11, 13),
(8)                     ]
(8)                     mapped = {
(12)                        1: "n",
(12)                        2: "m",
(12)                        3: "l",
(12)                        4: "j",
(12)                        5: "k",
(12)                        6: "i",
(12)                        7: "g",
(12)                        8: "h",
(12)                        9: "f",
(12)                        10: "b",
(12)                        11: "a",
(12)                        12: "d",
(12)                        13: "e",
(8)                     }
(8)                     G1.add_edges_from(edges1)
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
(8)                     nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     G1.add_edges_from([(2, 2), (2, 3), (2, 8), (3, 4)])
(8)                     G2.add_edges_from([("m", "m"), ("m", "l"), ("m", "h"), ("l", "j")])
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m
(8)                     H1 = nx.MultiGraph(G1.subgraph([2, 3, 4, 6, 10, 11, 12, 13]))
(8)                     H2 = nx.MultiGraph(
(12)                        G2.subgraph(
(16)                            [
(20)                                mapped[2],
(20)                                mapped[3],
(20)                                mapped[8],
(20)                                mapped[9],
```

```
(20)                                mapped[10],
(20)                                mapped[11],
(20)                                mapped[12],
(20)                                mapped[13],
(16)                            ]
(12)                        )
(8)                     )
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert not m
(8)                     H2.remove_edges_from(
(12)                        [(mapped[3], mapped[2]), (mapped[9], mapped[8]), (mapped[2],
mapped[2])]
(8)                     )
(8)                     H2.add_edges_from([(mapped[9], mapped[9]), (mapped[2], mapped[8])])
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     H1.remove_node(12)
(8)                     H2.remove_node(mapped[12])
(8)                     H1.add_edge(13, 13)
(8)                     H2.add_edge(mapped[13], mapped[13])
(8)                     H1.add_edges_from([(3, 13), (6, 11)])
(8)                     H2.add_edges_from([(mapped[8], mapped[10]), (mapped[2], mapped[11])])
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     H1.remove_edges_from([(2, 2), (3, 6)])
(8)                     H1.add_edges_from([(6, 6), (2, 3)])
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert not m
(4)                 def test_custom_multigraph4_different_labels(self):
(8)                     G1 = nx.MultiGraph()
(8)                     edges1 = [
(12)                        (1, 2),
(12)                        (1, 2),
(12)                        (2, 2),
(12)                        (2, 3),
(12)                        (3, 8),
(12)                        (3, 8),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 5),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (3, 6),
(12)                        (3, 6),
(12)                        (6, 6),
(12)                        (8, 7),
(12)                        (7, 7),
(12)                        (8, 9),
(12)                        (9, 9),
(12)                        (8, 9),
(12)                        (8, 9),
(12)                        (5, 9),
(12)                        (10, 11),
(12)                        (11, 12),
(12)                        (12, 13),
(12)                        (11, 13),
(8)                     ]
(8)                     mapped = {
(12)                        1: "n",
(12)                        2: "m",
(12)                        3: "l",
(12)                        4: "j",
(12)                        5: "k",
(12)                        6: "i",
(12)                        7: "g",
(12)                        8: "h",
(12)                        9: "f",
(12)                        10: "b",
(12)                        11: "a",
```

```
(12)                        12: "d",
(12)                        13: "e",
(8)                     }
(8)                     G1.add_edges_from(edges1)
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
(8)                     nx.set_node_attributes(
(12)                        G2,
(12)                        dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
(12)                        "label",
(8)                     )
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m == mapped
(8)                     G1.add_edges_from([(2, 2), (2, 3), (2, 8), (3, 4)])
(8)                     G2.add_edges_from([("m", "m"), ("m", "l"), ("m", "h"), ("l", "j")])
(8)                     m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                     assert m == mapped
(8)                     H1 = nx.MultiGraph(G1.subgraph([2, 3, 4, 6]))
(8)                     H2 = nx.MultiGraph(G2.subgraph(["m", "l", "j", "i"]))
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     H1.remove_node(4)
(8)                     H2.remove_node("j")
(8)                     H1.remove_edges_from([(2, 2), (2, 3), (6, 6)])
(8)                     H2.remove_edges_from([("l", "i"), ("m", "m"), ("m", "m")])
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert not m
(8)                     for n1, n2 in zip(H1.nodes(), H2.nodes()):
(12)                        H1.nodes[n1]["label"] = "red"
(12)                        H2.nodes[n2]["label"] = "red"
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     H1.remove_nodes_from([3, 6])
(8)                     H2.remove_nodes_from(["m", "l"])
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     H1.remove_edge(2, 2)
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert not m
(8)                     H2.remove_edge("i", "i")
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     S1 = nx.compose(H1, nx.MultiGraph(G1.subgraph([10, 11, 12, 13])))
(8)                     S2 = nx.compose(H2, nx.MultiGraph(G2.subgraph(["a", "b", "d", "e"])))
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(8)                     S1.add_edges_from([(13, 13), (13, 13), (2, 13)])
(8)                     S2.add_edges_from([("a", "a"), ("a", "a"), ("i", "e")])
(8)                     m = vf2pp_isomorphism(H1, H2, node_label="label")
(8)                     assert m
(4)                 def test_custom_multigraph5_same_labels(self):
(8)                     G1 = nx.MultiGraph()
(8)                     edges1 = [
(12)                        (1, 5),
(12)                        (1, 2),
(12)                        (1, 4),
(12)                        (2, 3),
(12)                        (2, 6),
(12)                        (3, 4),
(12)                        (3, 7),
(12)                        (4, 8),
(12)                        (5, 8),
(12)                        (5, 6),
(12)                        (6, 7),
(12)                        (7, 8),
(8)                     ]
(8)                     mapped = {1: "a", 2: "h", 3: "d", 4: "i", 5: "g", 6: "b", 7: "j", 8:
"c"}
```

```
(8)                 G1.add_edges_from(edges1)
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
(8)                 nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(8)                 G1.add_edges_from(
(12)                    [(1, 2), (1, 2), (3, 7), (8, 8), (8, 8), (7, 8), (2, 3), (5, 6)]
(8)                 )
(8)                 G2.add_edges_from(
(12)                    [
(16)                        ("a", "h"),
(16)                        ("a", "h"),
(16)                        ("d", "j"),
(16)                        ("c", "c"),
(16)                        ("c", "c"),
(16)                        ("j", "c"),
(16)                        ("d", "h"),
(16)                        ("g", "b"),
(12)                    ]
(8)                 )
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(8)                 G2.remove_edges_from(
(12)                    [
(16)                        ("a", "h"),
(16)                        ("a", "h"),
(16)                        ("d", "j"),
(16)                        ("c", "c"),
(16)                        ("c", "c"),
(16)                        ("j", "c"),
(16)                        ("d", "h"),
(16)                        ("g", "b"),
(12)                    ]
(8)                 )
(8)                 G2.add_edges_from(
(12)                    [
(16)                        ("d", "i"),
(16)                        ("a", "h"),
(16)                        ("g", "b"),
(16)                        ("g", "b"),
(16)                        ("i", "i"),
(16)                        ("i", "i"),
(16)                        ("b", "j"),
(16)                        ("d", "j"),
(12)                    ]
(8)                 )
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(4)             def test_disconnected_multigraph_all_same_labels(self):
(8)                 G1 = nx.MultiGraph()
(8)                 G1.add_nodes_from(list(range(10)))
(8)                 G1.add_edges_from([(i, i) for i in range(10)])
(8)                 mapped = {0: 9, 1: 8, 2: 7, 3: 6, 4: 5, 5: 4, 6: 3, 7: 2, 8: 1, 9: 0}
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_same))),
"label")
(8)                 nx.set_node_attributes(G2, dict(zip(G2, it.cycle(labels_same))),
"label")
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(8)                 G1.add_edges_from([(i, i) for i in range(5, 8)] * 3)
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert not m
(8)                 G2.add_edges_from([(i, i) for i in range(3)] * 3)
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
```

```
(8)                 G2.add_edges_from([(0, 0), (1, 1), (1, 1)])
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert not m
(8)                 G1.add_edges_from([(5, 5), (7, 7), (7, 7)])
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(4)             def test_disconnected_multigraph_all_different_labels(self):
(8)                 G1 = nx.MultiGraph()
(8)                 G1.add_nodes_from(list(range(10)))
(8)                 G1.add_edges_from([(i, i) for i in range(10)])
(8)                 mapped = {0: 9, 1: 8, 2: 7, 3: 6, 4: 5, 5: 4, 6: 3, 7: 2, 8: 1, 9: 0}
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
(8)                 nx.set_node_attributes(
(12)                    G2,
(12)                    dict(zip([mapped[n] for n in G1], it.cycle(labels_many))),
(12)                    "label",
(8)                 )
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(8)                 assert m == mapped
(8)                 G1.add_edges_from([(i, i) for i in range(5, 8)] * 3)
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert not m
(8)                 G2.add_edges_from([(mapped[i], mapped[i]) for i in range(3)] * 7)
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert not m
(8)                 G2.add_edges_from([(mapped[i], mapped[i]) for i in range(5, 8)] * 3)
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert not m
(8)                 G1.add_edges_from([(i, i) for i in range(3)] * 7)
(8)                 m = vf2pp_isomorphism(G1, G2, node_label="label")
(8)                 assert m
(0)         class TestDiGraphISOVF2pp:
(4)             def test_wikipedia_graph(self):
(8)                 edges1 = [
(12)                    (1, 5),
(12)                    (1, 2),
(12)                    (1, 4),
(12)                    (3, 2),
(12)                    (6, 2),
(12)                    (3, 4),
(12)                    (7, 3),
(12)                    (4, 8),
(12)                    (5, 8),
(12)                    (6, 5),
(12)                    (6, 7),
(12)                    (7, 8),
(8)                 ]
(8)                 mapped = {1: "a", 2: "h", 3: "d", 4: "i", 5: "g", 6: "b", 7: "j", 8:
"c"}
(8)                 G1 = nx.DiGraph(edges1)
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 assert vf2pp_isomorphism(G1, G2) == mapped
(8)                 G1.remove_edge(1, 5)
(8)                 G1.add_edge(5, 1)
(8)                 assert vf2pp_isomorphism(G1, G2) is None
(4)             def test_non_isomorphic_same_degree_sequence(self):
(8)                 r"""
(16)                        G1                          G2
(8)                 x--------------x             x--------------x
(8)                 | \            |             | \            |
(8)                 |  x-------x   |             |  x-------x   |
(8)                 |  |       |   |             |  |       |   |
(8)                 |  x-------x   |             |  x-------x   |
(8)                 | /            |             |            \ |
(8)                 x--------------x             x--------------x
(8)                 """
```

```
1245: (8)                   edges1 = [
1246: (12)                      (1, 5),
1247: (12)                      (1, 2),
1248: (12)                      (4, 1),
1249: (12)                      (3, 2),
1250: (12)                      (3, 4),
1251: (12)                      (4, 8),
1252: (12)                      (5, 8),
1253: (12)                      (6, 5),
1254: (12)                      (6, 7),
1255: (12)                      (7, 8),
1256: (8)                   ]
1257: (8)                   edges2 = [
1258: (12)                      (1, 5),
1259: (12)                      (1, 2),
1260: (12)                      (4, 1),
1261: (12)                      (3, 2),
1262: (12)                      (4, 3),
1263: (12)                      (5, 8),
1264: (12)                      (6, 5),
1265: (12)                      (6, 7),
1266: (12)                      (3, 7),
1267: (12)                      (8, 7),
1268: (8)                   ]
1269: (8)                   G1 = nx.DiGraph(edges1)
1270: (8)                   G2 = nx.DiGraph(edges2)
1271: (8)                   assert vf2pp_isomorphism(G1, G2) is None
```

File 271 - test_vf2pp_helpers.py:

```
1: (0)              import itertools as it
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx import vf2pp_is_isomorphic, vf2pp_isomorphism
5: (0)              from networkx.algorithms.isomorphism.vf2pp import (
6: (4)                  _consistent_PT,
7: (4)                  _cut_PT,
8: (4)                  _feasibility,
9: (4)                  _find_candidates,
10: (4)                 _find_candidates_Di,
11: (4)                 _GraphParameters,
12: (4)                 _initialize_parameters,
13: (4)                 _matching_order,
14: (4)                 _restore_Tinout,
15: (4)                 _restore_Tinout_Di,
16: (4)                 _StateParameters,
17: (4)                 _update_Tinout,
18: (0)             )
19: (0)             labels_same = ["blue"]
20: (0)             labels_many = [
21: (4)                 "white",
22: (4)                 "red",
23: (4)                 "blue",
24: (4)                 "green",
25: (4)                 "orange",
26: (4)                 "black",
27: (4)                 "purple",
28: (4)                 "yellow",
29: (4)                 "brown",
30: (4)                 "cyan",
31: (4)                 "solarized",
32: (4)                 "pink",
33: (4)                 "none",
34: (0)             ]
35: (0)             class TestNodeOrdering:
36: (4)                 def test_empty_graph(self):
37: (8)                     G1 = nx.Graph()
```

```
(8)                     G2 = nx.Graph()
(8)                     gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                     assert len(set(_matching_order(gparams))) == 0
(4)                 def test_single_node(self):
(8)                     G1 = nx.Graph()
(8)                     G2 = nx.Graph()
(8)                     G1.add_node(1)
(8)                     G2.add_node(1)
(8)                     nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels_many))),
"label")
(8)                     nx.set_node_attributes(
(12)                        G2,
(12)                        dict(zip(G2, it.cycle(labels_many))),
(12)                        "label",
(8)                     )
(8)                     l1, l2 = (
(12)                        nx.get_node_attributes(G1, "label"),
(12)                        nx.get_node_attributes(G2, "label"),
(8)                     )
(8)                     gparams = _GraphParameters(
(12)                        G1,
(12)                        G2,
(12)                        l1,
(12)                        l2,
(12)                        nx.utils.groups(l1),
(12)                        nx.utils.groups(l2),
(12)                        nx.utils.groups(dict(G2.degree())),
(8)                     )
(8)                     m = _matching_order(gparams)
(8)                     assert m == [1]
(4)                 def test_matching_order(self):
(8)                     labels = [
(12)                        "blue",
(12)                        "blue",
(12)                        "red",
(12)                        "red",
(12)                        "red",
(12)                        "red",
(12)                        "green",
(12)                        "green",
(12)                        "green",
(12)                        "yellow",
(12)                        "purple",
(12)                        "purple",
(12)                        "blue",
(12)                        "blue",
(8)                     ]
(8)                     G1 = nx.Graph(
(12)                        [
(16)                            (0, 1),
(16)                            (0, 2),
(16)                            (1, 2),
(16)                            (2, 5),
(16)                            (2, 4),
(16)                            (1, 3),
(16)                            (1, 4),
(16)                            (3, 6),
(16)                            (4, 6),
(16)                            (6, 7),
(16)                            (7, 8),
(16)                            (9, 10),
(16)                            (9, 11),
(16)                            (11, 12),
(16)                            (11, 13),
(16)                            (12, 13),
(16)                            (10, 13),
(12)                        ]
(8)                     )
(8)                     G2 = G1.copy()
```

```
106: (8)                    nx.set_node_attributes(G1, dict(zip(G1, it.cycle(labels))), "label")
107: (8)                    nx.set_node_attributes(
108: (12)                       G2,
109: (12)                       dict(zip(G2, it.cycle(labels))),
110: (12)                       "label",
111: (8)                    )
112: (8)                    l1, l2 = (
113: (12)                       nx.get_node_attributes(G1, "label"),
114: (12)                       nx.get_node_attributes(G2, "label"),
115: (8)                    )
116: (8)                    gparams = _GraphParameters(
117: (12)                       G1,
118: (12)                       G2,
119: (12)                       l1,
120: (12)                       l2,
121: (12)                       nx.utils.groups(l1),
122: (12)                       nx.utils.groups(l2),
123: (12)                       nx.utils.groups(dict(G2.degree())),
124: (8)                    )
125: (8)                    expected = [9, 11, 10, 13, 12, 1, 2, 4, 0, 3, 6, 5, 7, 8]
126: (8)                    assert _matching_order(gparams) == expected
127: (4)                def test_matching_order_all_branches(self):
128: (8)                    G1 = nx.Graph(
129: (12)                       [(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 4),
(3, 4)]
130: (8)                    )
131: (8)                    G1.add_node(5)
132: (8)                    G2 = G1.copy()
133: (8)                    G1.nodes[0]["label"] = "black"
134: (8)                    G1.nodes[1]["label"] = "blue"
135: (8)                    G1.nodes[2]["label"] = "blue"
136: (8)                    G1.nodes[3]["label"] = "red"
137: (8)                    G1.nodes[4]["label"] = "red"
138: (8)                    G1.nodes[5]["label"] = "blue"
139: (8)                    G2.nodes[0]["label"] = "black"
140: (8)                    G2.nodes[1]["label"] = "blue"
141: (8)                    G2.nodes[2]["label"] = "blue"
142: (8)                    G2.nodes[3]["label"] = "red"
143: (8)                    G2.nodes[4]["label"] = "red"
144: (8)                    G2.nodes[5]["label"] = "blue"
145: (8)                    l1, l2 = (
146: (12)                       nx.get_node_attributes(G1, "label"),
147: (12)                       nx.get_node_attributes(G2, "label"),
148: (8)                    )
149: (8)                    gparams = _GraphParameters(
150: (12)                       G1,
151: (12)                       G2,
152: (12)                       l1,
153: (12)                       l2,
154: (12)                       nx.utils.groups(l1),
155: (12)                       nx.utils.groups(l2),
156: (12)                       nx.utils.groups(dict(G2.degree())),
157: (8)                    )
158: (8)                    expected = [0, 4, 1, 3, 2, 5]
159: (8)                    assert _matching_order(gparams) == expected
160: (0)            class TestGraphCandidateSelection:
161: (4)                G1_edges = [
162: (8)                    (1, 2),
163: (8)                    (1, 4),
164: (8)                    (1, 5),
165: (8)                    (2, 3),
166: (8)                    (2, 4),
167: (8)                    (3, 4),
168: (8)                    (4, 5),
169: (8)                    (1, 6),
170: (8)                    (6, 7),
171: (8)                    (6, 8),
172: (8)                    (8, 9),
173: (8)                    (7, 9),
```

```
(4)                 ]
(4)                 mapped = {
(8)                     0: "x",
(8)                     1: "a",
(8)                     2: "b",
(8)                     3: "c",
(8)                     4: "d",
(8)                     5: "e",
(8)                     6: "f",
(8)                     7: "g",
(8)                     8: "h",
(8)                     9: "i",
(4)                 }
(4)                 def test_no_covered_neighbors_no_labels(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_edges_from(self.G1_edges)
(8)                     G1.add_node(0)
(8)                     G2 = nx.relabel_nodes(G1, self.mapped)
(8)                     G1_degree = dict(G1.degree)
(8)                     l1 = dict(G1.nodes(data="label", default=-1))
(8)                     l2 = dict(G2.nodes(data="label", default=-1))
(8)                     gparams = _GraphParameters(
(12)                        G1,
(12)                        G2,
(12)                        l1,
(12)                        l2,
(12)                        nx.utils.groups(l1),
(12)                        nx.utils.groups(l2),
(12)                        nx.utils.groups(dict(G2.degree())),
(8)                     )
(8)                     m = {9: self.mapped[9], 1: self.mapped[1]}
(8)                     m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
(8)                     T1 = {7, 8, 2, 4, 5}
(8)                     T1_tilde = {0, 3, 6}
(8)                     T2 = {"g", "h", "b", "d", "e"}
(8)                     T2_tilde = {"x", "c", "f"}
(8)                     sparams = _StateParameters(
(12)                        m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                     )
(8)                     u = 3
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     u = 0
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     m.pop(9)
(8)                     m_rev.pop(self.mapped[9])
(8)                     T1 = {2, 4, 5, 6}
(8)                     T1_tilde = {0, 3, 7, 8, 9}
(8)                     T2 = {"g", "h", "b", "d", "e", "f"}
(8)                     T2_tilde = {"x", "c", "g", "h", "i"}
(8)                     sparams = _StateParameters(
(12)                        m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                     )
(8)                     u = 7
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {
(12)                        self.mapped[u],
(12)                        self.mapped[8],
(12)                        self.mapped[3],
(12)                        self.mapped[9],
(8)                     }
(4)                 def test_no_covered_neighbors_with_labels(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_edges_from(self.G1_edges)
(8)                     G1.add_node(0)
(8)                     G2 = nx.relabel_nodes(G1, self.mapped)
(8)                     G1_degree = dict(G1.degree)
(8)                     nx.set_node_attributes(
```

```
(12)                        G1,
(12)                        dict(zip(G1, it.cycle(labels_many))),
(12)                        "label",
(8)                     )
(8)                     nx.set_node_attributes(
(12)                        G2,
(12)                        dict(
(16)                            zip(
(20)                                [self.mapped[n] for n in G1],
(20)                                it.cycle(labels_many),
(16)                            )
(12)                        ),
(12)                        "label",
(8)                     )
(8)                     l1 = dict(G1.nodes(data="label", default=-1))
(8)                     l2 = dict(G2.nodes(data="label", default=-1))
(8)                     gparams = _GraphParameters(
(12)                        G1,
(12)                        G2,
(12)                        l1,
(12)                        l2,
(12)                        nx.utils.groups(l1),
(12)                        nx.utils.groups(l2),
(12)                        nx.utils.groups(dict(G2.degree())),
(8)                     )
(8)                     m = {9: self.mapped[9], 1: self.mapped[1]}
(8)                     m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
(8)                     T1 = {7, 8, 2, 4, 5, 6}
(8)                     T1_tilde = {0, 3}
(8)                     T2 = {"g", "h", "b", "d", "e", "f"}
(8)                     T2_tilde = {"x", "c"}
(8)                     sparams = _StateParameters(
(12)                        m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                     )
(8)                     u = 3
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     u = 0
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     G1.nodes[u]["label"] = "blue"
(8)                     l1 = dict(G1.nodes(data="label", default=-1))
(8)                     l2 = dict(G2.nodes(data="label", default=-1))
(8)                     gparams = _GraphParameters(
(12)                        G1,
(12)                        G2,
(12)                        l1,
(12)                        l2,
(12)                        nx.utils.groups(l1),
(12)                        nx.utils.groups(l2),
(12)                        nx.utils.groups(dict(G2.degree())),
(8)                     )
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == set()
(8)                     m.pop(9)
(8)                     m_rev.pop(self.mapped[9])
(8)                     T1 = {2, 4, 5, 6}
(8)                     T1_tilde = {0, 3, 7, 8, 9}
(8)                     T2 = {"b", "d", "e", "f"}
(8)                     T2_tilde = {"x", "c", "g", "h", "i"}
(8)                     sparams = _StateParameters(
(12)                        m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                     )
(8)                     u = 7
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     G1.nodes[8]["label"] = G1.nodes[7]["label"]
(8)                     G2.nodes[self.mapped[8]]["label"] = G1.nodes[7]["label"]
(8)                     l1 = dict(G1.nodes(data="label", default=-1))
```

```
(8)                 l2 = dict(G2.nodes(data="label", default=-1))
(8)                 gparams = _GraphParameters(
(12)                    G1,
(12)                    G2,
(12)                    l1,
(12)                    l2,
(12)                    nx.utils.groups(l1),
(12)                    nx.utils.groups(l2),
(12)                    nx.utils.groups(dict(G2.degree())),
(8)                 )
(8)                 candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                 assert candidates == {self.mapped[u], self.mapped[8]}
(4)             def test_covered_neighbors_no_labels(self):
(8)                 G1 = nx.Graph()
(8)                 G1.add_edges_from(self.G1_edges)
(8)                 G1.add_node(0)
(8)                 G2 = nx.relabel_nodes(G1, self.mapped)
(8)                 G1_degree = dict(G1.degree)
(8)                 l1 = dict(G1.nodes(data=None, default=-1))
(8)                 l2 = dict(G2.nodes(data=None, default=-1))
(8)                 gparams = _GraphParameters(
(12)                    G1,
(12)                    G2,
(12)                    l1,
(12)                    l2,
(12)                    nx.utils.groups(l1),
(12)                    nx.utils.groups(l2),
(12)                    nx.utils.groups(dict(G2.degree())),
(8)                 )
(8)                 m = {9: self.mapped[9], 1: self.mapped[1]}
(8)                 m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
(8)                 T1 = {7, 8, 2, 4, 5, 6}
(8)                 T1_tilde = {0, 3}
(8)                 T2 = {"g", "h", "b", "d", "e", "f"}
(8)                 T2_tilde = {"x", "c"}
(8)                 sparams = _StateParameters(
(12)                    m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                 )
(8)                 u = 5
(8)                 candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                 assert candidates == {self.mapped[u]}
(8)                 u = 6
(8)                 candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                 assert candidates == {self.mapped[u], self.mapped[2]}
(4)             def test_covered_neighbors_with_labels(self):
(8)                 G1 = nx.Graph()
(8)                 G1.add_edges_from(self.G1_edges)
(8)                 G1.add_node(0)
(8)                 G2 = nx.relabel_nodes(G1, self.mapped)
(8)                 G1_degree = dict(G1.degree)
(8)                 nx.set_node_attributes(
(12)                    G1,
(12)                    dict(zip(G1, it.cycle(labels_many))),
(12)                    "label",
(8)                 )
(8)                 nx.set_node_attributes(
(12)                    G2,
(12)                    dict(
(16)                        zip(
(20)                            [self.mapped[n] for n in G1],
(20)                            it.cycle(labels_many),
(16)                        )
(12)                    ),
(12)                    "label",
(8)                 )
(8)                 l1 = dict(G1.nodes(data="label", default=-1))
(8)                 l2 = dict(G2.nodes(data="label", default=-1))
(8)                 gparams = _GraphParameters(
(12)                    G1,
```

```
(12)                        G2,
(12)                        l1,
(12)                        l2,
(12)                        nx.utils.groups(l1),
(12)                        nx.utils.groups(l2),
(12)                        nx.utils.groups(dict(G2.degree())),
(8)                     )
(8)                     m = {9: self.mapped[9], 1: self.mapped[1]}
(8)                     m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
(8)                     T1 = {7, 8, 2, 4, 5, 6}
(8)                     T1_tilde = {0, 3}
(8)                     T2 = {"g", "h", "b", "d", "e", "f"}
(8)                     T2_tilde = {"x", "c"}
(8)                     sparams = _StateParameters(
(12)                        m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                     )
(8)                     u = 5
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     u = 6
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u]}
(8)                     G1.nodes[2]["label"] = G1.nodes[u]["label"]
(8)                     G2.nodes[self.mapped[2]]["label"] = G1.nodes[u]["label"]
(8)                     l1 = dict(G1.nodes(data="label", default=-1))
(8)                     l2 = dict(G2.nodes(data="label", default=-1))
(8)                     gparams = _GraphParameters(
(12)                        G1,
(12)                        G2,
(12)                        l1,
(12)                        l2,
(12)                        nx.utils.groups(l1),
(12)                        nx.utils.groups(l2),
(12)                        nx.utils.groups(dict(G2.degree())),
(8)                     )
(8)                     candidates = _find_candidates(u, gparams, sparams, G1_degree)
(8)                     assert candidates == {self.mapped[u], self.mapped[2]}
(0)             class TestDiGraphCandidateSelection:
(4)                 G1_edges = [
(8)                     (1, 2),
(8)                     (1, 4),
(8)                     (5, 1),
(8)                     (2, 3),
(8)                     (4, 2),
(8)                     (3, 4),
(8)                     (4, 5),
(8)                     (1, 6),
(8)                     (6, 7),
(8)                     (6, 8),
(8)                     (8, 9),
(8)                     (7, 9),
(4)                 ]
(4)                 mapped = {
(8)                     0: "x",
(8)                     1: "a",
(8)                     2: "b",
(8)                     3: "c",
(8)                     4: "d",
(8)                     5: "e",
(8)                     6: "f",
(8)                     7: "g",
(8)                     8: "h",
(8)                     9: "i",
(4)                 }
(4)                 def test_no_covered_neighbors_no_labels(self):
(8)                     G1 = nx.DiGraph()
(8)                     G1.add_edges_from(self.G1_edges)
(8)                     G1.add_node(0)
(8)                     G2 = nx.relabel_nodes(G1, self.mapped)
```

```
450: (8)                    G1_degree = {
451: (12)                       n: (in_degree, out_degree)
452: (12)                       for (n, in_degree), (_, out_degree) in zip(G1.in_degree,
G1.out_degree)
453: (8)                    }
454: (8)                    l1 = dict(G1.nodes(data="label", default=-1))
455: (8)                    l2 = dict(G2.nodes(data="label", default=-1))
456: (8)                    gparams = _GraphParameters(
457: (12)                       G1,
458: (12)                       G2,
459: (12)                       l1,
460: (12)                       l2,
461: (12)                       nx.utils.groups(l1),
462: (12)                       nx.utils.groups(l2),
463: (12)                       nx.utils.groups(
464: (16)                           {
465: (20)                               node: (in_degree, out_degree)
466: (20)                               for (node, in_degree), (_, out_degree) in zip(
467: (24)                                   G2.in_degree(), G2.out_degree()
468: (20)                               )
469: (16)                           }
470: (12)                       ),
471: (8)                    )
472: (8)                    m = {9: self.mapped[9], 1: self.mapped[1]}
473: (8)                    m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
474: (8)                    T1_out = {2, 4, 6}
475: (8)                    T1_in = {5, 7, 8}
476: (8)                    T1_tilde = {0, 3}
477: (8)                    T2_out = {"b", "d", "f"}
478: (8)                    T2_in = {"e", "g", "h"}
479: (8)                    T2_tilde = {"x", "c"}
480: (8)                    sparams = _StateParameters(
481: (12)                       m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
482: (8)                    )
483: (8)                    u = 3
484: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
485: (8)                    assert candidates == {self.mapped[u]}
486: (8)                    u = 0
487: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
488: (8)                    assert candidates == {self.mapped[u]}
489: (8)                    m.pop(9)
490: (8)                    m_rev.pop(self.mapped[9])
491: (8)                    T1_out = {2, 4, 6}
492: (8)                    T1_in = {5}
493: (8)                    T1_tilde = {0, 3, 7, 8, 9}
494: (8)                    T2_out = {"b", "d", "f"}
495: (8)                    T2_in = {"e"}
496: (8)                    T2_tilde = {"x", "c", "g", "h", "i"}
497: (8)                    sparams = _StateParameters(
498: (12)                       m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
499: (8)                    )
500: (8)                    u = 7
501: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
502: (8)                    assert candidates == {self.mapped[u], self.mapped[8], self.mapped[3]}
503: (4)                def test_no_covered_neighbors_with_labels(self):
504: (8)                    G1 = nx.DiGraph()
505: (8)                    G1.add_edges_from(self.G1_edges)
506: (8)                    G1.add_node(0)
507: (8)                    G2 = nx.relabel_nodes(G1, self.mapped)
508: (8)                    G1_degree = {
509: (12)                       n: (in_degree, out_degree)
510: (12)                       for (n, in_degree), (_, out_degree) in zip(G1.in_degree,
G1.out_degree)
511: (8)                    }
512: (8)                    nx.set_node_attributes(
513: (12)                       G1,
514: (12)                       dict(zip(G1, it.cycle(labels_many))),
```

```
515: (12)                        "label",
516: (8)                    )
517: (8)                    nx.set_node_attributes(
518: (12)                        G2,
519: (12)                        dict(
520: (16)                            zip(
521: (20)                                [self.mapped[n] for n in G1],
522: (20)                                it.cycle(labels_many),
523: (16)                            )
524: (12)                        ),
525: (12)                        "label",
526: (8)                    )
527: (8)                    l1 = dict(G1.nodes(data="label", default=-1))
528: (8)                    l2 = dict(G2.nodes(data="label", default=-1))
529: (8)                    gparams = _GraphParameters(
530: (12)                        G1,
531: (12)                        G2,
532: (12)                        l1,
533: (12)                        l2,
534: (12)                        nx.utils.groups(l1),
535: (12)                        nx.utils.groups(l2),
536: (12)                        nx.utils.groups(
537: (16)                            {
538: (20)                                node: (in_degree, out_degree)
539: (20)                                for (node, in_degree), (_, out_degree) in zip(
540: (24)                                    G2.in_degree(), G2.out_degree()
541: (20)                                )
542: (16)                            }
543: (12)                        ),
544: (8)                    )
545: (8)                    m = {9: self.mapped[9], 1: self.mapped[1]}
546: (8)                    m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
547: (8)                    T1_out = {2, 4, 6}
548: (8)                    T1_in = {5, 7, 8}
549: (8)                    T1_tilde = {0, 3}
550: (8)                    T2_out = {"b", "d", "f"}
551: (8)                    T2_in = {"e", "g", "h"}
552: (8)                    T2_tilde = {"x", "c"}
553: (8)                    sparams = _StateParameters(
554: (12)                        m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
555: (8)                    )
556: (8)                    u = 3
557: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
558: (8)                    assert candidates == {self.mapped[u]}
559: (8)                    u = 0
560: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
561: (8)                    assert candidates == {self.mapped[u]}
562: (8)                    G1.nodes[u]["label"] = "blue"
563: (8)                    l1 = dict(G1.nodes(data="label", default=-1))
564: (8)                    l2 = dict(G2.nodes(data="label", default=-1))
565: (8)                    gparams = _GraphParameters(
566: (12)                        G1,
567: (12)                        G2,
568: (12)                        l1,
569: (12)                        l2,
570: (12)                        nx.utils.groups(l1),
571: (12)                        nx.utils.groups(l2),
572: (12)                        nx.utils.groups(
573: (16)                            {
574: (20)                                node: (in_degree, out_degree)
575: (20)                                for (node, in_degree), (_, out_degree) in zip(
576: (24)                                    G2.in_degree(), G2.out_degree()
577: (20)                                )
578: (16)                            }
579: (12)                        ),
580: (8)                    )
581: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
582: (8)                    assert candidates == set()
```

```
583: (8)                    m.pop(9)
584: (8)                    m_rev.pop(self.mapped[9])
585: (8)                    T1_out = {2, 4, 6}
586: (8)                    T1_in = {5}
587: (8)                    T1_tilde = {0, 3, 7, 8, 9}
588: (8)                    T2_out = {"b", "d", "f"}
589: (8)                    T2_in = {"e"}
590: (8)                    T2_tilde = {"x", "c", "g", "h", "i"}
591: (8)                    sparams = _StateParameters(
592: (12)                       m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
593: (8)                    )
594: (8)                    u = 7
595: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
596: (8)                    assert candidates == {self.mapped[u]}
597: (8)                    G1.nodes[8]["label"] = G1.nodes[7]["label"]
598: (8)                    G2.nodes[self.mapped[8]]["label"] = G1.nodes[7]["label"]
599: (8)                    l1 = dict(G1.nodes(data="label", default=-1))
600: (8)                    l2 = dict(G2.nodes(data="label", default=-1))
601: (8)                    gparams = _GraphParameters(
602: (12)                       G1,
603: (12)                       G2,
604: (12)                       l1,
605: (12)                       l2,
606: (12)                       nx.utils.groups(l1),
607: (12)                       nx.utils.groups(l2),
608: (12)                       nx.utils.groups(
609: (16)                           {
610: (20)                               node: (in_degree, out_degree)
611: (20)                               for (node, in_degree), (_, out_degree) in zip(
612: (24)                                   G2.in_degree(), G2.out_degree()
613: (20)                               )
614: (16)                           }
615: (12)                       ),
616: (8)                    )
617: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
618: (8)                    assert candidates == {self.mapped[u], self.mapped[8]}
619: (4)                def test_covered_neighbors_no_labels(self):
620: (8)                    G1 = nx.DiGraph()
621: (8)                    G1.add_edges_from(self.G1_edges)
622: (8)                    G1.add_node(0)
623: (8)                    G2 = nx.relabel_nodes(G1, self.mapped)
624: (8)                    G1_degree = {
625: (12)                       n: (in_degree, out_degree)
626: (12)                       for (n, in_degree), (_, out_degree) in zip(G1.in_degree,
G1.out_degree)
627: (8)                    }
628: (8)                    l1 = dict(G1.nodes(data=None, default=-1))
629: (8)                    l2 = dict(G2.nodes(data=None, default=-1))
630: (8)                    gparams = _GraphParameters(
631: (12)                       G1,
632: (12)                       G2,
633: (12)                       l1,
634: (12)                       l2,
635: (12)                       nx.utils.groups(l1),
636: (12)                       nx.utils.groups(l2),
637: (12)                       nx.utils.groups(
638: (16)                           {
639: (20)                               node: (in_degree, out_degree)
640: (20)                               for (node, in_degree), (_, out_degree) in zip(
641: (24)                                   G2.in_degree(), G2.out_degree()
642: (20)                               )
643: (16)                           }
644: (12)                       ),
645: (8)                    )
646: (8)                    m = {9: self.mapped[9], 1: self.mapped[1]}
647: (8)                    m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
648: (8)                    T1_out = {2, 4, 6}
649: (8)                    T1_in = {5, 7, 8}
```

```
650: (8)                    T1_tilde = {0, 3}
651: (8)                    T2_out = {"b", "d", "f"}
652: (8)                    T2_in = {"e", "g", "h"}
653: (8)                    T2_tilde = {"x", "c"}
654: (8)                    sparams = _StateParameters(
655: (12)                       m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
656: (8)                    )
657: (8)                    u = 5
658: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
659: (8)                    assert candidates == {self.mapped[u]}
660: (8)                    u = 6
661: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
662: (8)                    assert candidates == {self.mapped[u]}
663: (8)                    G1.remove_edge(4, 2)
664: (8)                    G1.add_edge(2, 4)
665: (8)                    G2.remove_edge("d", "b")
666: (8)                    G2.add_edge("b", "d")
667: (8)                    gparams = _GraphParameters(
668: (12)                       G1,
669: (12)                       G2,
670: (12)                       l1,
671: (12)                       l2,
672: (12)                       nx.utils.groups(l1),
673: (12)                       nx.utils.groups(l2),
674: (12)                       nx.utils.groups(
675: (16)                           {
676: (20)                               node: (in_degree, out_degree)
677: (20)                               for (node, in_degree), (_, out_degree) in zip(
678: (24)                                   G2.in_degree(), G2.out_degree()
679: (20)                               )
680: (16)                           }
681: (12)                       ),
682: (8)                    )
683: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
684: (8)                    assert candidates == {self.mapped[u], self.mapped[2]}
685: (4)                def test_covered_neighbors_with_labels(self):
686: (8)                    G1 = nx.DiGraph()
687: (8)                    G1.add_edges_from(self.G1_edges)
688: (8)                    G1.add_node(0)
689: (8)                    G2 = nx.relabel_nodes(G1, self.mapped)
690: (8)                    G1.remove_edge(4, 2)
691: (8)                    G1.add_edge(2, 4)
692: (8)                    G2.remove_edge("d", "b")
693: (8)                    G2.add_edge("b", "d")
694: (8)                    G1_degree = {
695: (12)                       n: (in_degree, out_degree)
696: (12)                       for (n, in_degree), (_, out_degree) in zip(G1.in_degree,
G1.out_degree)
697: (8)                    }
698: (8)                    nx.set_node_attributes(
699: (12)                       G1,
700: (12)                       dict(zip(G1, it.cycle(labels_many))),
701: (12)                       "label",
702: (8)                    )
703: (8)                    nx.set_node_attributes(
704: (12)                       G2,
705: (12)                       dict(
706: (16)                           zip(
707: (20)                               [self.mapped[n] for n in G1],
708: (20)                               it.cycle(labels_many),
709: (16)                           )
710: (12)                       ),
711: (12)                       "label",
712: (8)                    )
713: (8)                    l1 = dict(G1.nodes(data="label", default=-1))
714: (8)                    l2 = dict(G2.nodes(data="label", default=-1))
715: (8)                    gparams = _GraphParameters(
716: (12)                       G1,
```

```
717: (12)                        G2,
718: (12)                        l1,
719: (12)                        l2,
720: (12)                        nx.utils.groups(l1),
721: (12)                        nx.utils.groups(l2),
722: (12)                        nx.utils.groups(
723: (16)                            {
724: (20)                                node: (in_degree, out_degree)
725: (20)                                for (node, in_degree), (_, out_degree) in zip(
726: (24)                                    G2.in_degree(), G2.out_degree()
727: (20)                                )
728: (16)                            }
729: (12)                        ),
730: (8)                    )
731: (8)                    m = {9: self.mapped[9], 1: self.mapped[1]}
732: (8)                    m_rev = {self.mapped[9]: 9, self.mapped[1]: 1}
733: (8)                    T1_out = {2, 4, 6}
734: (8)                    T1_in = {5, 7, 8}
735: (8)                    T1_tilde = {0, 3}
736: (8)                    T2_out = {"b", "d", "f"}
737: (8)                    T2_in = {"e", "g", "h"}
738: (8)                    T2_tilde = {"x", "c"}
739: (8)                    sparams = _StateParameters(
740: (12)                        m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
741: (8)                    )
742: (8)                    u = 5
743: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
744: (8)                    assert candidates == {self.mapped[u]}
745: (8)                    u = 6
746: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
747: (8)                    assert candidates == {self.mapped[u]}
748: (8)                    G1.nodes[2]["label"] = G1.nodes[u]["label"]
749: (8)                    G2.nodes[self.mapped[2]]["label"] = G1.nodes[u]["label"]
750: (8)                    l1 = dict(G1.nodes(data="label", default=-1))
751: (8)                    l2 = dict(G2.nodes(data="label", default=-1))
752: (8)                    gparams = _GraphParameters(
753: (12)                        G1,
754: (12)                        G2,
755: (12)                        l1,
756: (12)                        l2,
757: (12)                        nx.utils.groups(l1),
758: (12)                        nx.utils.groups(l2),
759: (12)                        nx.utils.groups(
760: (16)                            {
761: (20)                                node: (in_degree, out_degree)
762: (20)                                for (node, in_degree), (_, out_degree) in zip(
763: (24)                                    G2.in_degree(), G2.out_degree()
764: (20)                                )
765: (16)                            }
766: (12)                        ),
767: (8)                    )
768: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
769: (8)                    assert candidates == {self.mapped[u], self.mapped[2]}
770: (8)                    G1.remove_edge(2, 4)
771: (8)                    G1.add_edge(4, 2)
772: (8)                    G2.remove_edge("b", "d")
773: (8)                    G2.add_edge("d", "b")
774: (8)                    gparams = _GraphParameters(
775: (12)                        G1,
776: (12)                        G2,
777: (12)                        l1,
778: (12)                        l2,
779: (12)                        nx.utils.groups(l1),
780: (12)                        nx.utils.groups(l2),
781: (12)                        nx.utils.groups(
782: (16)                            {
783: (20)                                node: (in_degree, out_degree)
784: (20)                                for (node, in_degree), (_, out_degree) in zip(
```

```
785: (24)                                   G2.in_degree(), G2.out_degree()
786: (20)                               )
787: (16)                           }
788: (12)                       ),
789: (8)                    )
790: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, G1_degree)
791: (8)                    assert candidates == {self.mapped[u]}
792: (4)                def test_same_in_out_degrees_no_candidate(self):
793: (8)                    g1 = nx.DiGraph([(4, 1), (4, 2), (3, 4), (5, 4), (6, 4)])
794: (8)                    g2 = nx.DiGraph([(1, 4), (2, 4), (3, 4), (4, 5), (4, 6)])
795: (8)                    l1 = dict(g1.nodes(data=None, default=-1))
796: (8)                    l2 = dict(g2.nodes(data=None, default=-1))
797: (8)                    gparams = _GraphParameters(
798: (12)                       g1,
799: (12)                       g2,
800: (12)                       l1,
801: (12)                       l2,
802: (12)                       nx.utils.groups(l1),
803: (12)                       nx.utils.groups(l2),
804: (12)                       nx.utils.groups(
805: (16)                           {
806: (20)                               node: (in_degree, out_degree)
807: (20)                               for (node, in_degree), (_, out_degree) in zip(
808: (24)                                   g2.in_degree(), g2.out_degree()
809: (20)                               )
810: (16)                           }
811: (12)                       ),
812: (8)                    )
813: (8)                    g1_degree = {
814: (12)                       n: (in_degree, out_degree)
815: (12)                       for (n, in_degree), (_, out_degree) in zip(g1.in_degree,
g1.out_degree)
816: (8)                    }
817: (8)                    m = {1: 1, 2: 2, 3: 3}
818: (8)                    m_rev = m.copy()
819: (8)                    T1_out = {4}
820: (8)                    T1_in = {4}
821: (8)                    T1_tilde = {5, 6}
822: (8)                    T2_out = {4}
823: (8)                    T2_in = {4}
824: (8)                    T2_tilde = {5, 6}
825: (8)                    sparams = _StateParameters(
826: (12)                       m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
827: (8)                    )
828: (8)                    u = 4
829: (8)                    candidates = _find_candidates_Di(u, gparams, sparams, g1_degree)
830: (8)                    assert candidates == set()
831: (8)                    assert _find_candidates(u, gparams, sparams, g1_degree) == {4}
832: (0)            class TestGraphISOFeasibility:
833: (4)                def test_const_covered_neighbors(self):
834: (8)                    G1 = nx.Graph([(0, 1), (1, 2), (3, 0), (3, 2)])
835: (8)                    G2 = nx.Graph([("a", "b"), ("b", "c"), ("k", "a"), ("k", "c")])
836: (8)                    gparams = _GraphParameters(G1, G2, None, None, None, None, None)
837: (8)                    sparams = _StateParameters(
838: (12)                       {0: "a", 1: "b", 2: "c"},
839: (12)                       {"a": 0, "b": 1, "c": 2},
840: (12)                       None,
841: (12)                       None,
842: (12)                       None,
843: (12)                       None,
844: (12)                       None,
845: (12)                       None,
846: (12)                       None,
847: (12)                       None,
848: (8)                    )
849: (8)                    u, v = 3, "k"
850: (8)                    assert _consistent_PT(u, v, gparams, sparams)
851: (4)                def test_const_no_covered_neighbors(self):
```

```
        G1 = nx.Graph([(0, 1), (1, 2), (3, 4), (3, 5)])
        G2 = nx.Graph([("a", "b"), ("b", "c"), ("k", "w"), ("k", "z")])
        gparams = _GraphParameters(G1, G2, None, None, None, None, None)
        sparams = _StateParameters(
            {0: "a", 1: "b", 2: "c"},
            {"a": 0, "b": 1, "c": 2},
            None,
            None,
            None,
            None,
            None,
            None,
            None,
            None,
        )
        u, v = 3, "k"
        assert _consistent_PT(u, v, gparams, sparams)
    def test_const_mixed_covered_uncovered_neighbors(self):
        G1 = nx.Graph([(0, 1), (1, 2), (3, 0), (3, 2), (3, 4), (3, 5)])
        G2 = nx.Graph(
            [("a", "b"), ("b", "c"), ("k", "a"), ("k", "c"), ("k", "w"), ("k", "z")]
        )
        gparams = _GraphParameters(G1, G2, None, None, None, None, None)
        sparams = _StateParameters(
            {0: "a", 1: "b", 2: "c"},
            {"a": 0, "b": 1, "c": 2},
            None,
            None,
            None,
            None,
            None,
            None,
            None,
            None,
        )
        u, v = 3, "k"
        assert _consistent_PT(u, v, gparams, sparams)
    def test_const_fail_cases(self):
        G1 = nx.Graph(
            [
                (0, 1),
                (1, 2),
                (10, 0),
                (10, 3),
                (10, 4),
                (10, 5),
                (10, 6),
                (4, 1),
                (5, 3),
            ]
        )
        G2 = nx.Graph(
            [
                ("a", "b"),
                ("b", "c"),
                ("k", "a"),
                ("k", "d"),
                ("k", "e"),
                ("k", "f"),
                ("k", "g"),
                ("e", "b"),
                ("f", "d"),
            ]
        )
        gparams = _GraphParameters(G1, G2, None, None, None, None, None)
        sparams = _StateParameters(
            {0: "a", 1: "b", 2: "c", 3: "d"},
            {"a": 0, "b": 1, "c": 2, "d": 3},
```

```
920: (12)                        None,
921: (12)                        None,
922: (12)                        None,
923: (12)                        None,
924: (12)                        None,
925: (12)                        None,
926: (12)                        None,
927: (12)                        None,
928: (8)                    )
929: (8)                    u, v = 10, "k"
930: (8)                    assert _consistent_PT(u, v, gparams, sparams)
931: (8)                    G1.remove_node(6)
932: (8)                    assert _consistent_PT(u, v, gparams, sparams)
933: (8)                    G1.add_edge(u, 2)
934: (8)                    assert not _consistent_PT(u, v, gparams, sparams)
935: (8)                    G2.add_edge(v, "c")
936: (8)                    assert _consistent_PT(u, v, gparams, sparams)
937: (8)                    G2.add_edge(v, "x")
938: (8)                    G1.add_node(7)
939: (8)                    sparams.mapping.update({7: "x"})
940: (8)                    sparams.reverse_mapping.update({"x": 7})
941: (8)                    assert not _consistent_PT(u, v, gparams, sparams)
942: (8)                    G1.add_edge(u, 7)
943: (8)                    assert _consistent_PT(u, v, gparams, sparams)
944: (4)                @pytest.mark.parametrize("graph_type", (nx.Graph, nx.DiGraph))
945: (4)                def test_cut_inconsistent_labels(self, graph_type):
946: (8)                    G1 = graph_type(
947: (12)                        [
948: (16)                            (0, 1),
949: (16)                            (1, 2),
950: (16)                            (10, 0),
951: (16)                            (10, 3),
952: (16)                            (10, 4),
953: (16)                            (10, 5),
954: (16)                            (10, 6),
955: (16)                            (4, 1),
956: (16)                            (5, 3),
957: (12)                        ]
958: (8)                    )
959: (8)                    G2 = graph_type(
960: (12)                        [
961: (16)                            ("a", "b"),
962: (16)                            ("b", "c"),
963: (16)                            ("k", "a"),
964: (16)                            ("k", "d"),
965: (16)                            ("k", "e"),
966: (16)                            ("k", "f"),
967: (16)                            ("k", "g"),
968: (16)                            ("e", "b"),
969: (16)                            ("f", "d"),
970: (12)                        ]
971: (8)                    )
972: (8)                    l1 = {n: "blue" for n in G1.nodes()}
973: (8)                    l2 = {n: "blue" for n in G2.nodes()}
974: (8)                    l1.update({6: "green"})  # Change the label of one neighbor of u
975: (8)                    gparams = _GraphParameters(
976: (12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
977: (8)                    )
978: (8)                    sparams = _StateParameters(
979: (12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
980: (12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
981: (12)                        None,
982: (12)                        None,
983: (12)                        None,
984: (12)                        None,
985: (12)                        None,
986: (12)                        None,
987: (12)                        None,
988: (12)                        None,
```

```
        )
        u, v = 10, "k"
        assert _cut_PT(u, v, gparams, sparams)
    def test_cut_consistent_labels(self):
        G1 = nx.Graph(
            [
                (0, 1),
                (1, 2),
                (10, 0),
                (10, 3),
                (10, 4),
                (10, 5),
                (10, 6),
                (4, 1),
                (5, 3),
            ]
        )
        G2 = nx.Graph(
            [
                ("a", "b"),
                ("b", "c"),
                ("k", "a"),
                ("k", "d"),
                ("k", "e"),
                ("k", "f"),
                ("k", "g"),
                ("e", "b"),
                ("f", "d"),
            ]
        )
        l1 = {n: "blue" for n in G1.nodes()}
        l2 = {n: "blue" for n in G2.nodes()}
        gparams = _GraphParameters(
            G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
        )
        sparams = _StateParameters(
            {0: "a", 1: "b", 2: "c", 3: "d"},
            {"a": 0, "b": 1, "c": 2, "d": 3},
            {4, 5},
            None,
            {6},
            None,
            {"e", "f"},
            None,
            {"g"},
            None,
        )
        u, v = 10, "k"
        assert not _cut_PT(u, v, gparams, sparams)
    def test_cut_same_labels(self):
        G1 = nx.Graph(
            [
                (0, 1),
                (1, 2),
                (10, 0),
                (10, 3),
                (10, 4),
                (10, 5),
                (10, 6),
                (4, 1),
                (5, 3),
            ]
        )
        mapped = {0: "a", 1: "b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g", 10: "k"}
        G2 = nx.relabel_nodes(G1, mapped)
        l1 = {n: "blue" for n in G1.nodes()}
        l2 = {n: "blue" for n in G2.nodes()}
        gparams = _GraphParameters(
```

```
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    {4, 5},
(12)                    None,
(12)                    {6},
(12)                    None,
(12)                    {"e", "f"},
(12)                    None,
(12)                    {"g"},
(12)                    None,
(8)                 )
(8)                 u, v = 10, "k"
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 4)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[4])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[6])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 6)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_nodes_from([6, 7, 8])
(8)                 G2.add_nodes_from(["g", "y", "z"])
(8)                 sparams.T1_tilde.update({6, 7, 8})
(8)                 sparams.T2_tilde.update({"g", "y", "z"})
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 sparams.mapping.update({6: "g", 7: "y"})
(8)                 sparams.reverse_mapping.update({"g": 6, "y": 7})
(8)                 G1.add_edges_from([(6, 20), (7, 20), (6, 21)])
(8)                 G2.add_edges_from([("g", "i"), ("g", "j"), ("y", "j")])
(8)                 sparams.mapping.update({20: "j", 21: "i"})
(8)                 sparams.reverse_mapping.update({"j": 20, "i": 21})
(8)                 sparams.T1.update({20, 21})
(8)                 sparams.T2.update({"i", "j"})
(8)                 sparams.T1_tilde.difference_update({6, 7})
(8)                 sparams.T2_tilde.difference_update({"g", "y"})
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edges_from([(u, 20), (u, 21)])
(8)                 G2.add_edges_from([(v, "i"), (v, "j")])
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 20)
(8)                 G1.add_edge(u, 4)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 8)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, "z")
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(4)             def test_cut_different_labels(self):
(8)                 G1 = nx.Graph(
(12)                    [
(16)                        (0, 1),
(16)                        (1, 2),
(16)                        (1, 14),
(16)                        (0, 4),
(16)                        (1, 5),
(16)                        (2, 6),
```

```
(16)                            (3, 7),
(16)                            (3, 6),
(16)                            (4, 10),
(16)                            (4, 9),
(16)                            (6, 10),
(16)                            (20, 9),
(16)                            (20, 15),
(16)                            (20, 12),
(16)                            (20, 11),
(16)                            (12, 13),
(16)                            (11, 13),
(16)                            (20, 8),
(16)                            (20, 3),
(16)                            (20, 5),
(16)                            (20, 0),
(12)                        ]
(8)                     )
(8)                     mapped = {
(12)                        0: "a",
(12)                        1: "b",
(12)                        2: "c",
(12)                        3: "d",
(12)                        4: "e",
(12)                        5: "f",
(12)                        6: "g",
(12)                        7: "h",
(12)                        8: "i",
(12)                        9: "j",
(12)                        10: "k",
(12)                        11: "l",
(12)                        12: "m",
(12)                        13: "n",
(12)                        14: "o",
(12)                        15: "p",
(12)                        20: "x",
(8)                     }
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     l1 = {n: "none" for n in G1.nodes()}
(8)                     l2 = {}
(8)                     l1.update(
(12)                        {
(16)                            9: "blue",
(16)                            15: "blue",
(16)                            12: "blue",
(16)                            11: "green",
(16)                            3: "green",
(16)                            8: "red",
(16)                            0: "red",
(16)                            5: "yellow",
(12)                        }
(8)                     )
(8)                     l2.update({mapped[n]: l for n, l in l1.items()})
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                        {4, 5, 6, 7, 14},
(12)                        None,
(12)                        {9, 10, 15, 12, 11, 13, 8},
(12)                        None,
(12)                        {"e", "f", "g", "h", "o"},
(12)                        None,
(12)                        {"j", "k", "l", "m", "n", "i", "p"},
(12)                        None,
(8)                     )
(8)                     u, v = 20, "x"
(8)                     assert not _cut_PT(u, v, gparams, sparams)
```

```
(8)                 l1.update({9: "red"})
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 l2.update({mapped[9]: "red"})
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 4)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, mapped[4])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[8])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 8)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(8, 3)
(8)                 G2.add_edge(mapped[8], mapped[3])
(8)                 sparams.T1.add(8)
(8)                 sparams.T2.add(mapped[8])
(8)                 sparams.T1_tilde.remove(8)
(8)                 sparams.T2_tilde.remove(mapped[8])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_node(5)
(8)                 l1.pop(5)
(8)                 sparams.T1.remove(5)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_node(mapped[5])
(8)                 l2.pop(mapped[5])
(8)                 sparams.T2.remove(mapped[5])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(4)             def test_feasibility_same_labels(self):
(8)                 G1 = nx.Graph(
(12)                    [
(16)                        (0, 1),
(16)                        (1, 2),
(16)                        (1, 14),
(16)                        (0, 4),
(16)                        (1, 5),
(16)                        (2, 6),
(16)                        (3, 7),
(16)                        (3, 6),
(16)                        (4, 10),
(16)                        (4, 9),
(16)                        (6, 10),
(16)                        (20, 9),
(16)                        (20, 15),
(16)                        (20, 12),
(16)                        (20, 11),
(16)                        (12, 13),
(16)                        (11, 13),
(16)                        (20, 8),
(16)                        (20, 2),
(16)                        (20, 5),
(16)                        (20, 0),
(12)                    ]
(8)                 )
(8)                 mapped = {
(12)                    0: "a",
(12)                    1: "b",
(12)                    2: "c",
(12)                    3: "d",
(12)                    4: "e",
(12)                    5: "f",
(12)                    6: "g",
(12)                    7: "h",
(12)                    8: "i",
(12)                    9: "j",
(12)                    10: "k",
(12)                    11: "l",
(12)                    12: "m",
(12)                    13: "n",
(12)                    14: "o",
```

```
(12)                        15: "p",
(12)                        20: "x",
(8)                     }
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     l1 = {n: "blue" for n in G1.nodes()}
(8)                     l2 = {mapped[n]: "blue" for n in G1.nodes()}
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                        {4, 5, 6, 7, 14},
(12)                        None,
(12)                        {9, 10, 15, 12, 11, 13, 8},
(12)                        None,
(12)                        {"e", "f", "g", "h", "o"},
(12)                        None,
(12)                        {"j", "k", "l", "m", "n", "i", "p"},
(12)                        None,
(8)                     )
(8)                     u, v = 20, "x"
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G2.remove_edge(mapped[20], mapped[2])
(8)                     G2.add_edge(mapped[20], mapped[3])
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     assert not _consistent_PT(u, v, gparams, sparams)
(8)                     G1.remove_edge(20, 2)
(8)                     G1.add_edge(20, 3)
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(8)                     G2.add_edge(v, mapped[10])
(8)                     assert _cut_PT(u, v, gparams, sparams)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(4)                 def test_feasibility_different_labels(self):
(8)                     G1 = nx.Graph(
(12)                        [
(16)                            (0, 1),
(16)                            (1, 2),
(16)                            (1, 14),
(16)                            (0, 4),
(16)                            (1, 5),
(16)                            (2, 6),
(16)                            (3, 7),
(16)                            (3, 6),
(16)                            (4, 10),
(16)                            (4, 9),
(16)                            (6, 10),
(16)                            (20, 9),
(16)                            (20, 15),
(16)                            (20, 12),
(16)                            (20, 11),
(16)                            (12, 13),
(16)                            (11, 13),
(16)                            (20, 8),
(16)                            (20, 2),
(16)                            (20, 5),
(16)                            (20, 0),
(12)                        ]
(8)                     )
(8)                     mapped = {
(12)                        0: "a",
(12)                        1: "b",
(12)                        2: "c",
(12)                        3: "d",
(12)                        4: "e",
(12)                        5: "f",
(12)                        6: "g",
(12)                        7: "h",
```

```
(12)                        8: "i",
(12)                        9: "j",
(12)                        10: "k",
(12)                        11: "l",
(12)                        12: "m",
(12)                        13: "n",
(12)                        14: "o",
(12)                        15: "p",
(12)                        20: "x",
(8)                     }
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     l1 = {n: "none" for n in G1.nodes()}
(8)                     l2 = {}
(8)                     l1.update(
(12)                        {
(16)                            9: "blue",
(16)                            15: "blue",
(16)                            12: "blue",
(16)                            11: "green",
(16)                            2: "green",
(16)                            8: "red",
(16)                            0: "red",
(16)                            5: "yellow",
(12)                        }
(8)                     )
(8)                     l2.update({mapped[n]: l for n, l in l1.items()})
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                        {4, 5, 6, 7, 14},
(12)                        None,
(12)                        {9, 10, 15, 12, 11, 13, 8},
(12)                        None,
(12)                        {"e", "f", "g", "h", "o"},
(12)                        None,
(12)                        {"j", "k", "l", "m", "n", "i", "p"},
(12)                        None,
(8)                     )
(8)                     u, v = 20, "x"
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G2.remove_edge(mapped[20], mapped[2])
(8)                     G2.add_edge(mapped[20], mapped[3])
(8)                     l2.update({mapped[3]: "green"})
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     assert not _consistent_PT(u, v, gparams, sparams)
(8)                     G1.remove_edge(20, 2)
(8)                     G1.add_edge(20, 3)
(8)                     l1.update({3: "green"})
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(8)                     l1.update({5: "red"})
(8)                     assert _cut_PT(u, v, gparams, sparams)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(0)             class TestMultiGraphISOFeasibility:
(4)                 def test_const_covered_neighbors(self):
(8)                     G1 = nx.MultiGraph(
(12)                        [(0, 1), (0, 1), (1, 2), (3, 0), (3, 0), (3, 0), (3, 2), (3, 2)]
(8)                     )
(8)                     G2 = nx.MultiGraph(
(12)                        [
(16)                            ("a", "b"),
(16)                            ("a", "b"),
(16)                            ("b", "c"),
(16)                            ("k", "a"),
(16)                            ("k", "a"),
(16)                            ("k", "a"),
```

```
(16)                        ("k", "c"),
(16)                        ("k", "c"),
(12)                    ]
(8)                 )
(8)                 gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c"},
(12)                    {"a": 0, "b": 1, "c": 2},
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(8)                 )
(8)                 u, v = 3, "k"
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(4)             def test_const_no_covered_neighbors(self):
(8)                 G1 = nx.MultiGraph([(0, 1), (0, 1), (1, 2), (3, 4), (3, 4), (3, 5)])
(8)                 G2 = nx.MultiGraph([("a", "b"), ("b", "c"), ("k", "w"), ("k", "w"),
("k", "z")])
(8)                 gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c"},
(12)                    {"a": 0, "b": 1, "c": 2},
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(8)                 )
(8)                 u, v = 3, "k"
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(4)             def test_const_mixed_covered_uncovered_neighbors(self):
(8)                 G1 = nx.MultiGraph(
(12)                    [(0, 1), (1, 2), (3, 0), (3, 0), (3, 0), (3, 2), (3, 2), (3, 4),
(3, 5)]
(8)                 )
(8)                 G2 = nx.MultiGraph(
(12)                    [
(16)                        ("a", "b"),
(16)                        ("b", "c"),
(16)                        ("k", "a"),
(16)                        ("k", "a"),
(16)                        ("k", "a"),
(16)                        ("k", "c"),
(16)                        ("k", "c"),
(16)                        ("k", "w"),
(16)                        ("k", "z"),
(12)                    ]
(8)                 )
(8)                 gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c"},
(12)                    {"a": 0, "b": 1, "c": 2},
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(8)                 )
```

```
(8)                 u, v = 3, "k"
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(4)             def test_const_fail_cases(self):
(8)                 G1 = nx.MultiGraph(
(12)                    [
(16)                        (0, 1),
(16)                        (1, 2),
(16)                        (10, 0),
(16)                        (10, 0),
(16)                        (10, 0),
(16)                        (10, 3),
(16)                        (10, 3),
(16)                        (10, 4),
(16)                        (10, 5),
(16)                        (10, 6),
(16)                        (10, 6),
(16)                        (4, 1),
(16)                        (5, 3),
(12)                    ]
(8)                 )
(8)                 mapped = {0: "a", 1: "b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g", 10:
"k"}
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(8)                 )
(8)                 u, v = 10, "k"
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(8)                 G1.remove_node(6)
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 2)
(8)                 assert not _consistent_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, "c")
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, "x")
(8)                 G1.add_node(7)
(8)                 sparams.mapping.update({7: "x"})
(8)                 sparams.reverse_mapping.update({"x": 7})
(8)                 assert not _consistent_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 7)
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(8)                 G1.remove_edges_from([(u, 0), (u, 0)])
(8)                 assert not _consistent_PT(u, v, gparams, sparams)
(8)                 G2.remove_edges_from([(v, mapped[0]), (v, mapped[0])])
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[3])
(8)                 assert not _consistent_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 3)
(8)                 assert _consistent_PT(u, v, gparams, sparams)
(4)             def test_cut_same_labels(self):
(8)                 G1 = nx.MultiGraph(
(12)                    [
(16)                        (0, 1),
(16)                        (1, 2),
(16)                        (10, 0),
(16)                        (10, 0),
(16)                        (10, 0),
(16)                        (10, 3),
(16)                        (10, 3),
```

```
(16)                        (10, 4),
(16)                        (10, 4),
(16)                        (10, 5),
(16)                        (10, 5),
(16)                        (10, 5),
(16)                        (10, 5),
(16)                        (10, 6),
(16)                        (10, 6),
(16)                        (4, 1),
(16)                        (5, 3),
(12)                    ]
(8)                 )
(8)                 mapped = {0: "a", 1: "b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g", 10:
"k"}
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    {4, 5},
(12)                    None,
(12)                    {6},
(12)                    None,
(12)                    {"e", "f"},
(12)                    None,
(12)                    {"g"},
(12)                    None,
(8)                 )
(8)                 u, v = 10, "k"
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 4)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 4)
(8)                 G2.remove_edges_from([(v, mapped[4]), (v, mapped[4])])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[6])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 6)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edges_from([(u, 5), (u, 5), (u, 5)])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.add_edges_from([(v, mapped[5]), (v, mapped[5]), (v, mapped[5])])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_nodes_from([6, 7, 8])
(8)                 G2.add_nodes_from(["g", "y", "z"])
(8)                 G1.add_edges_from([(u, 6), (u, 6), (u, 6), (u, 8)])
(8)                 G2.add_edges_from([(v, "g"), (v, "g"), (v, "g"), (v, "z")])
(8)                 sparams.T1_tilde.update({6, 7, 8})
(8)                 sparams.T2_tilde.update({"g", "y", "z"})
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 sparams.mapping.update({6: "g", 7: "y"})
(8)                 sparams.reverse_mapping.update({"g": 6, "y": 7})
(8)                 G1.add_edges_from([(6, 20), (7, 20), (6, 21)])
(8)                 G2.add_edges_from([("g", "i"), ("g", "j"), ("y", "j")])
(8)                 sparams.T1.update({20, 21})
(8)                 sparams.T2.update({"i", "j"})
(8)                 sparams.T1_tilde.difference_update({6, 7})
(8)                 sparams.T2_tilde.difference_update({"g", "y"})
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, "g")
```

```
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 6)
(8)                 G1.add_edge(u, 8)
(8)                 G2.add_edge(v, "z")
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edges_from([(u, 20), (u, 20), (u, 20), (u, 21)])
(8)                 G2.add_edges_from([(v, "i"), (v, "i"), (v, "i"), (v, "j")])
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 20)
(8)                 G1.add_edge(u, 4)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, "i")
(8)                 G2.add_edge(v, mapped[4])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(4)             def test_cut_different_labels(self):
(8)                 G1 = nx.MultiGraph(
(12)                    [
(16)                        (0, 1),
(16)                        (0, 1),
(16)                        (1, 2),
(16)                        (1, 2),
(16)                        (1, 14),
(16)                        (0, 4),
(16)                        (1, 5),
(16)                        (2, 6),
(16)                        (3, 7),
(16)                        (3, 6),
(16)                        (4, 10),
(16)                        (4, 9),
(16)                        (6, 10),
(16)                        (20, 9),
(16)                        (20, 9),
(16)                        (20, 9),
(16)                        (20, 15),
(16)                        (20, 15),
(16)                        (20, 12),
(16)                        (20, 11),
(16)                        (20, 11),
(16)                        (20, 11),
(16)                        (12, 13),
(16)                        (11, 13),
(16)                        (20, 8),
(16)                        (20, 8),
(16)                        (20, 3),
(16)                        (20, 3),
(16)                        (20, 5),
(16)                        (20, 5),
(16)                        (20, 5),
(16)                        (20, 0),
(16)                        (20, 0),
(16)                        (20, 0),
(12)                    ]
(8)                 )
(8)                 mapped = {
(12)                    0: "a",
(12)                    1: "b",
(12)                    2: "c",
(12)                    3: "d",
(12)                    4: "e",
(12)                    5: "f",
(12)                    6: "g",
(12)                    7: "h",
(12)                    8: "i",
(12)                    9: "j",
```

```
(12)                    10: "k",
(12)                    11: "l",
(12)                    12: "m",
(12)                    13: "n",
(12)                    14: "o",
(12)                    15: "p",
(12)                    20: "x",
(8)                 }
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 l1 = {n: "none" for n in G1.nodes()}
(8)                 l2 = {}
(8)                 l1.update(
(12)                    {
(16)                        9: "blue",
(16)                        15: "blue",
(16)                        12: "blue",
(16)                        11: "green",
(16)                        3: "green",
(16)                        8: "red",
(16)                        0: "red",
(16)                        5: "yellow",
(12)                    }
(8)                 )
(8)                 l2.update({mapped[n]: l for n, l in l1.items()})
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    {4, 5, 6, 7, 14},
(12)                    None,
(12)                    {9, 10, 15, 12, 11, 13, 8},
(12)                    None,
(12)                    {"e", "f", "g", "h", "o"},
(12)                    None,
(12)                    {"j", "k", "l", "m", "n", "i", "p"},
(12)                    None,
(8)                 )
(8)                 u, v = 20, "x"
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 l1.update({9: "red"})
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 l2.update({mapped[9]: "red"})
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 4)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, mapped[4])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[8])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 8)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edges_from([(8, 3), (8, 3), (8, u)])
(8)                 G2.add_edges_from([(mapped[8], mapped[3]), (mapped[8], mapped[3])])
(8)                 sparams.T1.add(8)
(8)                 sparams.T2.add(mapped[8])
(8)                 sparams.T1_tilde.remove(8)
(8)                 sparams.T2_tilde.remove(mapped[8])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(8, u)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_node(5)
(8)                 l1.pop(5)
(8)                 sparams.T1.remove(5)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_node(mapped[5])
(8)                 l2.pop(mapped[5])
(8)                 sparams.T2.remove(mapped[5])
```

```
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(4)             def test_feasibility_same_labels(self):
(8)                 G1 = nx.MultiGraph(
(12)                    [
(16)                        (0, 1),
(16)                        (0, 1),
(16)                        (1, 2),
(16)                        (1, 2),
(16)                        (1, 14),
(16)                        (0, 4),
(16)                        (1, 5),
(16)                        (2, 6),
(16)                        (3, 7),
(16)                        (3, 6),
(16)                        (4, 10),
(16)                        (4, 9),
(16)                        (6, 10),
(16)                        (20, 9),
(16)                        (20, 9),
(16)                        (20, 9),
(16)                        (20, 15),
(16)                        (20, 15),
(16)                        (20, 12),
(16)                        (20, 11),
(16)                        (20, 11),
(16)                        (20, 11),
(16)                        (12, 13),
(16)                        (11, 13),
(16)                        (20, 8),
(16)                        (20, 8),
(16)                        (20, 3),
(16)                        (20, 3),
(16)                        (20, 5),
(16)                        (20, 5),
(16)                        (20, 5),
(16)                        (20, 0),
(16)                        (20, 0),
(16)                        (20, 0),
(12)                    ]
(8)                 )
(8)                 mapped = {
(12)                    0: "a",
(12)                    1: "b",
(12)                    2: "c",
(12)                    3: "d",
(12)                    4: "e",
(12)                    5: "f",
(12)                    6: "g",
(12)                    7: "h",
(12)                    8: "i",
(12)                    9: "j",
(12)                    10: "k",
(12)                    11: "l",
(12)                    12: "m",
(12)                    13: "n",
(12)                    14: "o",
(12)                    15: "p",
(12)                    20: "x",
(8)                 }
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {mapped[n]: "blue" for n in G1.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    {4, 5, 6, 7, 14},
```

```
1812: (12)                      None,
1813: (12)                      {9, 10, 15, 12, 11, 13, 8},
1814: (12)                      None,
1815: (12)                      {"e", "f", "g", "h", "o"},
1816: (12)                      None,
1817: (12)                      {"j", "k", "l", "m", "n", "i", "p"},
1818: (12)                      None,
1819: (8)                   )
1820: (8)                   u, v = 20, "x"
1821: (8)                   assert not _cut_PT(u, v, gparams, sparams)
1822: (8)                   G2.remove_edges_from([(mapped[20], mapped[3]), (mapped[20],
mapped[3])])
1823: (8)                   G2.add_edges_from([(mapped[20], mapped[2]), (mapped[20], mapped[2])])
1824: (8)                   assert not _cut_PT(u, v, gparams, sparams)
1825: (8)                   assert not _consistent_PT(u, v, gparams, sparams)
1826: (8)                   G1.remove_edges_from([(20, 3), (20, 3)])
1827: (8)                   G1.add_edges_from([(20, 2), (20, 2)])
1828: (8)                   assert not _cut_PT(u, v, gparams, sparams)
1829: (8)                   assert _consistent_PT(u, v, gparams, sparams)
1830: (8)                   G2.add_edges_from([(v, mapped[10])] * 5)
1831: (8)                   assert _cut_PT(u, v, gparams, sparams)
1832: (8)                   assert _consistent_PT(u, v, gparams, sparams)
1833: (8)                   G1.add_edges_from([(u, 10)] * 5)
1834: (8)                   assert not _cut_PT(u, v, gparams, sparams)
1835: (8)                   assert _consistent_PT(u, v, gparams, sparams)
1836: (4)               def test_feasibility_different_labels(self):
1837: (8)                   G1 = nx.MultiGraph(
1838: (12)                      [
1839: (16)                          (0, 1),
1840: (16)                          (0, 1),
1841: (16)                          (1, 2),
1842: (16)                          (1, 2),
1843: (16)                          (1, 14),
1844: (16)                          (0, 4),
1845: (16)                          (1, 5),
1846: (16)                          (2, 6),
1847: (16)                          (3, 7),
1848: (16)                          (3, 6),
1849: (16)                          (4, 10),
1850: (16)                          (4, 9),
1851: (16)                          (6, 10),
1852: (16)                          (20, 9),
1853: (16)                          (20, 9),
1854: (16)                          (20, 9),
1855: (16)                          (20, 15),
1856: (16)                          (20, 15),
1857: (16)                          (20, 12),
1858: (16)                          (20, 11),
1859: (16)                          (20, 11),
1860: (16)                          (20, 11),
1861: (16)                          (12, 13),
1862: (16)                          (11, 13),
1863: (16)                          (20, 8),
1864: (16)                          (20, 8),
1865: (16)                          (20, 2),
1866: (16)                          (20, 2),
1867: (16)                          (20, 5),
1868: (16)                          (20, 5),
1869: (16)                          (20, 5),
1870: (16)                          (20, 0),
1871: (16)                          (20, 0),
1872: (16)                          (20, 0),
1873: (12)                      ]
1874: (8)                   )
1875: (8)                   mapped = {
1876: (12)                      0: "a",
1877: (12)                      1: "b",
1878: (12)                      2: "c",
1879: (12)                      3: "d",
```

```
(12)                        4: "e",
(12)                        5: "f",
(12)                        6: "g",
(12)                        7: "h",
(12)                        8: "i",
(12)                        9: "j",
(12)                        10: "k",
(12)                        11: "l",
(12)                        12: "m",
(12)                        13: "n",
(12)                        14: "o",
(12)                        15: "p",
(12)                        20: "x",
(8)                     }
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     l1 = {n: "none" for n in G1.nodes()}
(8)                     l2 = {}
(8)                     l1.update(
(12)                        {
(16)                            9: "blue",
(16)                            15: "blue",
(16)                            12: "blue",
(16)                            11: "green",
(16)                            2: "green",
(16)                            8: "red",
(16)                            0: "red",
(16)                            5: "yellow",
(12)                        }
(8)                     )
(8)                     l2.update({mapped[n]: l for n, l in l1.items()})
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                        {4, 5, 6, 7, 14},
(12)                        None,
(12)                        {9, 10, 15, 12, 11, 13, 8},
(12)                        None,
(12)                        {"e", "f", "g", "h", "o"},
(12)                        None,
(12)                        {"j", "k", "l", "m", "n", "i", "p"},
(12)                        None,
(8)                     )
(8)                     u, v = 20, "x"
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G2.remove_edges_from([(mapped[20], mapped[2]), (mapped[20],
mapped[2])])
(8)                     G2.add_edges_from([(mapped[20], mapped[3]), (mapped[20], mapped[3])])
(8)                     l2.update({mapped[3]: "green"})
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     assert not _consistent_PT(u, v, gparams, sparams)
(8)                     G1.remove_edges_from([(20, 2), (20, 2)])
(8)                     G1.add_edges_from([(20, 3), (20, 3)])
(8)                     l1.update({3: "green"})
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(8)                     l1.update({5: "red"})
(8)                     assert _cut_PT(u, v, gparams, sparams)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(0)             class TestDiGraphISOFeasibility:
(4)                 def test_const_covered_neighbors(self):
(8)                     G1 = nx.DiGraph([(0, 1), (1, 2), (0, 3), (2, 3)])
(8)                     G2 = nx.DiGraph([("a", "b"), ("b", "c"), ("a", "k"), ("c", "k")])
(8)                     gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c"},
(12)                        {"a": 0, "b": 1, "c": 2},
```

```
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(8)                     )
(8)                     u, v = 3, "k"
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(4)                 def test_const_no_covered_neighbors(self):
(8)                     G1 = nx.DiGraph([(0, 1), (1, 2), (3, 4), (3, 5)])
(8)                     G2 = nx.DiGraph([("a", "b"), ("b", "c"), ("k", "w"), ("k", "z")])
(8)                     gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c"},
(12)                        {"a": 0, "b": 1, "c": 2},
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(8)                     )
(8)                     u, v = 3, "k"
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(4)                 def test_const_mixed_covered_uncovered_neighbors(self):
(8)                     G1 = nx.DiGraph([(0, 1), (1, 2), (3, 0), (3, 2), (3, 4), (3, 5)])
(8)                     G2 = nx.DiGraph(
(12)                        [("a", "b"), ("b", "c"), ("k", "a"), ("k", "c"), ("k", "w"), ("k",
"z")]
(8)                     )
(8)                     gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c"},
(12)                        {"a": 0, "b": 1, "c": 2},
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(8)                     )
(8)                     u, v = 3, "k"
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(4)                 def test_const_fail_cases(self):
(8)                     G1 = nx.DiGraph(
(12)                        [
(16)                            (0, 1),
(16)                            (2, 1),
(16)                            (10, 0),
(16)                            (10, 3),
(16)                            (10, 4),
(16)                            (5, 10),
(16)                            (10, 6),
(16)                            (1, 4),
(16)                            (5, 3),
(12)                        ]
(8)                     )
(8)                     G2 = nx.DiGraph(
(12)                        [
(16)                            ("a", "b"),
(16)                            ("c", "b"),
(16)                            ("k", "a"),
```

```
(16)                            ("k", "d"),
(16)                            ("k", "e"),
(16)                            ("f", "k"),
(16)                            ("k", "g"),
(16)                            ("b", "e"),
(16)                            ("f", "d"),
(12)                        ]
(8)                     )
(8)                     gparams = _GraphParameters(G1, G2, None, None, None, None, None)
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(12)                        None,
(8)                     )
(8)                     u, v = 10, "k"
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(8)                     G1.remove_node(6)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(8)                     G1.add_edge(u, 2)
(8)                     assert not _consistent_PT(u, v, gparams, sparams)
(8)                     G2.add_edge(v, "c")
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(8)                     G2.add_edge(v, "x")
(8)                     G1.add_node(7)
(8)                     sparams.mapping.update({7: "x"})
(8)                     sparams.reverse_mapping.update({"x": 7})
(8)                     assert not _consistent_PT(u, v, gparams, sparams)
(8)                     G1.add_edge(u, 7)
(8)                     assert _consistent_PT(u, v, gparams, sparams)
(4)                 def test_cut_inconsistent_labels(self):
(8)                     G1 = nx.DiGraph(
(12)                        [
(16)                            (0, 1),
(16)                            (2, 1),
(16)                            (10, 0),
(16)                            (10, 3),
(16)                            (10, 4),
(16)                            (5, 10),
(16)                            (10, 6),
(16)                            (1, 4),
(16)                            (5, 3),
(12)                        ]
(8)                     )
(8)                     G2 = nx.DiGraph(
(12)                        [
(16)                            ("a", "b"),
(16)                            ("c", "b"),
(16)                            ("k", "a"),
(16)                            ("k", "d"),
(16)                            ("k", "e"),
(16)                            ("f", "k"),
(16)                            ("k", "g"),
(16)                            ("b", "e"),
(16)                            ("f", "d"),
(12)                        ]
(8)                     )
(8)                     l1 = {n: "blue" for n in G1.nodes()}
(8)                     l2 = {n: "blue" for n in G2.nodes()}
(8)                     l1.update({5: "green"})  # Change the label of one neighbor of u
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
```

```
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(12)                    None,
(8)                 )
(8)                 u, v = 10, "k"
(8)                 assert _cut_PT(u, v, gparams, sparams)
(4)             def test_cut_consistent_labels(self):
(8)                 G1 = nx.DiGraph(
(12)                    [
(16)                        (0, 1),
(16)                        (2, 1),
(16)                        (10, 0),
(16)                        (10, 3),
(16)                        (10, 4),
(16)                        (5, 10),
(16)                        (10, 6),
(16)                        (1, 4),
(16)                        (5, 3),
(12)                    ]
(8)                 )
(8)                 G2 = nx.DiGraph(
(12)                    [
(16)                        ("a", "b"),
(16)                        ("c", "b"),
(16)                        ("k", "a"),
(16)                        ("k", "d"),
(16)                        ("k", "e"),
(16)                        ("f", "k"),
(16)                        ("k", "g"),
(16)                        ("b", "e"),
(16)                        ("f", "d"),
(12)                    ]
(8)                 )
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    {4},
(12)                    {5, 10},
(12)                    {6},
(12)                    None,
(12)                    {"e"},
(12)                    {"f", "k"},
(12)                    {"g"},
(12)                    None,
(8)                 )
(8)                 u, v = 10, "k"
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(4)             def test_cut_same_labels(self):
(8)                 G1 = nx.DiGraph(
(12)                    [
(16)                        (0, 1),
(16)                        (2, 1),
(16)                        (10, 0),
(16)                        (10, 3),
(16)                        (10, 4),
(16)                        (5, 10),
```

```
(16)                            (10, 6),
(16)                            (1, 4),
(16)                            (5, 3),
(12)                        ]
(8)                     )
(8)                     mapped = {0: "a", 1: "b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g", 10:
"k"}
(8)                     G2 = nx.relabel_nodes(G1, mapped)
(8)                     l1 = {n: "blue" for n in G1.nodes()}
(8)                     l2 = {n: "blue" for n in G2.nodes()}
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
(8)                     sparams = _StateParameters(
(12)                        {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                        {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                        {4},
(12)                        {5, 10},
(12)                        {6},
(12)                        None,
(12)                        {"e"},
(12)                        {"f", "k"},
(12)                        {"g"},
(12)                        None,
(8)                     )
(8)                     u, v = 10, "k"
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G1.remove_edge(u, 4)
(8)                     assert _cut_PT(u, v, gparams, sparams)
(8)                     G2.remove_edge(v, mapped[4])
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G1.remove_edge(5, u)
(8)                     assert _cut_PT(u, v, gparams, sparams)
(8)                     G2.remove_edge(mapped[5], v)
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G2.remove_edge(v, mapped[6])
(8)                     assert _cut_PT(u, v, gparams, sparams)
(8)                     G1.remove_edge(u, 6)
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(8)                     G1.add_nodes_from([6, 7, 8])
(8)                     G2.add_nodes_from(["g", "y", "z"])
(8)                     sparams.T1_tilde.update({6, 7, 8})
(8)                     sparams.T2_tilde.update({"g", "y", "z"})
(8)                     l1 = {n: "blue" for n in G1.nodes()}
(8)                     l2 = {n: "blue" for n in G2.nodes()}
(8)                     gparams = _GraphParameters(
(12)                        G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                     )
(8)                     assert not _cut_PT(u, v, gparams, sparams)
(4)                 def test_cut_different_labels(self):
(8)                     G1 = nx.DiGraph(
(12)                        [
(16)                            (0, 1),
(16)                            (1, 2),
(16)                            (14, 1),
(16)                            (0, 4),
(16)                            (1, 5),
(16)                            (2, 6),
(16)                            (3, 7),
(16)                            (3, 6),
(16)                            (10, 4),
(16)                            (4, 9),
(16)                            (6, 10),
(16)                            (20, 9),
(16)                            (20, 15),
(16)                            (20, 12),
(16)                            (20, 11),
(16)                            (12, 13),
(16)                            (11, 13),
```

```
(16)                        (20, 8),
(16)                        (20, 3),
(16)                        (20, 5),
(16)                        (0, 20),
(12)                    ]
(8)                 )
(8)                 mapped = {
(12)                    0: "a",
(12)                    1: "b",
(12)                    2: "c",
(12)                    3: "d",
(12)                    4: "e",
(12)                    5: "f",
(12)                    6: "g",
(12)                    7: "h",
(12)                    8: "i",
(12)                    9: "j",
(12)                    10: "k",
(12)                    11: "l",
(12)                    12: "m",
(12)                    13: "n",
(12)                    14: "o",
(12)                    15: "p",
(12)                    20: "x",
(8)                 }
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 l1 = {n: "none" for n in G1.nodes()}
(8)                 l2 = {}
(8)                 l1.update(
(12)                    {
(16)                        9: "blue",
(16)                        15: "blue",
(16)                        12: "blue",
(16)                        11: "green",
(16)                        3: "green",
(16)                        8: "red",
(16)                        0: "red",
(16)                        5: "yellow",
(12)                    }
(8)                 )
(8)                 l2.update({mapped[n]: l for n, l in l1.items()})
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c", 3: "d"},
(12)                    {"a": 0, "b": 1, "c": 2, "d": 3},
(12)                    {4, 5, 6, 7, 20},
(12)                    {14, 20},
(12)                    {9, 10, 15, 12, 11, 13, 8},
(12)                    None,
(12)                    {"e", "f", "g", "x"},
(12)                    {"o", "x"},
(12)                    {"j", "k", "l", "m", "n", "i", "p"},
(12)                    None,
(8)                 )
(8)                 u, v = 20, "x"
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 l1.update({9: "red"})
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 l2.update({mapped[9]: "red"})
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 4)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, mapped[4])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(u, 14)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.add_edge(v, mapped[14])
```

```
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_edge(v, mapped[8])
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_edge(u, 8)
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.add_edge(8, 3)
(8)                 G2.add_edge(mapped[8], mapped[3])
(8)                 sparams.T1.add(8)
(8)                 sparams.T2.add(mapped[8])
(8)                 sparams.T1_tilde.remove(8)
(8)                 sparams.T2_tilde.remove(mapped[8])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(8)                 G1.remove_node(5)
(8)                 l1.pop(5)
(8)                 sparams.T1.remove(5)
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 G2.remove_node(mapped[5])
(8)                 l2.pop(mapped[5])
(8)                 sparams.T2.remove(mapped[5])
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(4)             def test_predecessor_T1_in_fail(self):
(8)                 G1 = nx.DiGraph(
(12)                    [(0, 1), (0, 3), (4, 0), (1, 5), (5, 2), (3, 6), (4, 6), (6, 5)]
(8)                 )
(8)                 mapped = {0: "a", 1: "b", 2: "c", 3: "d", 4: "e", 5: "f", 6: "g"}
(8)                 G2 = nx.relabel_nodes(G1, mapped)
(8)                 l1 = {n: "blue" for n in G1.nodes()}
(8)                 l2 = {n: "blue" for n in G2.nodes()}
(8)                 gparams = _GraphParameters(
(12)                    G1, G2, l1, l2, nx.utils.groups(l1), nx.utils.groups(l2), None
(8)                 )
(8)                 sparams = _StateParameters(
(12)                    {0: "a", 1: "b", 2: "c"},
(12)                    {"a": 0, "b": 1, "c": 2},
(12)                    {3, 5},
(12)                    {4, 5},
(12)                    {6},
(12)                    None,
(12)                    {"d", "f"},
(12)                    {"f"},  # mapped[4] is missing from T2_in
(12)                    {"g"},
(12)                    None,
(8)                 )
(8)                 u, v = 6, "g"
(8)                 assert _cut_PT(u, v, gparams, sparams)
(8)                 sparams.T2_in.add("e")
(8)                 assert not _cut_PT(u, v, gparams, sparams)
(0)         class TestGraphTinoutUpdating:
(4)             edges = [
(8)                 (1, 3),
(8)                 (2, 3),
(8)                 (3, 4),
(8)                 (4, 9),
(8)                 (4, 5),
(8)                 (3, 9),
(8)                 (5, 8),
(8)                 (5, 7),
(8)                 (8, 7),
(8)                 (6, 7),
(4)             ]
(4)             mapped = {
(8)                 0: "x",
(8)                 1: "a",
(8)                 2: "b",
(8)                 3: "c",
(8)                 4: "d",
(8)                 5: "e",
(8)                 6: "f",
(8)                 7: "g",
```

```
(8)                 8: "h",
(8)                 9: "i",
(4)             }
(4)             G1 = nx.Graph()
(4)             G1.add_edges_from(edges)
(4)             G1.add_node(0)
(4)             G2 = nx.relabel_nodes(G1, mapping=mapped)
(4)             def test_updating(self):
(8)                 G2_degree = dict(self.G2.degree)
(8)                 gparams, sparams = _initialize_parameters(self.G1, self.G2, G2_degree)
(8)                 m, m_rev, T1, _, T1_tilde, _, T2, _, T2_tilde, _ = sparams
(8)                 m[4] = self.mapped[4]
(8)                 m_rev[self.mapped[4]] = 4
(8)                 _update_Tinout(4, self.mapped[4], gparams, sparams)
(8)                 assert T1 == {3, 5, 9}
(8)                 assert T2 == {"c", "i", "e"}
(8)                 assert T1_tilde == {0, 1, 2, 6, 7, 8}
(8)                 assert T2_tilde == {"x", "a", "b", "f", "g", "h"}
(8)                 m[5] = self.mapped[5]
(8)                 m_rev.update({self.mapped[5]: 5})
(8)                 _update_Tinout(5, self.mapped[5], gparams, sparams)
(8)                 assert T1 == {3, 9, 8, 7}
(8)                 assert T2 == {"c", "i", "h", "g"}
(8)                 assert T1_tilde == {0, 1, 2, 6}
(8)                 assert T2_tilde == {"x", "a", "b", "f"}
(8)                 m[6] = self.mapped[6]
(8)                 m_rev.update({self.mapped[6]: 6})
(8)                 _update_Tinout(6, self.mapped[6], gparams, sparams)
(8)                 assert T1 == {3, 9, 8, 7}
(8)                 assert T2 == {"c", "i", "h", "g"}
(8)                 assert T1_tilde == {0, 1, 2}
(8)                 assert T2_tilde == {"x", "a", "b"}
(8)                 m[3] = self.mapped[3]
(8)                 m_rev.update({self.mapped[3]: 3})
(8)                 _update_Tinout(3, self.mapped[3], gparams, sparams)
(8)                 assert T1 == {1, 2, 9, 8, 7}
(8)                 assert T2 == {"a", "b", "i", "h", "g"}
(8)                 assert T1_tilde == {0}
(8)                 assert T2_tilde == {"x"}
(8)                 m[0] = self.mapped[0]
(8)                 m_rev.update({self.mapped[0]: 0})
(8)                 _update_Tinout(0, self.mapped[0], gparams, sparams)
(8)                 assert T1 == {1, 2, 9, 8, 7}
(8)                 assert T2 == {"a", "b", "i", "h", "g"}
(8)                 assert T1_tilde == set()
(8)                 assert T2_tilde == set()
(4)             def test_restoring(self):
(8)                 m = {0: "x", 3: "c", 4: "d", 5: "e", 6: "f"}
(8)                 m_rev = {"x": 0, "c": 3, "d": 4, "e": 5, "f": 6}
(8)                 T1 = {1, 2, 7, 9, 8}
(8)                 T2 = {"a", "b", "g", "i", "h"}
(8)                 T1_tilde = set()
(8)                 T2_tilde = set()
(8)                 gparams = _GraphParameters(self.G1, self.G2, {}, {}, {}, {}, {})
(8)                 sparams = _StateParameters(
(12)                    m, m_rev, T1, None, T1_tilde, None, T2, None, T2_tilde, None
(8)                 )
(8)                 m.pop(0)
(8)                 m_rev.pop("x")
(8)                 _restore_Tinout(0, self.mapped[0], gparams, sparams)
(8)                 assert T1 == {1, 2, 7, 9, 8}
(8)                 assert T2 == {"a", "b", "g", "i", "h"}
(8)                 assert T1_tilde == {0}
(8)                 assert T2_tilde == {"x"}
(8)                 m.pop(6)
(8)                 m_rev.pop("f")
(8)                 _restore_Tinout(6, self.mapped[6], gparams, sparams)
(8)                 assert T1 == {1, 2, 7, 9, 8}
(8)                 assert T2 == {"a", "b", "g", "i", "h"}
```

```
2429: (8)                   assert T1_tilde == {0, 6}
2430: (8)                   assert T2_tilde == {"x", "f"}
2431: (8)                   m.pop(3)
2432: (8)                   m_rev.pop("c")
2433: (8)                   _restore_Tinout(3, self.mapped[3], gparams, sparams)
2434: (8)                   assert T1 == {7, 9, 8, 3}
2435: (8)                   assert T2 == {"g", "i", "h", "c"}
2436: (8)                   assert T1_tilde == {0, 6, 1, 2}
2437: (8)                   assert T2_tilde == {"x", "f", "a", "b"}
2438: (8)                   m.pop(5)
2439: (8)                   m_rev.pop("e")
2440: (8)                   _restore_Tinout(5, self.mapped[5], gparams, sparams)
2441: (8)                   assert T1 == {9, 3, 5}
2442: (8)                   assert T2 == {"i", "c", "e"}
2443: (8)                   assert T1_tilde == {0, 6, 1, 2, 7, 8}
2444: (8)                   assert T2_tilde == {"x", "f", "a", "b", "g", "h"}
2445: (8)                   m.pop(4)
2446: (8)                   m_rev.pop("d")
2447: (8)                   _restore_Tinout(4, self.mapped[4], gparams, sparams)
2448: (8)                   assert T1 == set()
2449: (8)                   assert T2 == set()
2450: (8)                   assert T1_tilde == set(self.G1.nodes())
2451: (8)                   assert T2_tilde == set(self.G2.nodes())
2452: (0)           class TestDiGraphTinoutUpdating:
2453: (4)               edges = [
2454: (8)                   (1, 3),
2455: (8)                   (3, 2),
2456: (8)                   (3, 4),
2457: (8)                   (4, 9),
2458: (8)                   (4, 5),
2459: (8)                   (3, 9),
2460: (8)                   (5, 8),
2461: (8)                   (5, 7),
2462: (8)                   (8, 7),
2463: (8)                   (7, 6),
2464: (4)               ]
2465: (4)               mapped = {
2466: (8)                   0: "x",
2467: (8)                   1: "a",
2468: (8)                   2: "b",
2469: (8)                   3: "c",
2470: (8)                   4: "d",
2471: (8)                   5: "e",
2472: (8)                   6: "f",
2473: (8)                   7: "g",
2474: (8)                   8: "h",
2475: (8)                   9: "i",
2476: (4)               }
2477: (4)               G1 = nx.DiGraph(edges)
2478: (4)               G1.add_node(0)
2479: (4)               G2 = nx.relabel_nodes(G1, mapping=mapped)
2480: (4)               def test_updating(self):
2481: (8)                   G2_degree = {
2482: (12)                      n: (in_degree, out_degree)
2483: (12)                      for (n, in_degree), (_, out_degree) in zip(
2484: (16)                          self.G2.in_degree, self.G2.out_degree
2485: (12)                      )
2486: (8)                   }
2487: (8)                   gparams, sparams = _initialize_parameters(self.G1, self.G2, G2_degree)
2488: (8)                   m, m_rev, T1_out, T1_in, T1_tilde, _, T2_out, T2_in, T2_tilde, _ =
sparams
2489: (8)                   m[4] = self.mapped[4]
2490: (8)                   m_rev[self.mapped[4]] = 4
2491: (8)                   _update_Tinout(4, self.mapped[4], gparams, sparams)
2492: (8)                   assert T1_out == {5, 9}
2493: (8)                   assert T1_in == {3}
2494: (8)                   assert T2_out == {"i", "e"}
2495: (8)                   assert T2_in == {"c"}
2496: (8)                   assert T1_tilde == {0, 1, 2, 6, 7, 8}
```

```
(8)                 assert T2_tilde == {"x", "a", "b", "f", "g", "h"}
(8)                 m[5] = self.mapped[5]
(8)                 m_rev[self.mapped[5]] = 5
(8)                 _update_Tinout(5, self.mapped[5], gparams, sparams)
(8)                 assert T1_out == {9, 8, 7}
(8)                 assert T1_in == {3}
(8)                 assert T2_out == {"i", "g", "h"}
(8)                 assert T2_in == {"c"}
(8)                 assert T1_tilde == {0, 1, 2, 6}
(8)                 assert T2_tilde == {"x", "a", "b", "f"}
(8)                 m[6] = self.mapped[6]
(8)                 m_rev[self.mapped[6]] = 6
(8)                 _update_Tinout(6, self.mapped[6], gparams, sparams)
(8)                 assert T1_out == {9, 8, 7}
(8)                 assert T1_in == {3, 7}
(8)                 assert T2_out == {"i", "g", "h"}
(8)                 assert T2_in == {"c", "g"}
(8)                 assert T1_tilde == {0, 1, 2}
(8)                 assert T2_tilde == {"x", "a", "b"}
(8)                 m[3] = self.mapped[3]
(8)                 m_rev[self.mapped[3]] = 3
(8)                 _update_Tinout(3, self.mapped[3], gparams, sparams)
(8)                 assert T1_out == {9, 8, 7, 2}
(8)                 assert T1_in == {7, 1}
(8)                 assert T2_out == {"i", "g", "h", "b"}
(8)                 assert T2_in == {"g", "a"}
(8)                 assert T1_tilde == {0}
(8)                 assert T2_tilde == {"x"}
(8)                 m[0] = self.mapped[0]
(8)                 m_rev[self.mapped[0]] = 0
(8)                 _update_Tinout(0, self.mapped[0], gparams, sparams)
(8)                 assert T1_out == {9, 8, 7, 2}
(8)                 assert T1_in == {7, 1}
(8)                 assert T2_out == {"i", "g", "h", "b"}
(8)                 assert T2_in == {"g", "a"}
(8)                 assert T1_tilde == set()
(8)                 assert T2_tilde == set()
(4)             def test_restoring(self):
(8)                 m = {0: "x", 3: "c", 4: "d", 5: "e", 6: "f"}
(8)                 m_rev = {"x": 0, "c": 3, "d": 4, "e": 5, "f": 6}
(8)                 T1_out = {2, 7, 9, 8}
(8)                 T1_in = {1, 7}
(8)                 T2_out = {"b", "g", "i", "h"}
(8)                 T2_in = {"a", "g"}
(8)                 T1_tilde = set()
(8)                 T2_tilde = set()
(8)                 gparams = _GraphParameters(self.G1, self.G2, {}, {}, {}, {}, {})
(8)                 sparams = _StateParameters(
(12)                    m, m_rev, T1_out, T1_in, T1_tilde, None, T2_out, T2_in, T2_tilde,
None
(8)                 )
(8)                 m.pop(0)
(8)                 m_rev.pop("x")
(8)                 _restore_Tinout_Di(0, self.mapped[0], gparams, sparams)
(8)                 assert T1_out == {2, 7, 9, 8}
(8)                 assert T1_in == {1, 7}
(8)                 assert T2_out == {"b", "g", "i", "h"}
(8)                 assert T2_in == {"a", "g"}
(8)                 assert T1_tilde == {0}
(8)                 assert T2_tilde == {"x"}
(8)                 m.pop(6)
(8)                 m_rev.pop("f")
(8)                 _restore_Tinout_Di(6, self.mapped[6], gparams, sparams)
(8)                 assert T1_out == {2, 9, 8, 7}
(8)                 assert T1_in == {1}
(8)                 assert T2_out == {"b", "i", "h", "g"}
(8)                 assert T2_in == {"a"}
(8)                 assert T1_tilde == {0, 6}
(8)                 assert T2_tilde == {"x", "f"}
```

```
2565: (8)                    m.pop(3)
2566: (8)                    m_rev.pop("c")
2567: (8)                    _restore_Tinout_Di(3, self.mapped[3], gparams, sparams)
2568: (8)                    assert T1_out == {9, 8, 7}
2569: (8)                    assert T1_in == {3}
2570: (8)                    assert T2_out == {"i", "h", "g"}
2571: (8)                    assert T2_in == {"c"}
2572: (8)                    assert T1_tilde == {0, 6, 1, 2}
2573: (8)                    assert T2_tilde == {"x", "f", "a", "b"}
2574: (8)                    m.pop(5)
2575: (8)                    m_rev.pop("e")
2576: (8)                    _restore_Tinout_Di(5, self.mapped[5], gparams, sparams)
2577: (8)                    assert T1_out == {9, 5}
2578: (8)                    assert T1_in == {3}
2579: (8)                    assert T2_out == {"i", "e"}
2580: (8)                    assert T2_in == {"c"}
2581: (8)                    assert T1_tilde == {0, 6, 1, 2, 8, 7}
2582: (8)                    assert T2_tilde == {"x", "f", "a", "b", "h", "g"}
2583: (8)                    m.pop(4)
2584: (8)                    m_rev.pop("d")
2585: (8)                    _restore_Tinout_Di(4, self.mapped[4], gparams, sparams)
2586: (8)                    assert T1_out == set()
2587: (8)                    assert T1_in == set()
2588: (8)                    assert T2_out == set()
2589: (8)                    assert T2_in == set()
2590: (8)                    assert T1_tilde == set(self.G1.nodes())
2591: (8)                    assert T2_tilde == set(self.G2.nodes())

----------------------------------------

File 272 - test_vf2userfunc.py:

1: (0)              """
2: (4)                  Tests for VF2 isomorphism algorithm for weighted graphs.
3: (0)              """
4: (0)              import math
5: (0)              from operator import eq
6: (0)              import networkx as nx
7: (0)              import networkx.algorithms.isomorphism as iso
8: (0)              def test_simple():
9: (4)                  w = "weight"
10: (4)                 edges = [(0, 0, 1), (0, 0, 1.5), (0, 1, 2), (1, 0, 3)]
11: (4)                 for g1 in [nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph()]:
12: (8)                     g1.add_weighted_edges_from(edges)
13: (8)                     g2 = g1.subgraph(g1.nodes())
14: (8)                     if g1.is_multigraph():
15: (12)                        em = iso.numerical_multiedge_match("weight", 1)
16: (8)                     else:
17: (12)                        em = iso.numerical_edge_match("weight", 1)
18: (8)                     assert nx.is_isomorphic(g1, g2, edge_match=em)
19: (8)                     for mod1, mod2 in [(False, True), (True, False), (True, True)]:
20: (12)                        if g2.is_multigraph():
21: (16)                            if mod1:
22: (20)                                data1 = {0: {"weight": 10}}
23: (16)                            if mod2:
24: (20)                                data2 = {0: {"weight": 1}, 1: {"weight": 2.5}}
25: (12)                        else:
26: (16)                            if mod1:
27: (20)                                data1 = {"weight": 10}
28: (16)                            if mod2:
29: (20)                                data2 = {"weight": 2.5}
30: (12)                        g2 = g1.subgraph(g1.nodes()).copy()
31: (12)                        if mod1:
32: (16)                            if not g1.is_directed():
33: (20)                                g2._adj[1][0] = data1
34: (20)                                g2._adj[0][1] = data1
35: (16)                            else:
36: (20)                                g2._succ[1][0] = data1
37: (20)                                g2._pred[0][1] = data1
```

```
(12)                        if mod2:
(16)                            if not g1.is_directed():
(20)                                g2._adj[0][0] = data2
(16)                            else:
(20)                                g2._succ[0][0] = data2
(20)                                g2._pred[0][0] = data2
(12)                        assert not nx.is_isomorphic(g1, g2, edge_match=em)
(0)             def test_weightkey():
(4)                 g1 = nx.DiGraph()
(4)                 g2 = nx.DiGraph()
(4)                 g1.add_edge("A", "B", weight=1)
(4)                 g2.add_edge("C", "D", weight=0)
(4)                 assert nx.is_isomorphic(g1, g2)
(4)                 em = iso.numerical_edge_match("nonexistent attribute", 1)
(4)                 assert nx.is_isomorphic(g1, g2, edge_match=em)
(4)                 em = iso.numerical_edge_match("weight", 1)
(4)                 assert not nx.is_isomorphic(g1, g2, edge_match=em)
(4)                 g2 = nx.DiGraph()
(4)                 g2.add_edge("C", "D")
(4)                 assert nx.is_isomorphic(g1, g2, edge_match=em)
(0)             class TestNodeMatch_Graph:
(4)                 def setup_method(self):
(8)                     self.g1 = nx.Graph()
(8)                     self.g2 = nx.Graph()
(8)                     self.build()
(4)                 def build(self):
(8)                     self.nm = iso.categorical_node_match("color", "")
(8)                     self.em = iso.numerical_edge_match("weight", 1)
(8)                     self.g1.add_node("A", color="red")
(8)                     self.g2.add_node("C", color="blue")
(8)                     self.g1.add_edge("A", "B", weight=1)
(8)                     self.g2.add_edge("C", "D", weight=1)
(4)                 def test_noweight_nocolor(self):
(8)                     assert nx.is_isomorphic(self.g1, self.g2)
(4)                 def test_color1(self):
(8)                     assert not nx.is_isomorphic(self.g1, self.g2, node_match=self.nm)
(4)                 def test_color2(self):
(8)                     self.g1.nodes["A"]["color"] = "blue"
(8)                     assert nx.is_isomorphic(self.g1, self.g2, node_match=self.nm)
(4)                 def test_weight1(self):
(8)                     assert nx.is_isomorphic(self.g1, self.g2, edge_match=self.em)
(4)                 def test_weight2(self):
(8)                     self.g1.add_edge("A", "B", weight=2)
(8)                     assert not nx.is_isomorphic(self.g1, self.g2, edge_match=self.em)
(4)                 def test_colorsandweights1(self):
(8)                     iso = nx.is_isomorphic(self.g1, self.g2, node_match=self.nm,
edge_match=self.em)
(8)                     assert not iso
(4)                 def test_colorsandweights2(self):
(8)                     self.g1.nodes["A"]["color"] = "blue"
(8)                     iso = nx.is_isomorphic(self.g1, self.g2, node_match=self.nm,
edge_match=self.em)
(8)                     assert iso
(4)                 def test_colorsandweights3(self):
(8)                     self.g1.add_edge("A", "B", weight=2)
(8)                     assert not nx.is_isomorphic(
(12)                        self.g1, self.g2, node_match=self.nm, edge_match=self.em
(8)                     )
(0)             class TestEdgeMatch_MultiGraph:
(4)                 def setup_method(self):
(8)                     self.g1 = nx.MultiGraph()
(8)                     self.g2 = nx.MultiGraph()
(8)                     self.GM = iso.MultiGraphMatcher
(8)                     self.build()
(4)                 def build(self):
(8)                     g1 = self.g1
(8)                     g2 = self.g2
(8)                     g1.add_edge("A", "B", color="green", weight=0, size=0.5)
(8)                     g1.add_edge("A", "B", color="red", weight=1, size=0.35)
```

```
105: (8)                    g1.add_edge("A", "B", color="red", weight=2, size=0.65)
106: (8)                    g2.add_edge("C", "D", color="green", weight=1, size=0.5)
107: (8)                    g2.add_edge("C", "D", color="red", weight=0, size=0.45)
108: (8)                    g2.add_edge("C", "D", color="red", weight=2, size=0.65)
109: (8)                    if g1.is_multigraph():
110: (12)                       self.em = iso.numerical_multiedge_match("weight", 1)
111: (12)                       self.emc = iso.categorical_multiedge_match("color", "")
112: (12)                       self.emcm = iso.categorical_multiedge_match(["color", "weight"],
["", 1])
113: (12)                       self.emg1 = iso.generic_multiedge_match("color", "red", eq)
114: (12)                       self.emg2 = iso.generic_multiedge_match(
115: (16)                           ["color", "weight", "size"],
116: (16)                           ["red", 1, 0.5],
117: (16)                           [eq, eq, math.isclose],
118: (12)                       )
119: (8)                    else:
120: (12)                       self.em = iso.numerical_edge_match("weight", 1)
121: (12)                       self.emc = iso.categorical_edge_match("color", "")
122: (12)                       self.emcm = iso.categorical_edge_match(["color", "weight"], ["",
1])
123: (12)                       self.emg1 = iso.generic_multiedge_match("color", "red", eq)
124: (12)                       self.emg2 = iso.generic_edge_match(
125: (16)                           ["color", "weight", "size"],
126: (16)                           ["red", 1, 0.5],
127: (16)                           [eq, eq, math.isclose],
128: (12)                       )
129: (4)                ef test_weights_only(self):
130: (8)                    assert nx.is_isomorphic(self.g1, self.g2, edge_match=self.em)
131: (4)                ef test_colors_only(self):
132: (8)                    gm = self.GM(self.g1, self.g2, edge_match=self.emc)
133: (8)                    assert gm.is_isomorphic()
134: (4)                ef test_colorsandweights(self):
135: (8)                    gm = self.GM(self.g1, self.g2, edge_match=self.emcm)
136: (8)                    assert not gm.is_isomorphic()
137: (4)                ef test_generic1(self):
138: (8)                    gm = self.GM(self.g1, self.g2, edge_match=self.emg1)
139: (8)                    assert gm.is_isomorphic()
140: (4)                ef test_generic2(self):
141: (8)                    gm = self.GM(self.g1, self.g2, edge_match=self.emg2)
142: (8)                    assert not gm.is_isomorphic()
143: (0)            class TestEdgeMatch_DiGraph(TestNodeMatch_Graph):
144: (4)                ef setup_method(self):
145: (8)                    TestNodeMatch_Graph.setup_method(self)
146: (8)                    self.g1 = nx.DiGraph()
147: (8)                    self.g2 = nx.DiGraph()
148: (8)                    self.build()
149: (0)            class TestEdgeMatch_MultiDiGraph(TestEdgeMatch_MultiGraph):
150: (4)                ef setup_method(self):
151: (8)                    TestEdgeMatch_MultiGraph.setup_method(self)
152: (8)                    self.g1 = nx.MultiDiGraph()
153: (8)                    self.g2 = nx.MultiDiGraph()
154: (8)                    self.GM = iso.MultiDiGraphMatcher
155: (8)                    self.build()

----------------------------------------

File 273 - __init__.py:

1: (0)              from networkx.algorithms.link_analysis.hits_alg import *
2: (0)              from networkx.algorithms.link_analysis.pagerank_alg import *

----------------------------------------

File 274 - hits_alg.py:

1: (0)              """Hubs and authorities analysis of graph structure.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              __all__ = ["hits"]
```

```
5: (0)              @nx._dispatchable(preserve_edge_attrs={"G": {"weight": 1}})
6: (0)              def hits(G, max_iter=100, tol=1.0e-8, nstart=None, normalized=True):
7: (4)                  """Returns HITS hubs and authorities values for nodes.
8: (4)                  The HITS algorithm computes two numbers for a node.
9: (4)                  Authorities estimates the node value based on the incoming links.
10: (4)                 Hubs estimates the node value based on outgoing links.
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 G : graph
14: (6)                   A NetworkX graph
15: (4)                 max_iter : integer, optional
16: (6)                   Maximum number of iterations in power method.
17: (4)                 tol : float, optional
18: (6)                   Error tolerance used to check convergence in power method iteration.
19: (4)                 nstart : dictionary, optional
20: (6)                   Starting value of each node for power method iteration.
21: (4)                 normalized : bool (default=True)
22: (7)                    Normalize results by the sum of all of the values.
23: (4)                 Returns
24: (4)                 -------
25: (4)                 (hubs,authorities) : two-tuple of dictionaries
26: (7)                    Two dictionaries keyed by node containing the hub and authority
27: (7)                    values.
28: (4)                 Raises
29: (4)                 ------
30: (4)                 PowerIterationFailedConvergence
31: (8)                     If the algorithm fails to converge to the specified tolerance
32: (8)                     within the specified number of iterations of the power iteration
33: (8)                     method.
34: (4)                 Examples
35: (4)                 --------
36: (4)                 >>> G = nx.path_graph(4)
37: (4)                 >>> h, a = nx.hits(G)
38: (4)                 Notes
39: (4)                 -----
40: (4)                 The eigenvector calculation is done by the power iteration method
41: (4)                 and has no guarantee of convergence.  The iteration will stop
42: (4)                 after max_iter iterations or an error tolerance of
43: (4)                 number_of_nodes(G)*tol has been reached.
44: (4)                 The HITS algorithm was designed for directed graphs but this
45: (4)                 algorithm does not check if the input graph is directed and will
46: (4)                 execute on undirected graphs.
47: (4)                 References
48: (4)                 ----------
49: (4)                 .. [1] A. Langville and C. Meyer,
50: (7)                    "A survey of eigenvector methods of web information retrieval."
51: (7)                    http://citeseer.ist.psu.edu/713792.html
52: (4)                 .. [2] Jon Kleinberg,
53: (7)                    Authoritative sources in a hyperlinked environment
54: (7)                    Journal of the ACM 46 (5): 604-32, 1999.
55: (7)                    doi:10.1145/324133.324140.
56: (7)                    http://www.cs.cornell.edu/home/kleinber/auth.pdf.
57: (4)                 """
58: (4)                 import numpy as np
59: (4)                 import scipy as sp
60: (4)                 if len(G) == 0:
61: (8)                     return {}, {}
62: (4)                 A = nx.adjacency_matrix(G, nodelist=list(G), dtype=float)
63: (4)                 if nstart is not None:
64: (8)                     nstart = np.array(list(nstart.values()))
65: (4)                 if max_iter <= 0:
66: (8)                     raise nx.PowerIterationFailedConvergence(max_iter)
67: (4)                 try:
68: (8)                     _, _, vt = sp.sparse.linalg.svds(A, k=1, v0=nstart, maxiter=max_iter,
tol=tol)
69: (4)                 except sp.sparse.linalg.ArpackNoConvergence as exc:
70: (8)                     raise nx.PowerIterationFailedConvergence(max_iter) from exc
71: (4)                 a = vt.flatten().real
72: (4)                 h = A @ a
```

```
 73: (4)                 if normalized:
 74: (8)                     h /= h.sum()
 75: (8)                     a /= a.sum()
 76: (4)                 hubs = dict(zip(G, map(float, h)))
 77: (4)                 authorities = dict(zip(G, map(float, a)))
 78: (4)                 return hubs, authorities
 79: (0)             def _hits_python(G, max_iter=100, tol=1.0e-8, nstart=None, normalized=True):
 80: (4)                 if isinstance(G, nx.MultiGraph | nx.MultiDiGraph):
 81: (8)                     raise Exception("hits() not defined for graphs with multiedges.")
 82: (4)                 if len(G) == 0:
 83: (8)                     return {}, {}
 84: (4)                 if nstart is None:
 85: (8)                     h = dict.fromkeys(G, 1.0 / G.number_of_nodes())
 86: (4)                 else:
 87: (8)                     h = nstart
 88: (8)                     s = 1.0 / sum(h.values())
 89: (8)                     for k in h:
 90: (12)                        h[k] *= s
 91: (4)                 for _ in range(max_iter):  # power iteration: make up to max_iter
iterations
 92: (8)                     hlast = h
 93: (8)                     h = dict.fromkeys(hlast.keys(), 0)
 94: (8)                     a = dict.fromkeys(hlast.keys(), 0)
 95: (8)                     for n in h:
 96: (12)                        for nbr in G[n]:
 97: (16)                           a[nbr] += hlast[n] * G[n][nbr].get("weight", 1)
 98: (8)                     for n in h:
 99: (12)                        for nbr in G[n]:
 100: (16)                          h[n] += a[nbr] * G[n][nbr].get("weight", 1)
 101: (8)                    s = 1.0 / max(h.values())
 102: (8)                    for n in h:
 103: (12)                       h[n] *= s
 104: (8)                    s = 1.0 / max(a.values())
 105: (8)                    for n in a:
 106: (12)                       a[n] *= s
 107: (8)                    err = sum(abs(h[n] - hlast[n]) for n in h)
 108: (8)                    if err < tol:
 109: (12)                       break
 110: (4)                else:
 111: (8)                    raise nx.PowerIterationFailedConvergence(max_iter)
 112: (4)                if normalized:
 113: (8)                    s = 1.0 / sum(a.values())
 114: (8)                    for n in a:
 115: (12)                       a[n] *= s
 116: (8)                    s = 1.0 / sum(h.values())
 117: (8)                    for n in h:
 118: (12)                       h[n] *= s
 119: (4)                return h, a
 120: (0)            def _hits_numpy(G, normalized=True):
 121: (4)                """Returns HITS hubs and authorities values for nodes.
 122: (4)                The HITS algorithm computes two numbers for a node.
 123: (4)                Authorities estimates the node value based on the incoming links.
 124: (4)                Hubs estimates the node value based on outgoing links.
 125: (4)                Parameters
 126: (4)                ----------
 127: (4)                G : graph
 128: (6)                  A NetworkX graph
 129: (4)                normalized : bool (default=True)
 130: (7)                   Normalize results by the sum of all of the values.
 131: (4)                Returns
 132: (4)                -------
 133: (4)                (hubs,authorities) : two-tuple of dictionaries
 134: (7)                   Two dictionaries keyed by node containing the hub and authority
 135: (7)                   values.
 136: (4)                Examples
 137: (4)                --------
 138: (4)                >>> G = nx.path_graph(4)
 139: (4)                The `hubs` and `authorities` are given by the eigenvectors corresponding
to the
```

```
140: (4)                maximum eigenvalues of the hubs_matrix and the authority_matrix,
respectively.
141: (4)                The ``hubs`` and ``authority`` matrices are computed from the adjacency
142: (4)                matrix:
143: (4)                >>> adj_ary = nx.to_numpy_array(G)
144: (4)                >>> hubs_matrix = adj_ary @ adj_ary.T
145: (4)                >>> authority_matrix = adj_ary.T @ adj_ary
146: (4)                `_hits_numpy` maps the eigenvector corresponding to the maximum eigenvalue
147: (4)                of the respective matrices to the nodes in `G`:
148: (4)                >>> from networkx.algorithms.link_analysis.hits_alg import _hits_numpy
149: (4)                >>> hubs, authority = _hits_numpy(G)
150: (4)                Notes
151: (4)                -----
152: (4)                The eigenvector calculation uses NumPy's interface to LAPACK.
153: (4)                The HITS algorithm was designed for directed graphs but this
154: (4)                algorithm does not check if the input graph is directed and will
155: (4)                execute on undirected graphs.
156: (4)                References
157: (4)                ----------
158: (4)                .. [1] A. Langville and C. Meyer,
159: (7)                   "A survey of eigenvector methods of web information retrieval."
160: (7)                   http://citeseer.ist.psu.edu/713792.html
161: (4)                .. [2] Jon Kleinberg,
162: (7)                   Authoritative sources in a hyperlinked environment
163: (7)                   Journal of the ACM 46 (5): 604-32, 1999.
164: (7)                   doi:10.1145/324133.324140.
165: (7)                   http://www.cs.cornell.edu/home/kleinber/auth.pdf.
166: (4)                """
167: (4)                import numpy as np
168: (4)                if len(G) == 0:
169: (8)                    return {}, {}
170: (4)                adj_ary = nx.to_numpy_array(G)
171: (4)                H = adj_ary @ adj_ary.T
172: (4)                e, ev = np.linalg.eig(H)
173: (4)                h = ev[:, np.argmax(e)]  # eigenvector corresponding to the maximum
eigenvalue
174: (4)                A = adj_ary.T @ adj_ary
175: (4)                e, ev = np.linalg.eig(A)
176: (4)                a = ev[:, np.argmax(e)]  # eigenvector corresponding to the maximum
eigenvalue
177: (4)                if normalized:
178: (8)                    h /= h.sum()
179: (8)                    a /= a.sum()
180: (4)                else:
181: (8)                    h /= h.max()
182: (8)                    a /= a.max()
183: (4)                hubs = dict(zip(G, map(float, h)))
184: (4)                authorities = dict(zip(G, map(float, a)))
185: (4)                return hubs, authorities
186: (0)            def _hits_scipy(G, max_iter=100, tol=1.0e-6, nstart=None, normalized=True):
187: (4)                """Returns HITS hubs and authorities values for nodes.
188: (4)                The HITS algorithm computes two numbers for a node.
189: (4)                Authorities estimates the node value based on the incoming links.
190: (4)                Hubs estimates the node value based on outgoing links.
191: (4)                Parameters
192: (4)                ----------
193: (4)                G : graph
194: (6)                  A NetworkX graph
195: (4)                max_iter : integer, optional
196: (6)                  Maximum number of iterations in power method.
197: (4)                tol : float, optional
198: (6)                  Error tolerance used to check convergence in power method iteration.
199: (4)                nstart : dictionary, optional
200: (6)                  Starting value of each node for power method iteration.
201: (4)                normalized : bool (default=True)
202: (7)                   Normalize results by the sum of all of the values.
203: (4)                Returns
204: (4)                -------
205: (4)                (hubs,authorities) : two-tuple of dictionaries
```

```
   Two dictionaries keyed by node containing the hub and authority
   values.
Examples
--------
>>> from networkx.algorithms.link_analysis.hits_alg import _hits_scipy
>>> G = nx.path_graph(4)
>>> h, a = _hits_scipy(G)
Notes
-----
This implementation uses SciPy sparse matrices.
The eigenvector calculation is done by the power iteration method
and has no guarantee of convergence.  The iteration will stop
after max_iter iterations or an error tolerance of
number_of_nodes(G)*tol has been reached.
The HITS algorithm was designed for directed graphs but this
algorithm does not check if the input graph is directed and will
execute on undirected graphs.
Raises
------
PowerIterationFailedConvergence
    If the algorithm fails to converge to the specified tolerance
    within the specified number of iterations of the power iteration
    method.
References
----------
.. [1] A. Langville and C. Meyer,
   "A survey of eigenvector methods of web information retrieval."
   http://citeseer.ist.psu.edu/713792.html
.. [2] Jon Kleinberg,
   Authoritative sources in a hyperlinked environment
   Journal of the ACM 46 (5): 604-632, 1999.
   doi:10.1145/324133.324140.
   http://www.cs.cornell.edu/home/kleinber/auth.pdf.
"""
import numpy as np
if len(G) == 0:
    return {}, {}
A = nx.to_scipy_sparse_array(G, nodelist=list(G))
(n, _) = A.shape  # should be square
ATA = A.T @ A  # authority matrix
if nstart is None:
    x = np.ones((n, 1)) / n
else:
    x = np.array([nstart.get(n, 0) for n in list(G)], dtype=float)
    x /= x.sum()
i = 0
while True:
    xlast = x
    x = ATA @ x
    x /= x.max()
    err = np.absolute(x - xlast).sum()
    if err < tol:
        break
    if i > max_iter:
        raise nx.PowerIterationFailedConvergence(max_iter)
    i += 1
a = x.flatten()
h = A @ a
if normalized:
    h /= h.sum()
    a /= a.sum()
hubs = dict(zip(G, map(float, h)))
authorities = dict(zip(G, map(float, a)))
return hubs, authorities
```

File 275 - pagerank_alg.py:

```
(0)              """PageRank analysis of graph structure. """
(0)              from warnings import warn
(0)              import networkx as nx
(0)              __all__ = ["pagerank", "google_matrix"]
(0)              @nx._dispatchable(edge_attrs="weight")
(0)              def pagerank(
(4)                  G,
(4)                  alpha=0.85,
(4)                  personalization=None,
(4)                  max_iter=100,
(4)                  tol=1.0e-6,
(4)                  nstart=None,
(4)                  weight="weight",
(4)                  dangling=None,
(0)              ):
(4)                  """Returns the PageRank of the nodes in the graph.
(4)                  PageRank computes a ranking of the nodes in the graph G based on
(4)                  the structure of the incoming links. It was originally designed as
(4)                  an algorithm to rank web pages.
(4)                  Parameters
(4)                  ----------
(4)                  G : graph
(6)                    A NetworkX graph.  Undirected graphs will be converted to a directed
(6)                    graph with two directed edges for each undirected edge.
(4)                  alpha : float, optional
(6)                    Damping parameter for PageRank, default=0.85.
(4)                  personalization: dict, optional
(6)                    The "personalization vector" consisting of a dictionary with a
(6)                    key some subset of graph nodes and personalization value each of those.
(6)                    At least one personalization value must be non-zero.
(6)                    If not specified, a nodes personalization value will be zero.
(6)                    By default, a uniform distribution is used.
(4)                  max_iter : integer, optional
(6)                    Maximum number of iterations in power method eigenvalue solver.
(4)                  tol : float, optional
(6)                    Error tolerance used to check convergence in power method solver.
(6)                    The iteration will stop after a tolerance of ``len(G) * tol`` is
reached.
(4)                  nstart : dictionary, optional
(6)                    Starting value of PageRank iteration for each node.
(4)                  weight : key, optional
(6)                    Edge data key to use as weight.  If None weights are set to 1.
(4)                  dangling: dict, optional
(6)                    The outedges to be assigned to any "dangling" nodes, i.e., nodes without
(6)                    any outedges. The dict key is the node the outedge points to and the
dict
(6)                    value is the weight of that outedge. By default, dangling nodes are
given
(6)                    outedges according to the personalization vector (uniform if not
(6)                    specified). This must be selected to result in an irreducible transition
(6)                    matrix (see notes under google_matrix). It may be common to have the
(6)                    dangling dict to be the same as the personalization dict.
(4)                  Returns
(4)                  -------
(4)                  pagerank : dictionary
(7)                     Dictionary of nodes with PageRank as value
(4)                  Examples
(4)                  --------
(4)                  >>> G = nx.DiGraph(nx.path_graph(4))
(4)                  >>> pr = nx.pagerank(G, alpha=0.9)
(4)                  Notes
(4)                  -----
(4)                  The eigenvector calculation is done by the power iteration method
(4)                  and has no guarantee of convergence.  The iteration will stop after
(4)                  an error tolerance of ``len(G) * tol`` has been reached. If the
(4)                  number of iterations exceed `max_iter`, a
(4)                  :exc:`networkx.exception.PowerIterationFailedConvergence` exception
(4)                  is raised.
(4)                  The PageRank algorithm was designed for directed graphs but this
```

```
    algorithm does not check if the input graph is directed and will
    execute on undirected graphs by converting each edge in the
    directed graph to two edges.
    See Also
    --------
    google_matrix
    Raises
    ------
    PowerIterationFailedConvergence
        If the algorithm fails to converge to the specified tolerance
        within the specified number of iterations of the power iteration
        method.
    References
    ----------
    .. [1] A. Langville and C. Meyer,
       "A survey of eigenvector methods of web information retrieval."
       http://citeseer.ist.psu.edu/713792.html
    .. [2] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry,
       The PageRank citation ranking: Bringing order to the Web. 1999
       http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf
    """
    return _pagerank_scipy(
        G, alpha, personalization, max_iter, tol, nstart, weight, dangling
    )
def _pagerank_python(
    G,
    alpha=0.85,
    personalization=None,
    max_iter=100,
    tol=1.0e-6,
    nstart=None,
    weight="weight",
    dangling=None,
):
    if len(G) == 0:
        return {}
    D = G.to_directed()
    W = nx.stochastic_graph(D, weight=weight)
    N = W.number_of_nodes()
    if nstart is None:
        x = dict.fromkeys(W, 1.0 / N)
    else:
        s = sum(nstart.values())
        x = {k: v / s for k, v in nstart.items()}
    if personalization is None:
        p = dict.fromkeys(W, 1.0 / N)
    else:
        s = sum(personalization.values())
        p = {k: v / s for k, v in personalization.items()}
    if dangling is None:
        dangling_weights = p
    else:
        s = sum(dangling.values())
        dangling_weights = {k: v / s for k, v in dangling.items()}
    dangling_nodes = [n for n in W if W.out_degree(n, weight=weight) == 0.0]
    for _ in range(max_iter):
        xlast = x
        x = dict.fromkeys(xlast.keys(), 0)
        danglesum = alpha * sum(xlast[n] for n in dangling_nodes)
        for n in x:
            for _, nbr, wt in W.edges(n, data=weight):
                x[nbr] += alpha * xlast[n] * wt
            x[n] += danglesum * dangling_weights.get(n, 0) + (1.0 - alpha) * p.get(n, 0)
        err = sum(abs(x[n] - xlast[n]) for n in x)
        if err < N * tol:
            return x
    raise nx.PowerIterationFailedConvergence(max_iter)
```

```
(0)            @nx._dispatchable(edge_attrs="weight")
(0)            def google_matrix(
(4)                G, alpha=0.85, personalization=None, nodelist=None, weight="weight", dangling=None
(0)            ):
(4)                """Returns the Google matrix of the graph.
(4)                Parameters
(4)                ----------
(4)                G : graph
(6)                  A NetworkX graph.  Undirected graphs will be converted to a directed
(6)                  graph with two directed edges for each undirected edge.
(4)                alpha : float
(6)                  The damping factor.
(4)                personalization: dict, optional
(6)                  The "personalization vector" consisting of a dictionary with a
(6)                  key some subset of graph nodes and personalization value each of those.
(6)                  At least one personalization value must be non-zero.
(6)                  If not specified, a nodes personalization value will be zero.
(6)                  By default, a uniform distribution is used.
(4)                nodelist : list, optional
(6)                  The rows and columns are ordered according to the nodes in nodelist.
(6)                  If nodelist is None, then the ordering is produced by G.nodes().
(4)                weight : key, optional
(6)                  Edge data key to use as weight.  If None weights are set to 1.
(4)                dangling: dict, optional
(6)                  The outedges to be assigned to any "dangling" nodes, i.e., nodes without
(6)                  any outedges. The dict key is the node the outedge points to and the dict
(6)                  value is the weight of that outedge. By default, dangling nodes are given
(6)                  outedges according to the personalization vector (uniform if not
(6)                  specified) This must be selected to result in an irreducible transition
(6)                  matrix (see notes below). It may be common to have the dangling dict to
(6)                  be the same as the personalization dict.
(4)                Returns
(4)                -------
(4)                A : 2D NumPy ndarray
(7)                   Google matrix of the graph
(4)                Notes
(4)                -----
(4)                The array returned represents the transition matrix that describes the
(4)                Markov chain used in PageRank. For PageRank to converge to a unique
(4)                solution (i.e., a unique stationary distribution in a Markov chain), the
(4)                transition matrix must be irreducible. In other words, it must be that
(4)                there exists a path between every pair of nodes in the graph, or else there
(4)                is the potential of "rank sinks."
(4)                This implementation works with Multi(Di)Graphs. For multigraphs the
(4)                weight between two nodes is set to be the sum of all edge weights
(4)                between those nodes.
(4)                See Also
(4)                --------
(4)                pagerank
(4)                """
(4)                import numpy as np
(4)                if nodelist is None:
(8)                    nodelist = list(G)
(4)                A = nx.to_numpy_array(G, nodelist=nodelist, weight=weight)
(4)                N = len(G)
(4)                if N == 0:
(8)                    return A
(4)                if personalization is None:
(8)                    p = np.repeat(1.0 / N, N)
(4)                else:
(8)                    p = np.array([personalization.get(n, 0) for n in nodelist], dtype=float)
(8)                    if p.sum() == 0:
(12)                       raise ZeroDivisionError
(8)                    p /= p.sum()
```

```
198: (4)                if dangling is None:
199: (8)                    dangling_weights = p
200: (4)                else:
201: (8)                    dangling_weights = np.array([dangling.get(n, 0) for n in nodelist],
dtype=float)
202: (8)                    dangling_weights /= dangling_weights.sum()
203: (4)                dangling_nodes = np.where(A.sum(axis=1) == 0)[0]
204: (4)                A[dangling_nodes] = dangling_weights
205: (4)                A /= A.sum(axis=1)[:, np.newaxis]  # Normalize rows to sum to 1
206: (4)                return alpha * A + (1 - alpha) * p
207: (0)            def _pagerank_numpy(
208: (4)                G, alpha=0.85, personalization=None, weight="weight", dangling=None
209: (0)            ):
210: (4)                """Returns the PageRank of the nodes in the graph.
211: (4)                PageRank computes a ranking of the nodes in the graph G based on
212: (4)                the structure of the incoming links. It was originally designed as
213: (4)                an algorithm to rank web pages.
214: (4)                Parameters
215: (4)                ----------
216: (4)                G : graph
217: (6)                  A NetworkX graph.  Undirected graphs will be converted to a directed
218: (6)                  graph with two directed edges for each undirected edge.
219: (4)                alpha : float, optional
220: (6)                  Damping parameter for PageRank, default=0.85.
221: (4)                personalization: dict, optional
222: (6)                  The "personalization vector" consisting of a dictionary with a
223: (6)                  key some subset of graph nodes and personalization value each of those.
224: (6)                  At least one personalization value must be non-zero.
225: (6)                  If not specified, a nodes personalization value will be zero.
226: (6)                  By default, a uniform distribution is used.
227: (4)                weight : key, optional
228: (6)                  Edge data key to use as weight.  If None weights are set to 1.
229: (4)                dangling: dict, optional
230: (6)                  The outedges to be assigned to any "dangling" nodes, i.e., nodes without
231: (6)                  any outedges. The dict key is the node the outedge points to and the
dict
232: (6)                  value is the weight of that outedge. By default, dangling nodes are
given
233: (6)                  outedges according to the personalization vector (uniform if not
234: (6)                  specified) This must be selected to result in an irreducible transition
235: (6)                  matrix (see notes under google_matrix). It may be common to have the
236: (6)                  dangling dict to be the same as the personalization dict.
237: (4)                Returns
238: (4)                -------
239: (4)                pagerank : dictionary
240: (7)                   Dictionary of nodes with PageRank as value.
241: (4)                Examples
242: (4)                --------
243: (4)                >>> from networkx.algorithms.link_analysis.pagerank_alg import
_pagerank_numpy
244: (4)                >>> G = nx.DiGraph(nx.path_graph(4))
245: (4)                >>> pr = _pagerank_numpy(G, alpha=0.9)
246: (4)                Notes
247: (4)                -----
248: (4)                The eigenvector calculation uses NumPy's interface to the LAPACK
249: (4)                eigenvalue solvers.  This will be the fastest and most accurate
250: (4)                for small graphs.
251: (4)                This implementation works with Multi(Di)Graphs. For multigraphs the
252: (4)                weight between two nodes is set to be the sum of all edge weights
253: (4)                between those nodes.
254: (4)                See Also
255: (4)                --------
256: (4)                pagerank, google_matrix
257: (4)                References
258: (4)                ----------
259: (4)                .. [1] A. Langville and C. Meyer,
260: (7)                   "A survey of eigenvector methods of web information retrieval."
261: (7)                   http://citeseer.ist.psu.edu/713792.html
262: (4)                .. [2] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry,
```

```
263: (7)                   The PageRank citation ranking: Bringing order to the Web. 1999
264: (7)                   http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-
66&format=pdf
265: (4)                """
266: (4)                import numpy as np
267: (4)                if len(G) == 0:
268: (8)                    return {}
269: (4)                M = google_matrix(
270: (8)                    G, alpha, personalization=personalization, weight=weight,
dangling=dangling
271: (4)                )
272: (4)                eigenvalues, eigenvectors = np.linalg.eig(M.T)
273: (4)                ind = np.argmax(eigenvalues)
274: (4)                largest = np.array(eigenvectors[:, ind]).flatten().real
275: (4)                norm = largest.sum()
276: (4)                return dict(zip(G, map(float, largest / norm)))
277: (0)            def _pagerank_scipy(
278: (4)                G,
279: (4)                alpha=0.85,
280: (4)                personalization=None,
281: (4)                max_iter=100,
282: (4)                tol=1.0e-6,
283: (4)                nstart=None,
284: (4)                weight="weight",
285: (4)                dangling=None,
286: (0)            ):
287: (4)                """Returns the PageRank of the nodes in the graph.
288: (4)                PageRank computes a ranking of the nodes in the graph G based on
289: (4)                the structure of the incoming links. It was originally designed as
290: (4)                an algorithm to rank web pages.
291: (4)                Parameters
292: (4)                ----------
293: (4)                G : graph
294: (6)                  A NetworkX graph.  Undirected graphs will be converted to a directed
295: (6)                  graph with two directed edges for each undirected edge.
296: (4)                alpha : float, optional
297: (6)                  Damping parameter for PageRank, default=0.85.
298: (4)                personalization: dict, optional
299: (6)                  The "personalization vector" consisting of a dictionary with a
300: (6)                  key some subset of graph nodes and personalization value each of those.
301: (6)                  At least one personalization value must be non-zero.
302: (6)                  If not specified, a nodes personalization value will be zero.
303: (6)                  By default, a uniform distribution is used.
304: (4)                max_iter : integer, optional
305: (6)                  Maximum number of iterations in power method eigenvalue solver.
306: (4)                tol : float, optional
307: (6)                  Error tolerance used to check convergence in power method solver.
308: (6)                  The iteration will stop after a tolerance of ``len(G) * tol`` is
reached.
309: (4)                nstart : dictionary, optional
310: (6)                  Starting value of PageRank iteration for each node.
311: (4)                weight : key, optional
312: (6)                  Edge data key to use as weight.  If None weights are set to 1.
313: (4)                dangling: dict, optional
314: (6)                  The outedges to be assigned to any "dangling" nodes, i.e., nodes without
315: (6)                  any outedges. The dict key is the node the outedge points to and the
dict
316: (6)                  value is the weight of that outedge. By default, dangling nodes are
given
317: (6)                  outedges according to the personalization vector (uniform if not
318: (6)                  specified) This must be selected to result in an irreducible transition
319: (6)                  matrix (see notes under google_matrix). It may be common to have the
320: (6)                  dangling dict to be the same as the personalization dict.
321: (4)                Returns
322: (4)                -------
323: (4)                pagerank : dictionary
324: (7)                   Dictionary of nodes with PageRank as value
325: (4)                Examples
326: (4)                --------
```

```
    >>> from networkx.algorithms.link_analysis.pagerank_alg import _pagerank_scipy
    >>> G = nx.DiGraph(nx.path_graph(4))
    >>> pr = _pagerank_scipy(G, alpha=0.9)
    Notes
    -----
    The eigenvector calculation uses power iteration with a SciPy
    sparse matrix representation.
    This implementation works with Multi(Di)Graphs. For multigraphs the
    weight between two nodes is set to be the sum of all edge weights
    between those nodes.
    See Also
    --------
    pagerank
    Raises
    ------
    PowerIterationFailedConvergence
        If the algorithm fails to converge to the specified tolerance
        within the specified number of iterations of the power iteration
        method.
    References
    ----------
    .. [1] A. Langville and C. Meyer,
       "A survey of eigenvector methods of web information retrieval."
       http://citeseer.ist.psu.edu/713792.html
    .. [2] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry,
       The PageRank citation ranking: Bringing order to the Web. 1999
       http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf
    """
    import numpy as np
    import scipy as sp
    N = len(G)
    if N == 0:
        return {}
    nodelist = list(G)
    A = nx.to_scipy_sparse_array(G, nodelist=nodelist, weight=weight, dtype=float)
    S = A.sum(axis=1)
    S[S != 0] = 1.0 / S[S != 0]
    Q = sp.sparse.csr_array(sp.sparse.spdiags(S.T, 0, *A.shape))
    A = Q @ A
    if nstart is None:
        x = np.repeat(1.0 / N, N)
    else:
        x = np.array([nstart.get(n, 0) for n in nodelist], dtype=float)
        x /= x.sum()
    if personalization is None:
        p = np.repeat(1.0 / N, N)
    else:
        p = np.array([personalization.get(n, 0) for n in nodelist], dtype=float)
        if p.sum() == 0:
            raise ZeroDivisionError
        p /= p.sum()
    if dangling is None:
        dangling_weights = p
    else:
        dangling_weights = np.array([dangling.get(n, 0) for n in nodelist], dtype=float)
        dangling_weights /= dangling_weights.sum()
    is_dangling = np.where(S == 0)[0]
    for _ in range(max_iter):
        xlast = x
        x = alpha * (x @ A + sum(x[is_dangling]) * dangling_weights) + (1 - alpha) * p
        err = np.absolute(x - xlast).sum()
        if err < N * tol:
            return dict(zip(nodelist, map(float, x)))
```

```
390: (4)                raise nx.PowerIterationFailedConvergence(max_iter)

----------------------------------------

File 276 - __init__.py:

1: (0)

----------------------------------------

File 277 - test_hits.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              np = pytest.importorskip("numpy")
4: (0)              sp = pytest.importorskip("scipy")
5: (0)              from networkx.algorithms.link_analysis.hits_alg import (
6: (4)                  _hits_numpy,
7: (4)                  _hits_python,
8: (4)                  _hits_scipy,
9: (0)              )
10: (0)             class TestHITS:
11: (4)                 @classmethod
12: (4)                 def setup_class(cls):
13: (8)                     G = nx.DiGraph()
14: (8)                     edges = [(1, 3), (1, 5), (2, 1), (3, 5), (5, 4), (5, 3), (6, 5)]
15: (8)                     G.add_edges_from(edges, weight=1)
16: (8)                     cls.G = G
17: (8)                     cls.G.a = dict(
18: (12)                        zip(sorted(G), [0.000000, 0.000000, 0.366025, 0.133975, 0.500000,
0.000000])
19: (8)                     )
20: (8)                     cls.G.h = dict(
21: (12)                        zip(sorted(G), [0.366025, 0.000000, 0.211325, 0.000000, 0.211325,
0.211325])
22: (8)                     )
23: (4)                 def test_hits_numpy(self):
24: (8)                     G = self.G
25: (8)                     h, a = _hits_numpy(G)
26: (8)                     for n in G:
27: (12)                        assert h[n] == pytest.approx(G.h[n], abs=1e-4)
28: (8)                     for n in G:
29: (12)                        assert a[n] == pytest.approx(G.a[n], abs=1e-4)
30: (4)                 @pytest.mark.parametrize("hits_alg", (nx.hits, _hits_python, _hits_scipy))
31: (4)                 def test_hits(self, hits_alg):
32: (8)                     G = self.G
33: (8)                     h, a = hits_alg(G, tol=1.0e-08)
34: (8)                     for n in G:
35: (12)                        assert h[n] == pytest.approx(G.h[n], abs=1e-4)
36: (8)                     for n in G:
37: (12)                        assert a[n] == pytest.approx(G.a[n], abs=1e-4)
38: (8)                     nstart = {i: 1.0 / 2 for i in G}
39: (8)                     h, a = hits_alg(G, nstart=nstart)
40: (8)                     for n in G:
41: (12)                        assert h[n] == pytest.approx(G.h[n], abs=1e-4)
42: (8)                     for n in G:
43: (12)                        assert a[n] == pytest.approx(G.a[n], abs=1e-4)
44: (4)                 def test_empty(self):
45: (8)                     G = nx.Graph()
46: (8)                     assert nx.hits(G) == ({}, {})
47: (8)                     assert _hits_numpy(G) == ({}, {})
48: (8)                     assert _hits_python(G) == ({}, {})
49: (8)                     assert _hits_scipy(G) == ({}, {})
50: (4)                 def test_hits_not_convergent(self):
51: (8)                     G = nx.path_graph(50)
52: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
53: (12)                        _hits_scipy(G, max_iter=1)
54: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
55: (12)                        _hits_python(G, max_iter=1)
```

```
56: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
57: (12)                        _hits_scipy(G, max_iter=0)
58: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
59: (12)                        _hits_python(G, max_iter=0)
60: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
61: (12)                        nx.hits(G, max_iter=0)
62: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
63: (12)                        nx.hits(G, max_iter=1)

----------------------------------------

File 278 - test_pagerank.py:

1: (0)              import random
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.classes.tests import dispatch_interface
5: (0)              np = pytest.importorskip("numpy")
6: (0)              pytest.importorskip("scipy")
7: (0)              from networkx.algorithms.link_analysis.pagerank_alg import (
8: (4)                  _pagerank_numpy,
9: (4)                  _pagerank_python,
10: (4)                 _pagerank_scipy,
11: (0)             )
12: (0)             class TestPageRank:
13: (4)                 @classmethod
14: (4)                 def setup_class(cls):
15: (8)                     G = nx.DiGraph()
16: (8)                     edges = [
17: (12)                        (1, 2),
18: (12)                        (1, 3),
19: (12)                        (3, 1),
20: (12)                        (3, 2),
21: (12)                        (3, 5),
22: (12)                        (4, 5),
23: (12)                        (4, 6),
24: (12)                        (5, 4),
25: (12)                        (5, 6),
26: (12)                        (6, 4),
27: (8)                     ]
28: (8)                     G.add_edges_from(edges)
29: (8)                     cls.G = G
30: (8)                     cls.G.pagerank = dict(
31: (12)                        zip(
32: (16)                            sorted(G),
33: (16)                            [
34: (20)                                0.03721197,
35: (20)                                0.05395735,
36: (20)                                0.04150565,
37: (20)                                0.37508082,
38: (20)                                0.20599833,
39: (20)                                0.28624589,
40: (16)                            ],
41: (12)                        )
42: (8)                     )
43: (8)                     cls.dangling_node_index = 1
44: (8)                     cls.dangling_edges = {1: 2, 2: 3, 3: 0, 4: 0, 5: 0, 6: 0}
45: (8)                     cls.G.dangling_pagerank = dict(
46: (12)                        zip(
47: (16)                            sorted(G),
48: (16)                            [0.10844518, 0.18618601, 0.0710892, 0.2683668, 0.15919783,
0.20671497],
49: (12)                        )
50: (8)                     )
51: (4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python))
52: (4)                 def test_pagerank(self, alg):
53: (8)                     G = self.G
54: (8)                     p = alg(G, alpha=0.9, tol=1.0e-08)
55: (8)                     for n in G:
```

```
56: (12)                        assert p[n] == pytest.approx(G.pagerank[n], abs=1e-4)
57: (8)                     nstart = {n: random.random() for n in G}
58: (8)                     p = alg(G, alpha=0.9, tol=1.0e-08, nstart=nstart)
59: (8)                     for n in G:
60: (12)                        assert p[n] == pytest.approx(G.pagerank[n], abs=1e-4)
61: (4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python))
62: (4)                 def test_pagerank_max_iter(self, alg):
63: (8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
64: (12)                        alg(self.G, max_iter=0)
65: (4)                 def test_numpy_pagerank(self):
66: (8)                     G = self.G
67: (8)                     p = _pagerank_numpy(G, alpha=0.9)
68: (8)                     for n in G:
69: (12)                        assert p[n] == pytest.approx(G.pagerank[n], abs=1e-4)
70: (4)                 @pytest.mark.parametrize("wrapper", [lambda x: x,
dispatch_interface.convert])
71: (4)                 def test_google_matrix(self, wrapper):
72: (8)                     G = wrapper(self.G)
73: (8)                     M = nx.google_matrix(G, alpha=0.9, nodelist=sorted(G))
74: (8)                     _, ev = np.linalg.eig(M.T)
75: (8)                     p = ev[:, 0] / ev[:, 0].sum()
76: (8)                     for a, b in zip(p, self.G.pagerank.values()):
77: (12)                        assert a == pytest.approx(b, abs=1e-7)
78: (4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python,
_pagerank_numpy))
79: (4)                 def test_personalization(self, alg):
80: (8)                     G = nx.complete_graph(4)
81: (8)                     personalize = {0: 1, 1: 1, 2: 4, 3: 4}
82: (8)                     answer = {
83: (12)                        0: 0.23246732615667579,
84: (12)                        1: 0.23246732615667579,
85: (12)                        2: 0.267532673843324,
86: (12)                        3: 0.2675326738433241,
87: (8)                     }
88: (8)                     p = alg(G, alpha=0.85, personalization=personalize)
89: (8)                     for n in G:
90: (12)                        assert p[n] == pytest.approx(answer[n], abs=1e-4)
91: (4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python,
nx.google_matrix))
92: (4)                 def test_zero_personalization_vector(self, alg):
93: (8)                     G = nx.complete_graph(4)
94: (8)                     personalize = {0: 0, 1: 0, 2: 0, 3: 0}
95: (8)                     pytest.raises(ZeroDivisionError, alg, G, personalization=personalize)
96: (4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python))
97: (4)                 def test_one_nonzero_personalization_value(self, alg):
98: (8)                     G = nx.complete_graph(4)
99: (8)                     personalize = {0: 0, 1: 0, 2: 0, 3: 1}
100: (8)                    answer = {
101: (12)                       0: 0.22077931820379187,
102: (12)                       1: 0.22077931820379187,
103: (12)                       2: 0.22077931820379187,
104: (12)                       3: 0.3376620453886241,
105: (8)                    }
106: (8)                    p = alg(G, alpha=0.85, personalization=personalize)
107: (8)                    for n in G:
108: (12)                       assert p[n] == pytest.approx(answer[n], abs=1e-4)
109: (4)                @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python))
110: (4)                def test_incomplete_personalization(self, alg):
111: (8)                    G = nx.complete_graph(4)
112: (8)                    personalize = {3: 1}
113: (8)                    answer = {
114: (12)                       0: 0.22077931820379187,
115: (12)                       1: 0.22077931820379187,
116: (12)                       2: 0.22077931820379187,
117: (12)                       3: 0.3376620453886241,
118: (8)                    }
119: (8)                    p = alg(G, alpha=0.85, personalization=personalize)
120: (8)                    for n in G:
121: (12)                       assert p[n] == pytest.approx(answer[n], abs=1e-4)
```

```
(4)                 def test_dangling_matrix(self):
(8)                     """
(8)                     Tests that the google_matrix doesn't change except for the dangling
(8)                     nodes.
(8)                     """
(8)                     G = self.G
(8)                     dangling = self.dangling_edges
(8)                     dangling_sum = sum(dangling.values())
(8)                     M1 = nx.google_matrix(G, personalization=dangling)
(8)                     M2 = nx.google_matrix(G, personalization=dangling, dangling=dangling)
(8)                     for i in range(len(G)):
(12)                        for j in range(len(G)):
(16)                            if i == self.dangling_node_index and (j + 1) in dangling:
(20)                                assert M2[i, j] == pytest.approx(
(24)                                    dangling[j + 1] / dangling_sum, abs=1e-4
(20)                                )
(16)                            else:
(20)                                assert M2[i, j] == pytest.approx(M1[i, j], abs=1e-4)
(4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python,
_pagerank_numpy))
(4)                 def test_dangling_pagerank(self, alg):
(8)                     pr = alg(self.G, dangling=self.dangling_edges)
(8)                     for n in self.G:
(12)                        assert pr[n] == pytest.approx(self.G.dangling_pagerank[n], abs=1e-
4)
(4)                 def test_empty(self):
(8)                     G = nx.Graph()
(8)                     assert nx.pagerank(G) == {}
(8)                     assert _pagerank_python(G) == {}
(8)                     assert _pagerank_numpy(G) == {}
(8)                     assert nx.google_matrix(G).shape == (0, 0)
(4)                 @pytest.mark.parametrize("alg", (nx.pagerank, _pagerank_python))
(4)                 def test_multigraph(self, alg):
(8)                     G = nx.MultiGraph()
(8)                     G.add_edges_from([(1, 2), (1, 2), (1, 2), (2, 3), (2, 3), ("3", 3),
("3", 3)])
(8)                     answer = {
(12)                        1: 0.21066048614468322,
(12)                        2: 0.3395308825985378,
(12)                        3: 0.28933951385531687,
(12)                        "3": 0.16046911740146227,
(8)                     }
(8)                     p = alg(G)
(8)                     for n in G:
(12)                        assert p[n] == pytest.approx(answer[n], abs=1e-4)
(0)             class TestPageRankScipy(TestPageRank):
(4)                 def test_scipy_pagerank(self):
(8)                     G = self.G
(8)                     p = _pagerank_scipy(G, alpha=0.9, tol=1.0e-08)
(8)                     for n in G:
(12)                        assert p[n] == pytest.approx(G.pagerank[n], abs=1e-4)
(8)                     personalize = {n: random.random() for n in G}
(8)                     p = _pagerank_scipy(G, alpha=0.9, tol=1.0e-08,
personalization=personalize)
(8)                     nstart = {n: random.random() for n in G}
(8)                     p = _pagerank_scipy(G, alpha=0.9, tol=1.0e-08, nstart=nstart)
(8)                     for n in G:
(12)                        assert p[n] == pytest.approx(G.pagerank[n], abs=1e-4)
(4)                 def test_scipy_pagerank_max_iter(self):
(8)                     with pytest.raises(nx.PowerIterationFailedConvergence):
(12)                        _pagerank_scipy(self.G, max_iter=0)
(4)                 def test_dangling_scipy_pagerank(self):
(8)                     pr = _pagerank_scipy(self.G, dangling=self.dangling_edges)
(8)                     for n in self.G:
(12)                        assert pr[n] == pytest.approx(self.G.dangling_pagerank[n], abs=1e-
4)
(4)                 def test_empty_scipy(self):
(8)                     G = nx.Graph()
(8)                     assert _pagerank_scipy(G) == {}
```

```
----------------------------------------

File 279 - __init__.py:

1: (0)              """
2: (0)              Subpackages related to graph-minor problems.
3: (0)              In graph theory, an undirected graph H is called a minor of the graph G if H
4: (0)              can be formed from G by deleting edges and vertices and by contracting edges
5: (0)              [1]_.
6: (0)              References
7: (0)              ----------
8: (0)              .. [1] https://en.wikipedia.org/wiki/Graph_minor
9: (0)              """
10: (0)             from networkx.algorithms.minors.contraction import (
11: (4)                 contracted_edge,
12: (4)                 contracted_nodes,
13: (4)                 equivalence_classes,
14: (4)                 identified_nodes,
15: (4)                 quotient_graph,
16: (0)             )
17: (0)             __all__ = [
18: (4)                 "contracted_edge",
19: (4)                 "contracted_nodes",
20: (4)                 "equivalence_classes",
21: (4)                 "identified_nodes",
22: (4)                 "quotient_graph",
23: (0)             ]

----------------------------------------

File 280 - contraction.py:

1: (0)              """Provides functions for computing minors of a graph."""
2: (0)              from itertools import chain, combinations, permutations, product
3: (0)              import networkx as nx
4: (0)              from networkx import density
5: (0)              from networkx.exception import NetworkXException
6: (0)              from networkx.utils import arbitrary_element
7: (0)              __all__ = [
8: (4)                  "contracted_edge",
9: (4)                  "contracted_nodes",
10: (4)                 "equivalence_classes",
11: (4)                 "identified_nodes",
12: (4)                 "quotient_graph",
13: (0)             ]
14: (0)             chaini = chain.from_iterable
15: (0)             def equivalence_classes(iterable, relation):
16: (4)                 """Returns equivalence classes of `relation` when applied to `iterable`.
17: (4)                 The equivalence classes, or blocks, consist of objects from `iterable`
18: (4)                 which are all equivalent. They are defined to be equivalent if the
19: (4)                 `relation` function returns `True` when passed any two objects from that
20: (4)                 class, and `False` otherwise. To define an equivalence relation the
21: (4)                 function must be reflexive, symmetric and transitive.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 iterable : list, tuple, or set
25: (8)                     An iterable of elements/nodes.
26: (4)                 relation : function
27: (8)                     A Boolean-valued function that implements an equivalence relation
28: (8)                     (reflexive, symmetric, transitive binary relation) on the elements
29: (8)                     of `iterable` - it must take two elements and return `True` if
30: (8)                     they are related, or `False` if not.
31: (4)                 Returns
32: (4)                 -------
33: (4)                 set of frozensets
34: (8)                     A set of frozensets representing the partition induced by the
equivalence
35: (8)                     relation function `relation` on the elements of `iterable`. Each
```

```
        member set in the return set represents an equivalence class, or
        block, of the partition.
        Duplicate elements will be ignored so it makes the most sense for
        `iterable` to be a :class:`set`.
    Notes
    -----
    This function does not check that `relation` represents an equivalence
    relation. You can check that your equivalence classes provide a partition
    using `is_partition`.
    Examples
    --------
    Let `X` be the set of integers from `0` to `9`, and consider an equivalence
    relation `R` on `X` of congruence modulo `3`: this means that two integers
    `x` and `y` in `X` are equivalent under `R` if they leave the same
    remainder when divided by `3`, i.e. `(x - y) mod 3 = 0`.
    The equivalence classes of this relation are `{0, 3, 6, 9}`, `{1, 4, 7}`,
    `{2, 5, 8}`: `0`, `3`, `6`, `9` are all divisible by `3` and leave zero
    remainder; `1`, `4`, `7` leave remainder `1`; while `2`, `5` and `8` leave
    remainder `2`. We can see this by calling `equivalence_classes` with
    `X` and a function implementation of `R`.
    >>> X = set(range(10))
    >>> def mod3(x, y):
    ...     return (x - y) % 3 == 0
    >>> equivalence_classes(X, mod3)  # doctest: +SKIP
    {frozenset({1, 4, 7}), frozenset({8, 2, 5}), frozenset({0, 9, 3, 6})}
    """
    blocks = []
    for y in iterable:
        for block in blocks:
            x = arbitrary_element(block)
            if relation(x, y):
                block.append(y)
                break
        else:
            blocks.append([y])
    return {frozenset(block) for block in blocks}
@nx._dispatchable(edge_attrs="weight", returns_graph=True)
def quotient_graph(
    G,
    partition,
    edge_relation=None,
    node_data=None,
    edge_data=None,
    weight="weight",
    relabel=False,
    create_using=None,
):
    """Returns the quotient graph of `G` under the specified equivalence
    relation on nodes.
    Parameters
    ----------
    G : NetworkX graph
        The graph for which to return the quotient graph with the
        specified node relation.
    partition : function, or dict or list of lists, tuples or sets
        If a function, this function must represent an equivalence
        relation on the nodes of `G`. It must take two arguments *u*
        and *v* and return True exactly when *u* and *v* are in the
        same equivalence class. The equivalence classes form the nodes
        in the returned graph.
        If a dict of lists/tuples/sets, the keys can be any meaningful
        block labels, but the values must be the block lists/tuples/sets
        (one list/tuple/set per block), and the blocks must form a valid
        partition of the nodes of the graph. That is, each node must be
        in exactly one block of the partition.
        If a list of sets, the list must form a valid partition of
        the nodes of the graph. That is, each node must be in exactly
        one block of the partition.
```

```
    edge_relation : Boolean function with two arguments
        This function must represent an edge relation on the *blocks* of
        the `partition` of `G`. It must take two arguments, *B* and *C*,
        each one a set of nodes, and return True exactly when there should be
        an edge joining block *B* to block *C* in the returned graph.
        If `edge_relation` is not specified, it is assumed to be the
        following relation. Block *B* is related to block *C* if and
        only if some node in *B* is adjacent to some node in *C*,
        according to the edge set of `G`.
    node_data : function
        This function takes one argument, *B*, a set of nodes in `G`,
        and must return a dictionary representing the node data
        attributes to set on the node representing *B* in the quotient graph.
        If None, the following node attributes will be set:
        * 'graph', the subgraph of the graph `G` that this block
          represents,
        * 'nnodes', the number of nodes in this block,
        * 'nedges', the number of edges within this block,
        * 'density', the density of the subgraph of `G` that this
          block represents.
    edge_data : function
        This function takes two arguments, *B* and *C*, each one a set
        of nodes, and must return a dictionary representing the edge
        data attributes to set on the edge joining *B* and *C*, should
        there be an edge joining *B* and *C* in the quotient graph (if
        no such edge occurs in the quotient graph as determined by
        `edge_relation`, then the output of this function is ignored).
        If the quotient graph would be a multigraph, this function is
        not applied, since the edge data from each edge in the graph
        `G` appears in the edges of the quotient graph.
    weight : string or None, optional (default="weight")
        The name of an edge attribute that holds the numerical value
        used as a weight. If None then each edge has weight 1.
    relabel : bool
        If True, relabel the nodes of the quotient graph to be
        nonnegative integers. Otherwise, the nodes are identified with
        :class:`frozenset` instances representing the blocks given in
        `partition`.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    NetworkX graph
        The quotient graph of `G` under the equivalence relation
        specified by `partition`. If the partition were given as a
        list of :class:`set` instances and `relabel` is False,
        each node will be a :class:`frozenset` corresponding to the same
        :class:`set`.
    Raises
    ------
    NetworkXException
        If the given partition is not a valid partition of the nodes of
        `G`.
    Examples
    --------
    The quotient graph of the complete bipartite graph under the "same
    neighbors" equivalence relation is `K_2`. Under this relation, two nodes
    are equivalent if they are not adjacent but have the same neighbor set.
    >>> G = nx.complete_bipartite_graph(2, 3)
    >>> same_neighbors = lambda u, v: (u not in G[v] and v not in G[u] and
G[u] == G[v])
    >>> Q = nx.quotient_graph(G, same_neighbors)
    >>> K2 = nx.complete_graph(2)
    >>> nx.is_isomorphic(Q, K2)
    True
    The quotient graph of a directed graph under the "same strongly connected
    component" equivalence relation is the condensation of the graph (see
    :func:`condensation`). This example comes from the Wikipedia article
    *`Strongly connected component`_*.
```

```
(4)                 >>> G = nx.DiGraph()
(4)                 >>> edges = [
(4)                 ...     "ab",
(4)                 ...     "be",
(4)                 ...     "bf",
(4)                 ...     "bc",
(4)                 ...     "cg",
(4)                 ...     "cd",
(4)                 ...     "dc",
(4)                 ...     "dh",
(4)                 ...     "ea",
(4)                 ...     "ef",
(4)                 ...     "fg",
(4)                 ...     "gf",
(4)                 ...     "hd",
(4)                 ...     "hf",
(4)                 ... ]
(4)                 >>> G.add_edges_from(tuple(x) for x in edges)
(4)                 >>> components = list(nx.strongly_connected_components(G))
(4)                 >>> sorted(sorted(component) for component in components)
(4)                 [['a', 'b', 'e'], ['c', 'd', 'h'], ['f', 'g']]
(4)                 >>>
(4)                 >>> C = nx.condensation(G, components)
(4)                 >>> component_of = C.graph["mapping"]
(4)                 >>> same_component = lambda u, v: component_of[u] == component_of[v]
(4)                 >>> Q = nx.quotient_graph(G, same_component)
(4)                 >>> nx.is_isomorphic(C, Q)
(4)                 True
(4)                 Node identification can be represented as the quotient of a graph under
the
(4)                 equivalence relation that places the two nodes in one block and each other
(4)                 node in its own singleton block.
(4)                 >>> K24 = nx.complete_bipartite_graph(2, 4)
(4)                 >>> K34 = nx.complete_bipartite_graph(3, 4)
(4)                 >>> C = nx.contracted_nodes(K34, 1, 2)
(4)                 >>> nodes = {1, 2}
(4)                 >>> is_contracted = lambda u, v: u in nodes and v in nodes
(4)                 >>> Q = nx.quotient_graph(K34, is_contracted)
(4)                 >>> nx.is_isomorphic(Q, C)
(4)                 True
(4)                 >>> nx.is_isomorphic(Q, K24)
(4)                 True
(4)                 The blockmodeling technique described in [1]_ can be implemented as a
(4)                 quotient graph.
(4)                 >>> G = nx.path_graph(6)
(4)                 >>> partition = [{0, 1}, {2, 3}, {4, 5}]
(4)                 >>> M = nx.quotient_graph(G, partition, relabel=True)
(4)                 >>> list(M.edges())
(4)                 [(0, 1), (1, 2)]
(4)                 Here is the sample example but using partition as a dict of block sets.
(4)                 >>> G = nx.path_graph(6)
(4)                 >>> partition = {0: {0, 1}, 2: {2, 3}, 4: {4, 5}}
(4)                 >>> M = nx.quotient_graph(G, partition, relabel=True)
(4)                 >>> list(M.edges())
(4)                 [(0, 1), (1, 2)]
(4)                 Partitions can be represented in various ways:
(4)                 0. a list/tuple/set of block lists/tuples/sets
(4)                 1. a dict with block labels as keys and blocks lists/tuples/sets as values
(4)                 2. a dict with block lists/tuples/sets as keys and block labels as values
(4)                 3. a function from nodes in the original iterable to block labels
(4)                 4. an equivalence relation function on the target iterable
(4)                 As `quotient_graph` is designed to accept partitions represented as (0),
(1) or
(4)                 (4) only, the `equivalence_classes` function can be used to get the
partitions
(4)                 in the right form, in order to call `quotient_graph`.
(4)                 .. _Strongly connected component:
https://en.wikipedia.org/wiki/Strongly_connected_component
(4)                 References
```

```
(4)                 ----------
(4)                 .. [1] Patrick Doreian, Vladimir Batagelj, and Anuska Ferligoj.
(11)                       *Generalized Blockmodeling*.
(11)                       Cambridge University Press, 2004.
(4)                 """
(4)                 if callable(partition):
(8)                     partition = equivalence_classes(G, partition)
(8)                     if not nx.community.is_partition(G, partition):
(12)                        raise nx.NetworkXException(
(16)                            "Input `partition` is not an equivalence relation for nodes of
G"
(12)                        )
(8)                     return _quotient_graph(
(12)                        G,
(12)                        partition,
(12)                        edge_relation,
(12)                        node_data,
(12)                        edge_data,
(12)                        weight,
(12)                        relabel,
(12)                        create_using,
(8)                     )
(4)                 if isinstance(partition, dict):
(8)                     partition = list(partition.values())
(4)                 partition_nodes = set().union(*partition)
(4)                 if len(partition_nodes) != len(G):
(8)                     G = G.subgraph(partition_nodes)
(4)                 if not nx.community.is_partition(G, partition):
(8)                     raise NetworkXException("each node must be in exactly one part of
`partition`")
(4)                 return _quotient_graph(
(8)                     G,
(8)                     partition,
(8)                     edge_relation,
(8)                     node_data,
(8)                     edge_data,
(8)                     weight,
(8)                     relabel,
(8)                     create_using,
(4)                 )
(0)             def _quotient_graph(
(4)                 G, partition, edge_relation, node_data, edge_data, weight, relabel,
create_using
(0)             ):
(4)                 """Construct the quotient graph assuming input has been checked"""
(4)                 if create_using is None:
(8)                     H = G.__class__()
(4)                 else:
(8)                     H = nx.empty_graph(0, create_using)
(4)                 if node_data is None:
(8)                     def node_data(b):
(12)                        S = G.subgraph(b)
(12)                        return {
(16)                            "graph": S,
(16)                            "nnodes": len(S),
(16)                            "nedges": S.number_of_edges(),
(16)                            "density": density(S),
(12)                        }
(4)                 partition = [frozenset(b) for b in partition]
(4)                 H.add_nodes_from((b, node_data(b)) for b in partition)
(4)                 if edge_relation is None:
(8)                     def edge_relation(b, c):
(12)                        return any(v in G[u] for u, v in product(b, c))
(4)                 if edge_data is None:
(8)                     def edge_data(b, c):
(12)                        edgedata = (
(16)                            d
(16)                            for u, v, d in G.edges(b | c, data=True)
(16)                            if (u in b and v in c) or (u in c and v in b)
```

```
(12)                        )
(12)                        return {"weight": sum(d.get(weight, 1) for d in edgedata)}
(4)                 block_pairs = permutations(H, 2) if H.is_directed() else combinations(H,
2)
(4)                 if H.is_multigraph():
(8)                     edges = chaini(
(12)                        (
(16)                            (b, c, G.get_edge_data(u, v, default={}))
(16)                            for u, v in product(b, c)
(16)                            if v in G[u]
(12)                        )
(12)                        for b, c in block_pairs
(12)                        if edge_relation(b, c)
(8)                     )
(4)                 else:
(8)                     edges = (
(12)                        (b, c, edge_data(b, c)) for (b, c) in block_pairs if
edge_relation(b, c)
(8)                     )
(4)                 H.add_edges_from(edges)
(4)                 if relabel:
(8)                     labels = {b: i for i, b in enumerate(partition)}
(8)                     H = nx.relabel_nodes(H, labels)
(4)                 return H
(0)             @nx._dispatchable(
(4)                 preserve_all_attrs=True, mutates_input={"not copy": 4}, returns_graph=True
(0)             )
(0)             def contracted_nodes(G, u, v, self_loops=True, copy=True):
(4)                 """Returns the graph that results from contracting `u` and `v`.
(4)                 Node contraction identifies the two nodes as a single node incident to any
(4)                 edge that was incident to the original two nodes.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     The graph whose nodes will be contracted.
(4)                 u, v : nodes
(8)                     Must be nodes in `G`.
(4)                 self_loops : Boolean
(8)                     If this is True, any edges joining `u` and `v` in `G` become
(8)                     self-loops on the new node in the returned graph.
(4)                 copy : Boolean
(8)                     If this is True (default True), make a copy of
(8)                     `G` and return that instead of directly changing `G`.
(4)                 Returns
(4)                 -------
(4)                 Networkx graph
(8)                     If Copy is True,
(8)                     A new graph object of the same type as `G` (leaving `G` unmodified)
(8)                     with `u` and `v` identified in a single node. The right node `v`
(8)                     will be merged into the node `u`, so only `u` will appear in the
(8)                     returned graph.
(8)                     If copy is False,
(8)                     Modifies `G` with `u` and `v` identified in a single node.
(8)                     The right node `v` will be merged into the node `u`, so
(8)                     only `u` will appear in the returned graph.
(4)                 Notes
(4)                 -----
(4)                 For multigraphs, the edge keys for the realigned edges may
(4)                 not be the same as the edge keys for the old edges. This is
(4)                 natural because edge keys are unique only within each pair of nodes.
(4)                 For non-multigraphs where `u` and `v` are adjacent to a third node
(4)                 `w`, the edge (`v`, `w`) will be contracted into the edge (`u`,
(4)                 `w`) with its attributes stored into a "contraction" attribute.
(4)                 This function is also available as `identified_nodes`.
(4)                 Examples
(4)                 --------
(4)                 Contracting two nonadjacent nodes of the cycle graph on four nodes `C_4`
(4)                 yields the path graph (ignoring parallel edges):
(4)                 >>> G = nx.cycle_graph(4)
```

```
    >>> M = nx.contracted_nodes(G, 1, 3)
    >>> P3 = nx.path_graph(3)
    >>> nx.is_isomorphic(M, P3)
    True
    >>> G = nx.MultiGraph(P3)
    >>> M = nx.contracted_nodes(G, 0, 2)
    >>> M.edges
    MultiEdgeView([(0, 1, 0), (0, 1, 1)])
    >>> G = nx.Graph([(1, 2), (2, 2)])
    >>> H = nx.contracted_nodes(G, 1, 2, self_loops=False)
    >>> list(H.nodes())
    [1]
    >>> list(H.edges())
    [(1, 1)]
    In a ``MultiDiGraph`` with a self loop, the in and out edges will
    be treated separately as edges, so while contracting a node which
    has a self loop the contraction will add multiple edges:
    >>> G = nx.MultiDiGraph([(1, 2), (2, 2)])
    >>> H = nx.contracted_nodes(G, 1, 2)
    >>> list(H.edges())  # edge 1->2, 2->2, 2<-2 from the original Graph G
    [(1, 1), (1, 1), (1, 1)]
    >>> H = nx.contracted_nodes(G, 1, 2, self_loops=False)
    >>> list(H.edges())  # edge 2->2, 2<-2 from the original Graph G
    [(1, 1), (1, 1)]
    See Also
    --------
    contracted_edge
    quotient_graph
    """
    if copy:
        H = G.copy()
    else:
        H = G
    if H.is_directed():
        edges_to_remap = chain(G.in_edges(v, data=True), G.out_edges(v, data=True))
    else:
        edges_to_remap = G.edges(v, data=True)
    if not copy:
        edges_to_remap = list(edges_to_remap)
    v_data = H.nodes[v]
    H.remove_node(v)
    for prev_w, prev_x, d in edges_to_remap:
        w = prev_w if prev_w != v else u
        x = prev_x if prev_x != v else u
        if ({prev_w, prev_x} == {u, v}) and not self_loops:
            continue
        if not H.has_edge(w, x) or G.is_multigraph():
            H.add_edge(w, x, **d)
        else:
            if "contraction" in H.edges[(w, x)]:
                H.edges[(w, x)]["contraction"][(prev_w, prev_x)] = d
            else:
                H.edges[(w, x)]["contraction"] = {(prev_w, prev_x): d}
    if "contraction" in H.nodes[u]:
        H.nodes[u]["contraction"][v] = v_data
    else:
        H.nodes[u]["contraction"] = {v: v_data}
    return H
identified_nodes = contracted_nodes
@nx._dispatchable(
    preserve_edge_attrs=True, mutates_input={"not copy": 3}, returns_graph=True
)
def contracted_edge(G, edge, self_loops=True, copy=True):
    """Returns the graph that results from contracting the specified edge.
    Edge contraction identifies the two endpoints of the edge as a single node
    incident to any edge that was incident to the original two nodes. A graph
    that results from edge contraction is called a *minor* of the original
```

```
437: (4)                graph.
438: (4)                Parameters
439: (4)                ----------
440: (4)                G : NetworkX graph
441: (7)                   The graph whose edge will be contracted.
442: (4)                edge : tuple
443: (7)                   Must be a pair of nodes in `G`.
444: (4)                self_loops : Boolean
445: (7)                   If this is True, any edges (including `edge`) joining the
446: (7)                   endpoints of `edge` in `G` become self-loops on the new node in the
447: (7)                   returned graph.
448: (4)                copy : Boolean (default True)
449: (8)                    If this is True, a the contraction will be performed on a copy of `G`,
450: (8)                    otherwise the contraction will happen in place.
451: (4)                Returns
452: (4)                -------
453: (4)                Networkx graph
454: (7)                   A new graph object of the same type as `G` (leaving `G` unmodified)
455: (7)                   with endpoints of `edge` identified in a single node. The right node
456: (7)                   of `edge` will be merged into the left one, so only the left one will
457: (7)                   appear in the returned graph.
458: (4)                Raises
459: (4)                ------
460: (4)                ValueError
461: (7)                   If `edge` is not an edge in `G`.
462: (4)                Examples
463: (4)                --------
464: (4)                Attempting to contract two nonadjacent nodes yields an error:
465: (4)                >>> G = nx.cycle_graph(4)
466: (4)                >>> nx.contracted_edge(G, (1, 3))
467: (4)                Traceback (most recent call last):
468: (6)                  ...
469: (4)                ValueError: Edge (1, 3) does not exist in graph G; cannot contract it
470: (4)                Contracting two adjacent nodes in the cycle graph on *n* nodes yields the
471: (4)                cycle graph on *n - 1* nodes:
472: (4)                >>> C5 = nx.cycle_graph(5)
473: (4)                >>> C4 = nx.cycle_graph(4)
474: (4)                >>> M = nx.contracted_edge(C5, (0, 1), self_loops=False)
475: (4)                >>> nx.is_isomorphic(M, C4)
476: (4)                True
477: (4)                See also
478: (4)                --------
479: (4)                contracted_nodes
480: (4)                quotient_graph
481: (4)                """
482: (4)                u, v = edge[:2]
483: (4)                if not G.has_edge(u, v):
484: (8)                    raise ValueError(f"Edge {edge} does not exist in graph G; cannot
contract it")
485: (4)                return contracted_nodes(G, u, v, self_loops=self_loops, copy=copy)

----------------------------------------

File 281 - test_contraction.py:

1: (0)              """Unit tests for the :mod:`networkx.algorithms.minors.contraction` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import arbitrary_element, edges_equal, nodes_equal
5: (0)              def test_quotient_graph_complete_multipartite():
6: (4)                  """Tests that the quotient graph of the complete *n*-partite graph
7: (4)                  under the "same neighbors" node relation is the complete graph on *n*
8: (4)                  nodes.
9: (4)                  """
10: (4)                 G = nx.complete_multipartite_graph(2, 3, 4)
11: (4)                 def same_neighbors(u, v):
12: (8)                     return u not in G[v] and v not in G[u] and G[u] == G[v]
13: (4)                 expected = nx.complete_graph(3)
14: (4)                 actual = nx.quotient_graph(G, same_neighbors)
```

```
(4)                 assert nx.is_isomorphic(expected, actual)
(0)             def test_quotient_graph_complete_bipartite():
(4)                 """Tests that the quotient graph of the complete bipartite graph under
(4)                 the "same neighbors" node relation is `K_2`.
(4)                 """
(4)                 G = nx.complete_bipartite_graph(2, 3)
(4)                 def same_neighbors(u, v):
(8)                     return u not in G[v] and v not in G[u] and G[u] == G[v]
(4)                 expected = nx.complete_graph(2)
(4)                 actual = nx.quotient_graph(G, same_neighbors)
(4)                 assert nx.is_isomorphic(expected, actual)
(0)             def test_quotient_graph_edge_relation():
(4)                 """Tests for specifying an alternate edge relation for the quotient
(4)                 graph.
(4)                 """
(4)                 G = nx.path_graph(5)
(4)                 def identity(u, v):
(8)                     return u == v
(4)                 def same_parity(b, c):
(8)                     return arbitrary_element(b) % 2 == arbitrary_element(c) % 2
(4)                 actual = nx.quotient_graph(G, identity, same_parity)
(4)                 expected = nx.Graph()
(4)                 expected.add_edges_from([(0, 2), (0, 4), (2, 4)])
(4)                 expected.add_edge(1, 3)
(4)                 assert nx.is_isomorphic(actual, expected)
(0)             def test_condensation_as_quotient():
(4)                 """This tests that the condensation of a graph can be viewed as the
(4)                 quotient graph under the "in the same connected component" equivalence
(4)                 relation.
(4)                 """
(4)                 G = nx.DiGraph()
(4)                 G.add_edges_from(
(8)                     [
(12)                        (1, 2),
(12)                        (2, 3),
(12)                        (2, 11),
(12)                        (2, 12),
(12)                        (3, 4),
(12)                        (4, 3),
(12)                        (4, 5),
(12)                        (5, 6),
(12)                        (6, 5),
(12)                        (6, 7),
(12)                        (7, 8),
(12)                        (7, 9),
(12)                        (7, 10),
(12)                        (8, 9),
(12)                        (9, 7),
(12)                        (10, 6),
(12)                        (11, 2),
(12)                        (11, 4),
(12)                        (11, 6),
(12)                        (12, 6),
(12)                        (12, 11),
(8)                     ]
(4)                 )
(4)                 scc = list(nx.strongly_connected_components(G))
(4)                 C = nx.condensation(G, scc)
(4)                 component_of = C.graph["mapping"]
(4)                 def same_component(u, v):
(8)                     return component_of[u] == component_of[v]
(4)                 Q = nx.quotient_graph(G, same_component)
(4)                 assert nx.is_isomorphic(C, Q)
(0)             def test_path():
(4)                 G = nx.path_graph(6)
(4)                 partition = [{0, 1}, {2, 3}, {4, 5}]
(4)                 M = nx.quotient_graph(G, partition, relabel=True)
(4)                 assert nodes_equal(M, [0, 1, 2])
(4)                 assert edges_equal(M.edges(), [(0, 1), (1, 2)])
```

```
84: (4)                 for n in M:
85: (8)                     assert M.nodes[n]["nedges"] == 1
86: (8)                     assert M.nodes[n]["nnodes"] == 2
87: (8)                     assert M.nodes[n]["density"] == 1
88: (0)             def test_path__partition_provided_as_dict_of_lists():
89: (4)                 G = nx.path_graph(6)
90: (4)                 partition = {0: [0, 1], 2: [2, 3], 4: [4, 5]}
91: (4)                 M = nx.quotient_graph(G, partition, relabel=True)
92: (4)                 assert nodes_equal(M, [0, 1, 2])
93: (4)                 assert edges_equal(M.edges(), [(0, 1), (1, 2)])
94: (4)                 for n in M:
95: (8)                     assert M.nodes[n]["nedges"] == 1
96: (8)                     assert M.nodes[n]["nnodes"] == 2
97: (8)                     assert M.nodes[n]["density"] == 1
98: (0)             def test_path__partition_provided_as_dict_of_tuples():
99: (4)                 G = nx.path_graph(6)
100: (4)                partition = {0: (0, 1), 2: (2, 3), 4: (4, 5)}
101: (4)                M = nx.quotient_graph(G, partition, relabel=True)
102: (4)                assert nodes_equal(M, [0, 1, 2])
103: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
104: (4)                for n in M:
105: (8)                    assert M.nodes[n]["nedges"] == 1
106: (8)                    assert M.nodes[n]["nnodes"] == 2
107: (8)                    assert M.nodes[n]["density"] == 1
108: (0)            def test_path__partition_provided_as_dict_of_sets():
109: (4)                G = nx.path_graph(6)
110: (4)                partition = {0: {0, 1}, 2: {2, 3}, 4: {4, 5}}
111: (4)                M = nx.quotient_graph(G, partition, relabel=True)
112: (4)                assert nodes_equal(M, [0, 1, 2])
113: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
114: (4)                for n in M:
115: (8)                    assert M.nodes[n]["nedges"] == 1
116: (8)                    assert M.nodes[n]["nnodes"] == 2
117: (8)                    assert M.nodes[n]["density"] == 1
118: (0)            def test_multigraph_path():
119: (4)                G = nx.MultiGraph(nx.path_graph(6))
120: (4)                partition = [{0, 1}, {2, 3}, {4, 5}]
121: (4)                M = nx.quotient_graph(G, partition, relabel=True)
122: (4)                assert nodes_equal(M, [0, 1, 2])
123: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
124: (4)                for n in M:
125: (8)                    assert M.nodes[n]["nedges"] == 1
126: (8)                    assert M.nodes[n]["nnodes"] == 2
127: (8)                    assert M.nodes[n]["density"] == 1
128: (0)            def test_directed_path():
129: (4)                G = nx.DiGraph()
130: (4)                nx.add_path(G, range(6))
131: (4)                partition = [{0, 1}, {2, 3}, {4, 5}]
132: (4)                M = nx.quotient_graph(G, partition, relabel=True)
133: (4)                assert nodes_equal(M, [0, 1, 2])
134: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
135: (4)                for n in M:
136: (8)                    assert M.nodes[n]["nedges"] == 1
137: (8)                    assert M.nodes[n]["nnodes"] == 2
138: (8)                    assert M.nodes[n]["density"] == 0.5
139: (0)            def test_directed_multigraph_path():
140: (4)                G = nx.MultiDiGraph()
141: (4)                nx.add_path(G, range(6))
142: (4)                partition = [{0, 1}, {2, 3}, {4, 5}]
143: (4)                M = nx.quotient_graph(G, partition, relabel=True)
144: (4)                assert nodes_equal(M, [0, 1, 2])
145: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
146: (4)                for n in M:
147: (8)                    assert M.nodes[n]["nedges"] == 1
148: (8)                    assert M.nodes[n]["nnodes"] == 2
149: (8)                    assert M.nodes[n]["density"] == 0.5
150: (0)            def test_overlapping_blocks():
151: (4)                with pytest.raises(nx.NetworkXException):
152: (8)                    G = nx.path_graph(6)
```

```
153: (8)                    partition = [{0, 1, 2}, {2, 3}, {4, 5}]
154: (8)                    nx.quotient_graph(G, partition)
155: (0)            def test_weighted_path():
156: (4)                G = nx.path_graph(6)
157: (4)                for i in range(5):
158: (8)                    G[i][i + 1]["w"] = i + 1
159: (4)                partition = [{0, 1}, {2, 3}, {4, 5}]
160: (4)                M = nx.quotient_graph(G, partition, weight="w", relabel=True)
161: (4)                assert nodes_equal(M, [0, 1, 2])
162: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
163: (4)                assert M[0][1]["weight"] == 2
164: (4)                assert M[1][2]["weight"] == 4
165: (4)                for n in M:
166: (8)                    assert M.nodes[n]["nedges"] == 1
167: (8)                    assert M.nodes[n]["nnodes"] == 2
168: (8)                    assert M.nodes[n]["density"] == 1
169: (0)            def test_barbell():
170: (4)                G = nx.barbell_graph(3, 0)
171: (4)                partition = [{0, 1, 2}, {3, 4, 5}]
172: (4)                M = nx.quotient_graph(G, partition, relabel=True)
173: (4)                assert nodes_equal(M, [0, 1])
174: (4)                assert edges_equal(M.edges(), [(0, 1)])
175: (4)                for n in M:
176: (8)                    assert M.nodes[n]["nedges"] == 3
177: (8)                    assert M.nodes[n]["nnodes"] == 3
178: (8)                    assert M.nodes[n]["density"] == 1
179: (0)            def test_barbell_plus():
180: (4)                G = nx.barbell_graph(3, 0)
181: (4)                G.add_edge(0, 5)
182: (4)                partition = [{0, 1, 2}, {3, 4, 5}]
183: (4)                M = nx.quotient_graph(G, partition, relabel=True)
184: (4)                assert nodes_equal(M, [0, 1])
185: (4)                assert edges_equal(M.edges(), [(0, 1)])
186: (4)                assert M[0][1]["weight"] == 2
187: (4)                for n in M:
188: (8)                    assert M.nodes[n]["nedges"] == 3
189: (8)                    assert M.nodes[n]["nnodes"] == 3
190: (8)                    assert M.nodes[n]["density"] == 1
191: (0)            def test_blockmodel():
192: (4)                G = nx.path_graph(6)
193: (4)                partition = [[0, 1], [2, 3], [4, 5]]
194: (4)                M = nx.quotient_graph(G, partition, relabel=True)
195: (4)                assert nodes_equal(M.nodes(), [0, 1, 2])
196: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
197: (4)                for n in M.nodes():
198: (8)                    assert M.nodes[n]["nedges"] == 1
199: (8)                    assert M.nodes[n]["nnodes"] == 2
200: (8)                    assert M.nodes[n]["density"] == 1.0
201: (0)            def test_multigraph_blockmodel():
202: (4)                G = nx.MultiGraph(nx.path_graph(6))
203: (4)                partition = [[0, 1], [2, 3], [4, 5]]
204: (4)                M = nx.quotient_graph(G, partition, create_using=nx.MultiGraph(),
relabel=True)
205: (4)                assert nodes_equal(M.nodes(), [0, 1, 2])
206: (4)                assert edges_equal(M.edges(), [(0, 1), (1, 2)])
207: (4)                for n in M.nodes():
208: (8)                    assert M.nodes[n]["nedges"] == 1
209: (8)                    assert M.nodes[n]["nnodes"] == 2
210: (8)                    assert M.nodes[n]["density"] == 1.0
211: (0)            def test_quotient_graph_incomplete_partition():
212: (4)                G = nx.path_graph(6)
213: (4)                partition = []
214: (4)                H = nx.quotient_graph(G, partition, relabel=True)
215: (4)                assert nodes_equal(H.nodes(), [])
216: (4)                assert edges_equal(H.edges(), [])
217: (4)                partition = [[0, 1], [2, 3], [5]]
218: (4)                H = nx.quotient_graph(G, partition, relabel=True)
219: (4)                assert nodes_equal(H.nodes(), [0, 1, 2])
220: (4)                assert edges_equal(H.edges(), [(0, 1)])
```

```
(0)         def test_undirected_node_contraction():
(4)             """Tests for node contraction in an undirected graph."""
(4)             G = nx.cycle_graph(4)
(4)             actual = nx.contracted_nodes(G, 0, 1)
(4)             expected = nx.cycle_graph(3)
(4)             expected.add_edge(0, 0)
(4)             assert nx.is_isomorphic(actual, expected)
(0)         def test_directed_node_contraction():
(4)             """Tests for node contraction in a directed graph."""
(4)             G = nx.DiGraph(nx.cycle_graph(4))
(4)             actual = nx.contracted_nodes(G, 0, 1)
(4)             expected = nx.DiGraph(nx.cycle_graph(3))
(4)             expected.add_edge(0, 0)
(4)             expected.add_edge(0, 0)
(4)             assert nx.is_isomorphic(actual, expected)
(0)         def test_undirected_node_contraction_no_copy():
(4)             """Tests for node contraction in an undirected graph
(4)             by making changes in place."""
(4)             G = nx.cycle_graph(4)
(4)             actual = nx.contracted_nodes(G, 0, 1, copy=False)
(4)             expected = nx.cycle_graph(3)
(4)             expected.add_edge(0, 0)
(4)             assert nx.is_isomorphic(actual, G)
(4)             assert nx.is_isomorphic(actual, expected)
(0)         def test_directed_node_contraction_no_copy():
(4)             """Tests for node contraction in a directed graph
(4)             by making changes in place."""
(4)             G = nx.DiGraph(nx.cycle_graph(4))
(4)             actual = nx.contracted_nodes(G, 0, 1, copy=False)
(4)             expected = nx.DiGraph(nx.cycle_graph(3))
(4)             expected.add_edge(0, 0)
(4)             expected.add_edge(0, 0)
(4)             assert nx.is_isomorphic(actual, G)
(4)             assert nx.is_isomorphic(actual, expected)
(0)         def test_create_multigraph():
(4)             """Tests that using a MultiGraph creates multiple edges."""
(4)             G = nx.path_graph(3, create_using=nx.MultiGraph())
(4)             G.add_edge(0, 1)
(4)             G.add_edge(0, 0)
(4)             G.add_edge(0, 2)
(4)             actual = nx.contracted_nodes(G, 0, 2)
(4)             expected = nx.MultiGraph()
(4)             expected.add_edge(0, 1)
(4)             expected.add_edge(0, 1)
(4)             expected.add_edge(0, 1)
(4)             expected.add_edge(0, 0)
(4)             expected.add_edge(0, 0)
(4)             assert edges_equal(actual.edges, expected.edges)
(0)         def test_multigraph_keys():
(4)             """Tests that multiedge keys are reset in new graph."""
(4)             G = nx.path_graph(3, create_using=nx.MultiGraph())
(4)             G.add_edge(0, 1, 5)
(4)             G.add_edge(0, 0, 0)
(4)             G.add_edge(0, 2, 5)
(4)             actual = nx.contracted_nodes(G, 0, 2)
(4)             expected = nx.MultiGraph()
(4)             expected.add_edge(0, 1, 0)
(4)             expected.add_edge(0, 1, 5)
(4)             expected.add_edge(0, 1, 2)  # keyed as 2 b/c 2 edges already in G
(4)             expected.add_edge(0, 0, 0)
(4)             expected.add_edge(0, 0, 1)  # this comes from (0, 2, 5)
(4)             assert edges_equal(actual.edges, expected.edges)
(0)         def test_node_attributes():
(4)             """Tests that node contraction preserves node attributes."""
(4)             G = nx.cycle_graph(4)
(4)             G.nodes[0]["foo"] = "bar"
(4)             G.nodes[1]["baz"] = "xyzzy"
(4)             actual = nx.contracted_nodes(G, 0, 1)
(4)             expected = nx.complete_graph(3)
```

```
290: (4)                expected = nx.relabel_nodes(expected, {1: 2, 2: 3})
291: (4)                expected.add_edge(0, 0)
292: (4)                cdict = {1: {"baz": "xyzzy"}}
293: (4)                expected.nodes[0].update({"foo": "bar", "contraction": cdict})
294: (4)                assert nx.is_isomorphic(actual, expected)
295: (4)                assert actual.nodes == expected.nodes
296: (0)            def test_edge_attributes():
297: (4)                """Tests that node contraction preserves edge attributes."""
298: (4)                G = nx.DiGraph([("src1", "dest"), ("src2", "dest")])
299: (4)                G["src1"]["dest"]["value"] = "src1-->dest"
300: (4)                G["src2"]["dest"]["value"] = "src2-->dest"
301: (4)                H = nx.MultiDiGraph(G)
302: (4)                G = nx.contracted_nodes(G, "src1", "src2")  # New Shape: src1 --> dest
303: (4)                assert G.edges[("src1", "dest")]["value"] == "src1-->dest"
304: (4)                assert (
305: (8)                    G.edges[("src1", "dest")]["contraction"][("src2", "dest")]["value"]
306: (8)                    == "src2-->dest"
307: (4)                )
308: (4)                H = nx.contracted_nodes(H, "src1", "src2")  # New Shape: src1 -(x2)-> dest
309: (4)                assert len(H.edges(("src1", "dest"))) == 2
310: (0)            def test_without_self_loops():
311: (4)                """Tests for node contraction without preserving -loops."""
312: (4)                G = nx.cycle_graph(4)
313: (4)                actual = nx.contracted_nodes(G, 0, 1, self_loops=False)
314: (4)                expected = nx.complete_graph(3)
315: (4)                assert nx.is_isomorphic(actual, expected)
316: (0)            def test_contract_loop_graph():
317: (4)                """Tests for node contraction when nodes have loops."""
318: (4)                G = nx.cycle_graph(4)
319: (4)                G.add_edge(0, 0)
320: (4)                actual = nx.contracted_nodes(G, 0, 1)
321: (4)                expected = nx.complete_graph([0, 2, 3])
322: (4)                expected.add_edge(0, 0)
323: (4)                expected.add_edge(0, 0)
324: (4)                assert edges_equal(actual.edges, expected.edges)
325: (4)                actual = nx.contracted_nodes(G, 1, 0)
326: (4)                expected = nx.complete_graph([1, 2, 3])
327: (4)                expected.add_edge(1, 1)
328: (4)                expected.add_edge(1, 1)
329: (4)                assert edges_equal(actual.edges, expected.edges)
330: (0)            def test_undirected_edge_contraction():
331: (4)                """Tests for edge contraction in an undirected graph."""
332: (4)                G = nx.cycle_graph(4)
333: (4)                actual = nx.contracted_edge(G, (0, 1))
334: (4)                expected = nx.complete_graph(3)
335: (4)                expected.add_edge(0, 0)
336: (4)                assert nx.is_isomorphic(actual, expected)
337: (0)            def test_multigraph_edge_contraction():
338: (4)                """Tests for edge contraction in a multigraph"""
339: (4)                G = nx.cycle_graph(4)
340: (4)                actual = nx.contracted_edge(G, (0, 1, 0))
341: (4)                expected = nx.complete_graph(3)
342: (4)                expected.add_edge(0, 0)
343: (4)                assert nx.is_isomorphic(actual, expected)
344: (0)            def test_nonexistent_edge():
345: (4)                """Tests that attempting to contract a nonexistent edge raises an
346: (4)                exception.
347: (4)                """
348: (4)                with pytest.raises(ValueError):
349: (8)                    G = nx.cycle_graph(4)
350: (8)                    nx.contracted_edge(G, (0, 2))
```

File 282 - __init__.py:

```
1: (0)              from networkx.algorithms.operators.all import *
2: (0)              from networkx.algorithms.operators.binary import *
3: (0)              from networkx.algorithms.operators.product import *
```

```
4: (0)              from networkx.algorithms.operators.unary import *

----------------------------------------

File 283 - all.py:

1: (0)              """Operations on many graphs.
2: (0)              """
3: (0)              from itertools import chain, repeat
4: (0)              import networkx as nx
5: (0)              __all__ = ["union_all", "compose_all", "disjoint_union_all",
"intersection_all"]
6: (0)              @nx._dispatchable(graphs="[graphs]", preserve_all_attrs=True,
returns_graph=True)
7: (0)              def union_all(graphs, rename=()):
8: (4)                  """Returns the union of all graphs.
9: (4)                  The graphs must be disjoint, otherwise an exception is raised.
10: (4)                 Parameters
11: (4)                 ----------
12: (4)                 graphs : iterable
13: (7)                    Iterable of NetworkX graphs
14: (4)                 rename : iterable , optional
15: (7)                    Node names of graphs can be changed by specifying the tuple
16: (7)                    rename=('G-','H-') (for example).  Node "u" in G is then renamed
17: (7)                    "G-u" and "v" in H is renamed "H-v". Infinite generators (like
itertools.count)
18: (7)                    are also supported.
19: (4)                 Returns
20: (4)                 -------
21: (4)                 U : a graph with the same type as the first graph in list
22: (4)                 Raises
23: (4)                 ------
24: (4)                 ValueError
25: (7)                    If `graphs` is an empty list.
26: (4)                 NetworkXError
27: (8)                     In case of mixed type graphs, like MultiGraph and Graph, or directed
and undirected graphs.
28: (4)                 Notes
29: (4)                 -----
30: (4)                 For operating on mixed type graphs, they should be converted to the same
type.
31: (4)                 >>> G = nx.Graph()
32: (4)                 >>> H = nx.DiGraph()
33: (4)                 >>> GH = union_all([nx.DiGraph(G), H])
34: (4)                 To force a disjoint union with node relabeling, use
35: (4)                 disjoint_union_all(G,H) or convert_node_labels_to integers().
36: (4)                 Graph, edge, and node attributes are propagated to the union graph.
37: (4)                 If a graph attribute is present in multiple graphs, then the value
38: (4)                 from the last graph in the list with that attribute is used.
39: (4)                 Examples
40: (4)                 --------
41: (4)                 >>> G1 = nx.Graph([(1, 2), (2, 3)])
42: (4)                 >>> G2 = nx.Graph([(4, 5), (5, 6)])
43: (4)                 >>> result_graph = nx.union_all([G1, G2])
44: (4)                 >>> result_graph.nodes()
45: (4)                 NodeView((1, 2, 3, 4, 5, 6))
46: (4)                 >>> result_graph.edges()
47: (4)                 EdgeView([(1, 2), (2, 3), (4, 5), (5, 6)])
48: (4)                 See Also
49: (4)                 --------
50: (4)                 union
51: (4)                 disjoint_union_all
52: (4)                 """
53: (4)                 R = None
54: (4)                 seen_nodes = set()
55: (4)                 def add_prefix(graph, prefix):
56: (8)                     if prefix is None:
57: (12)                        return graph
58: (8)                     def label(x):
```

```
59: (12)                        return f"{prefix}{x}"
60: (8)                     return nx.relabel_nodes(graph, label)
61: (4)                 rename = chain(rename, repeat(None))
62: (4)                 graphs = (add_prefix(G, name) for G, name in zip(graphs, rename))
63: (4)                 for i, G in enumerate(graphs):
64: (8)                     G_nodes_set = set(G.nodes)
65: (8)                     if i == 0:
66: (12)                        R = G.__class__()
67: (8)                     elif G.is_directed() != R.is_directed():
68: (12)                        raise nx.NetworkXError("All graphs must be directed or
undirected.")
69: (8)                     elif G.is_multigraph() != R.is_multigraph():
70: (12)                        raise nx.NetworkXError("All graphs must be graphs or
multigraphs.")
71: (8)                     elif not seen_nodes.isdisjoint(G_nodes_set):
72: (12)                        raise nx.NetworkXError(
73: (16)                            "The node sets of the graphs are not disjoint.\n"
74: (16)                            "Use `rename` to specify prefixes for the graphs or use\n"
75: (16)                            "disjoint_union(G1, G2, ..., GN)."
76: (12)                        )
77: (8)                     seen_nodes |= G_nodes_set
78: (8)                     R.graph.update(G.graph)
79: (8)                     R.add_nodes_from(G.nodes(data=True))
80: (8)                     R.add_edges_from(
81: (12)                        G.edges(keys=True, data=True) if G.is_multigraph() else
G.edges(data=True)
82: (8)                     )
83: (4)                 if R is None:
84: (8)                     raise ValueError("cannot apply union_all to an empty list")
85: (4)                 return R
86: (0)             @nx._dispatchable(graphs="[graphs]", preserve_all_attrs=True,
returns_graph=True)
87: (0)             def disjoint_union_all(graphs):
88: (4)                 """Returns the disjoint union of all graphs.
89: (4)                 This operation forces distinct integer node labels starting with 0
90: (4)                 for the first graph in the list and numbering consecutively.
91: (4)                 Parameters
92: (4)                 ----------
93: (4)                 graphs : iterable
94: (7)                    Iterable of NetworkX graphs
95: (4)                 Returns
96: (4)                 -------
97: (4)                 U : A graph with the same type as the first graph in list
98: (4)                 Raises
99: (4)                 ------
100: (4)                ValueError
101: (7)                   If `graphs` is an empty list.
102: (4)                NetworkXError
103: (8)                    In case of mixed type graphs, like MultiGraph and Graph, or directed
and undirected graphs.
104: (4)                Examples
105: (4)                --------
106: (4)                >>> G1 = nx.Graph([(1, 2), (2, 3)])
107: (4)                >>> G2 = nx.Graph([(4, 5), (5, 6)])
108: (4)                >>> U = nx.disjoint_union_all([G1, G2])
109: (4)                >>> list(U.nodes())
110: (4)                [0, 1, 2, 3, 4, 5]
111: (4)                >>> list(U.edges())
112: (4)                [(0, 1), (1, 2), (3, 4), (4, 5)]
113: (4)                Notes
114: (4)                -----
115: (4)                For operating on mixed type graphs, they should be converted to the same
type.
116: (4)                Graph, edge, and node attributes are propagated to the union graph.
117: (4)                If a graph attribute is present in multiple graphs, then the value
118: (4)                from the last graph in the list with that attribute is used.
119: (4)                """
120: (4)                def yield_relabeled(graphs):
121: (8)                    first_label = 0
```

```
122: (8)                    for G in graphs:
123: (12)                       yield nx.convert_node_labels_to_integers(G,
first_label=first_label)
124: (12)                       first_label += len(G)
125: (4)                R = union_all(yield_relabeled(graphs))
126: (4)                return R
127: (0)            @nx._dispatchable(graphs="[graphs]", preserve_all_attrs=True,
returns_graph=True)
128: (0)            def compose_all(graphs):
129: (4)                """Returns the composition of all graphs.
130: (4)                Composition is the simple union of the node sets and edge sets.
131: (4)                The node sets of the supplied graphs need not be disjoint.
132: (4)                Parameters
133: (4)                ----------
134: (4)                graphs : iterable
135: (7)                   Iterable of NetworkX graphs
136: (4)                Returns
137: (4)                -------
138: (4)                C : A graph with the same type as the first graph in list
139: (4)                Raises
140: (4)                ------
141: (4)                ValueError
142: (7)                   If `graphs` is an empty list.
143: (4)                NetworkXError
144: (8)                    In case of mixed type graphs, like MultiGraph and Graph, or directed
and undirected graphs.
145: (4)                Examples
146: (4)                --------
147: (4)                >>> G1 = nx.Graph([(1, 2), (2, 3)])
148: (4)                >>> G2 = nx.Graph([(3, 4), (5, 6)])
149: (4)                >>> C = nx.compose_all([G1, G2])
150: (4)                >>> list(C.nodes())
151: (4)                [1, 2, 3, 4, 5, 6]
152: (4)                >>> list(C.edges())
153: (4)                [(1, 2), (2, 3), (3, 4), (5, 6)]
154: (4)                Notes
155: (4)                -----
156: (4)                For operating on mixed type graphs, they should be converted to the same
type.
157: (4)                Graph, edge, and node attributes are propagated to the union graph.
158: (4)                If a graph attribute is present in multiple graphs, then the value
159: (4)                from the last graph in the list with that attribute is used.
160: (4)                """
161: (4)                R = None
162: (4)                for i, G in enumerate(graphs):
163: (8)                    if i == 0:
164: (12)                       R = G.__class__()
165: (8)                    elif G.is_directed() != R.is_directed():
166: (12)                       raise nx.NetworkXError("All graphs must be directed or
undirected.")
167: (8)                    elif G.is_multigraph() != R.is_multigraph():
168: (12)                       raise nx.NetworkXError("All graphs must be graphs or
multigraphs.")
169: (8)                    R.graph.update(G.graph)
170: (8)                    R.add_nodes_from(G.nodes(data=True))
171: (8)                    R.add_edges_from(
172: (12)                       G.edges(keys=True, data=True) if G.is_multigraph() else
G.edges(data=True)
173: (8)                    )
174: (4)                if R is None:
175: (8)                    raise ValueError("cannot apply compose_all to an empty list")
176: (4)                return R
177: (0)            @nx._dispatchable(graphs="[graphs]", returns_graph=True)
178: (0)            def intersection_all(graphs):
179: (4)                """Returns a new graph that contains only the nodes and the edges that
exist in
180: (4)                all graphs.
181: (4)                Parameters
182: (4)                ----------
```

```
(4)                 graphs : iterable
(7)                    Iterable of NetworkX graphs
(4)                 Returns
(4)                 -------
(4)                 R : A new graph with the same type as the first graph in list
(4)                 Raises
(4)                 ------
(4)                 ValueError
(7)                    If `graphs` is an empty list.
(4)                 NetworkXError
(8)                     In case of mixed type graphs, like MultiGraph and Graph, or directed and undirected graphs.
(4)                 Notes
(4)                 -----
(4)                 For operating on mixed type graphs, they should be converted to the same type.
(4)                 Attributes from the graph, nodes, and edges are not copied to the new
(4)                 graph.
(4)                 The resulting graph can be updated with attributes if desired. For example, code which adds the minimum attribute for each node across all graphs could work.
(4)                 >>> g = nx.Graph()
(4)                 >>> g.add_node(0, capacity=4)
(4)                 >>> g.add_node(1, capacity=3)
(4)                 >>> g.add_edge(0, 1)
(4)                 >>> h = g.copy()
(4)                 >>> h.nodes[0]["capacity"] = 2
(4)                 >>> gh = nx.intersection_all([g, h])
(4)                 >>> new_node_attr = {
(4)                 ...     n: min(*(anyG.nodes[n].get("capacity", float("inf")) for anyG in [g, h]))
(4)                 ...     for n in gh
(4)                 ... }
(4)                 >>> nx.set_node_attributes(gh, new_node_attr, "new_capacity")
(4)                 >>> gh.nodes(data=True)
(4)                 NodeDataView({0: {'new_capacity': 2}, 1: {'new_capacity': 3}})
(4)                 Examples
(4)                 --------
(4)                 >>> G1 = nx.Graph([(1, 2), (2, 3)])
(4)                 >>> G2 = nx.Graph([(2, 3), (3, 4)])
(4)                 >>> R = nx.intersection_all([G1, G2])
(4)                 >>> list(R.nodes())
(4)                 [2, 3]
(4)                 >>> list(R.edges())
(4)                 [(2, 3)]
(4)                 """
(4)                 R = None
(4)                 for i, G in enumerate(graphs):
(8)                     G_nodes_set = set(G.nodes)
(8)                     G_edges_set = set(G.edges)
(8)                     if not G.is_directed():
(12)                        if G.is_multigraph():
(16)                            G_edges_set.update((v, u, k) for u, v, k in list(G_edges_set))
(12)                        else:
(16)                            G_edges_set.update((v, u) for u, v in list(G_edges_set))
(8)                     if i == 0:
(12)                        R = G.__class__()
(12)                        node_intersection = G_nodes_set
(12)                        edge_intersection = G_edges_set
(8)                     elif G.is_directed() != R.is_directed():
(12)                        raise nx.NetworkXError("All graphs must be directed or undirected.")
(8)                     elif G.is_multigraph() != R.is_multigraph():
(12)                        raise nx.NetworkXError("All graphs must be graphs or multigraphs.")
(8)                     else:
(12)                        node_intersection &= G_nodes_set
(12)                        edge_intersection &= G_edges_set
(4)                 if R is None:
(8)                     raise ValueError("cannot apply intersection_all to an empty list")
```

```
246: (4)                R.add_nodes_from(node_intersection)
247: (4)                R.add_edges_from(edge_intersection)
248: (4)                return R
```

File 284 - binary.py:

```
1: (0)              """
2: (0)              Operations on graphs including union, intersection, difference.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "union",
7: (4)                  "compose",
8: (4)                  "disjoint_union",
9: (4)                  "intersection",
10: (4)                 "difference",
11: (4)                 "symmetric_difference",
12: (4)                 "full_join",
13: (0)             ]
14: (0)             _G_H = {"G": 0, "H": 1}
15: (0)             @nx._dispatchable(graphs=_G_H, preserve_all_attrs=True, returns_graph=True)
16: (0)             def union(G, H, rename=()):
17: (4)                 """Combine graphs G and H. The names of nodes must be unique.
18: (4)                 A name collision between the graphs will raise an exception.
19: (4)                 A renaming facility is provided to avoid name collisions.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G, H : graph
23: (7)                    A NetworkX graph
24: (4)                 rename : iterable , optional
25: (7)                    Node names of G and H can be changed by specifying the tuple
26: (7)                    rename=('G-','H-') (for example).  Node "u" in G is then renamed
27: (7)                    "G-u" and "v" in H is renamed "H-v".
28: (4)                 Returns
29: (4)                 -------
30: (4)                 U : A union graph with the same type as G.
31: (4)                 See Also
32: (4)                 --------
33: (4)                 compose
34: (4)                 :func:`~networkx.Graph.update`
35: (4)                 disjoint_union
36: (4)                 Notes
37: (4)                 -----
38: (4)                 To combine graphs that have common nodes, consider compose(G, H)
39: (4)                 or the method, Graph.update().
40: (4)                 disjoint_union() is similar to union() except that it avoids name clashes
41: (4)                 by relabeling the nodes with sequential integers.
42: (4)                 Edge and node attributes are propagated from G and H to the union graph.
43: (4)                 Graph attributes are also propagated, but if they are present in both G
and H,
44: (4)                 then the value from H is used.
45: (4)                 Examples
46: (4)                 --------
47: (4)                 >>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
48: (4)                 >>> H = nx.Graph([(0, 1), (0, 3), (1, 3), (1, 2)])
49: (4)                 >>> U = nx.union(G, H, rename=("G", "H"))
50: (4)                 >>> U.nodes
51: (4)                 NodeView(('G0', 'G1', 'G2', 'H0', 'H1', 'H3', 'H2'))
52: (4)                 >>> U.edges
53: (4)                 EdgeView([('G0', 'G1'), ('G0', 'G2'), ('G1', 'G2'), ('H0', 'H1'), ('H0',
'H3'), ('H1', 'H3'), ('H1', 'H2')])
54: (4)                 """
55: (4)                 return nx.union_all([G, H], rename)
56: (0)             @nx._dispatchable(graphs=_G_H, preserve_all_attrs=True, returns_graph=True)
57: (0)             def disjoint_union(G, H):
58: (4)                 """Combine graphs G and H. The nodes are assumed to be unique (disjoint).
59: (4)                 This algorithm automatically relabels nodes to avoid name collisions.
```

```
60: (4)                 Parameters
61: (4)                 ----------
62: (4)                 G,H : graph
63: (7)                    A NetworkX graph
64: (4)                 Returns
65: (4)                 -------
66: (4)                 U : A union graph with the same type as G.
67: (4)                 See Also
68: (4)                 --------
69: (4)                 union
70: (4)                 compose
71: (4)                 :func:`~networkx.Graph.update`
72: (4)                 Notes
73: (4)                 -----
74: (4)                 A new graph is created, of the same class as G.  It is recommended
75: (4)                 that G and H be either both directed or both undirected.
76: (4)                 The nodes of G are relabeled 0 to len(G)-1, and the nodes of H are
77: (4)                 relabeled len(G) to len(G)+len(H)-1.
78: (4)                 Renumbering forces G and H to be disjoint, so no exception is ever raised
for a name collision.
79: (4)                 To preserve the check for common nodes, use union().
80: (4)                 Edge and node attributes are propagated from G and H to the union graph.
81: (4)                 Graph attributes are also propagated, but if they are present in both G
and H,
82: (4)                 then the value from H is used.
83: (4)                 To combine graphs that have common nodes, consider compose(G, H)
84: (4)                 or the method, Graph.update().
85: (4)                 Examples
86: (4)                 --------
87: (4)                 >>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
88: (4)                 >>> H = nx.Graph([(0, 3), (1, 2), (2, 3)])
89: (4)                 >>> G.nodes[0]["key1"] = 5
90: (4)                 >>> H.nodes[0]["key2"] = 10
91: (4)                 >>> U = nx.disjoint_union(G, H)
92: (4)                 >>> U.nodes(data=True)
93: (4)                 NodeDataView({0: {'key1': 5}, 1: {}, 2: {}, 3: {'key2': 10}, 4: {}, 5: {},
6: {}})
94: (4)                 >>> U.edges
95: (4)                 EdgeView([(0, 1), (0, 2), (1, 2), (3, 4), (4, 6), (5, 6)])
96: (4)                 """
97: (4)                 return nx.disjoint_union_all([G, H])
98: (0)             @nx._dispatchable(graphs=_G_H, returns_graph=True)
99: (0)             def intersection(G, H):
100: (4)                """Returns a new graph that contains only the nodes and the edges that
exist in
101: (4)                both G and H.
102: (4)                Parameters
103: (4)                ----------
104: (4)                G,H : graph
105: (7)                   A NetworkX graph. G and H can have different node sets but must be both
graphs or both multigraphs.
106: (4)                Raises
107: (4)                ------
108: (4)                NetworkXError
109: (8)                    If one is a MultiGraph and the other one is a graph.
110: (4)                Returns
111: (4)                -------
112: (4)                GH : A new graph with the same type as G.
113: (4)                Notes
114: (4)                -----
115: (4)                Attributes from the graph, nodes, and edges are not copied to the new
116: (4)                graph.  If you want a new graph of the intersection of G and H
117: (4)                with the attributes (including edge data) from G use remove_nodes_from()
118: (4)                as follows
119: (4)                >>> G = nx.path_graph(3)
120: (4)                >>> H = nx.path_graph(5)
121: (4)                >>> R = G.copy()
122: (4)                >>> R.remove_nodes_from(n for n in G if n not in H)
123: (4)                >>> R.remove_edges_from(e for e in G.edges if e not in H.edges)
```

```
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
(4)                 >>> H = nx.Graph([(0, 3), (1, 2), (2, 3)])
(4)                 >>> R = nx.intersection(G, H)
(4)                 >>> R.nodes
(4)                 NodeView((0, 1, 2))
(4)                 >>> R.edges
(4)                 EdgeView([(1, 2)])
(4)                 """
(4)                 return nx.intersection_all([G, H])
(0)             @nx._dispatchable(graphs=_G_H, returns_graph=True)
(0)             def difference(G, H):
(4)                 """Returns a new graph that contains the edges that exist in G but not in
H.
(4)                 The node sets of H and G must be the same.
(4)                 Parameters
(4)                 ----------
(4)                 G,H : graph
(7)                    A NetworkX graph. G and H must have the same node sets.
(4)                 Returns
(4)                 -------
(4)                 D : A new graph with the same type as G.
(4)                 Notes
(4)                 -----
(4)                 Attributes from the graph, nodes, and edges are not copied to the new
(4)                 graph.  If you want a new graph of the difference of G and H with
(4)                 the attributes (including edge data) from G use remove_nodes_from()
(4)                 as follows:
(4)                 >>> G = nx.path_graph(3)
(4)                 >>> H = nx.path_graph(5)
(4)                 >>> R = G.copy()
(4)                 >>> R.remove_nodes_from(n for n in G if n in H)
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3)])
(4)                 >>> H = nx.Graph([(0, 1), (1, 2), (0, 3)])
(4)                 >>> R = nx.difference(G, H)
(4)                 >>> R.nodes
(4)                 NodeView((0, 1, 2, 3))
(4)                 >>> R.edges
(4)                 EdgeView([(0, 2), (1, 3)])
(4)                 """
(4)                 if not G.is_multigraph() == H.is_multigraph():
(8)                     raise nx.NetworkXError("G and H must both be graphs or multigraphs.")
(4)                 R = nx.create_empty_copy(G, with_data=False)
(4)                 if set(G) != set(H):
(8)                     raise nx.NetworkXError("Node sets of graphs not equal")
(4)                 if G.is_multigraph():
(8)                     edges = G.edges(keys=True)
(4)                 else:
(8)                     edges = G.edges()
(4)                 for e in edges:
(8)                     if not H.has_edge(*e):
(12)                        R.add_edge(*e)
(4)                 return R
(0)             @nx._dispatchable(graphs=_G_H, returns_graph=True)
(0)             def symmetric_difference(G, H):
(4)                 """Returns new graph with edges that exist in either G or H but not both.
(4)                 The node sets of H and G must be the same.
(4)                 Parameters
(4)                 ----------
(4)                 G,H : graph
(7)                    A NetworkX graph.  G and H must have the same node sets.
(4)                 Returns
(4)                 -------
(4)                 D : A new graph with the same type as G.
(4)                 Notes
(4)                 -----
```

```
    Attributes from the graph, nodes, and edges are not copied to the new
    graph.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3)])
    >>> H = nx.Graph([(0, 1), (1, 2), (0, 3)])
    >>> R = nx.symmetric_difference(G, H)
    >>> R.nodes
    NodeView((0, 1, 2, 3))
    >>> R.edges
    EdgeView([(0, 2), (0, 3), (1, 3)])
    """
    if not G.is_multigraph() == H.is_multigraph():
        raise nx.NetworkXError("G and H must both be graphs or multigraphs.")
    R = nx.create_empty_copy(G, with_data=False)
    if set(G) != set(H):
        raise nx.NetworkXError("Node sets of graphs not equal")
    gnodes = set(G)  # set of nodes in G
    hnodes = set(H)  # set of nodes in H
    nodes = gnodes.symmetric_difference(hnodes)
    R.add_nodes_from(nodes)
    if G.is_multigraph():
        edges = G.edges(keys=True)
    else:
        edges = G.edges()
    for e in edges:
        if not H.has_edge(*e):
            R.add_edge(*e)
    if H.is_multigraph():
        edges = H.edges(keys=True)
    else:
        edges = H.edges()
    for e in edges:
        if not G.has_edge(*e):
            R.add_edge(*e)
    return R
@nx._dispatchable(graphs=_G_H, preserve_all_attrs=True, returns_graph=True)
def compose(G, H):
    """Compose graph G with H by combining nodes and edges into a single
graph.
    The node sets and edges sets do not need to be disjoint.
    Composing preserves the attributes of nodes and edges.
    Attribute values from H take precedent over attribute values from G.
    Parameters
    ----------
    G, H : graph
       A NetworkX graph
    Returns
    -------
    C: A new graph with the same type as G
    See Also
    --------
    :func:`~networkx.Graph.update`
    union
    disjoint_union
    Notes
    -----
    It is recommended that G and H be either both directed or both undirected.
    For MultiGraphs, the edges are identified by incident nodes AND edge-key.
    This can cause surprises (i.e., edge `(1, 2)` may or may not be the same
    in two graphs) if you use MultiGraph without keeping track of edge keys.
    If combining the attributes of common nodes is not desired, consider
union(),
    which raises an exception for name collisions.
    Examples
    --------
    >>> G = nx.Graph([(0, 1), (0, 2)])
    >>> H = nx.Graph([(0, 1), (1, 2)])
    >>> R = nx.compose(G, H)
```

```
259: (4)                >>> R.nodes
260: (4)                NodeView((0, 1, 2))
261: (4)                >>> R.edges
262: (4)                EdgeView([(0, 1), (0, 2), (1, 2)])
263: (4)                By default, the attributes from `H` take precedent over attributes from
`G`.
264: (4)                If you prefer another way of combining attributes, you can update them
after the compose operation:
265: (4)                >>> G = nx.Graph([(0, 1, {"weight": 2.0}), (3, 0, {"weight": 100.0})])
266: (4)                >>> H = nx.Graph([(0, 1, {"weight": 10.0}), (1, 2, {"weight": -1.0})])
267: (4)                >>> nx.set_node_attributes(G, {0: "dark", 1: "light", 3: "black"},
name="color")
268: (4)                >>> nx.set_node_attributes(H, {0: "green", 1: "orange", 2: "yellow"},
name="color")
269: (4)                >>> GcomposeH = nx.compose(G, H)
270: (4)                Normally, color attribute values of nodes of GcomposeH come from H. We can
workaround this as follows:
271: (4)                >>> node_data = {
272: (4)                ...     n: G.nodes[n]["color"] + " " + H.nodes[n]["color"] for n in
G.nodes & H.nodes
273: (4)                ... }
274: (4)                >>> nx.set_node_attributes(GcomposeH, node_data, "color")
275: (4)                >>> print(GcomposeH.nodes[0]["color"])
276: (4)                dark green
277: (4)                >>> print(GcomposeH.nodes[3]["color"])
278: (4)                black
279: (4)                Similarly, we can update edge attributes after the compose operation in a
way we prefer:
280: (4)                >>> edge_data = {
281: (4)                ...     e: G.edges[e]["weight"] * H.edges[e]["weight"] for e in G.edges &
H.edges
282: (4)                ... }
283: (4)                >>> nx.set_edge_attributes(GcomposeH, edge_data, "weight")
284: (4)                >>> print(GcomposeH.edges[(0, 1)]["weight"])
285: (4)                20.0
286: (4)                >>> print(GcomposeH.edges[(3, 0)]["weight"])
287: (4)                100.0
288: (4)                """
289: (4)                return nx.compose_all([G, H])
290: (0)            @nx._dispatchable(graphs=_G_H, preserve_all_attrs=True, returns_graph=True)
291: (0)            def full_join(G, H, rename=(None, None)):
292: (4)                """Returns the full join of graphs G and H.
293: (4)                Full join is the union of G and H in which all edges between
294: (4)                G and H are added.
295: (4)                The node sets of G and H must be disjoint,
296: (4)                otherwise an exception is raised.
297: (4)                Parameters
298: (4)                ----------
299: (4)                G, H : graph
300: (7)                   A NetworkX graph
301: (4)                rename : tuple , default=(None, None)
302: (7)                   Node names of G and H can be changed by specifying the tuple
303: (7)                   rename=('G-','H-') (for example).  Node "u" in G is then renamed
304: (7)                   "G-u" and "v" in H is renamed "H-v".
305: (4)                Returns
306: (4)                -------
307: (4)                U : The full join graph with the same type as G.
308: (4)                Notes
309: (4)                -----
310: (4)                It is recommended that G and H be either both directed or both undirected.
311: (4)                If G is directed, then edges from G to H are added as well as from H to G.
312: (4)                Note that full_join() does not produce parallel edges for MultiGraphs.
313: (4)                The full join operation of graphs G and H is the same as getting
314: (4)                their complement, performing a disjoint union, and finally getting
315: (4)                the complement of the resulting graph.
316: (4)                Graph, edge, and node attributes are propagated from G and H
317: (4)                to the union graph.  If a graph attribute is present in both
318: (4)                G and H the value from H is used.
319: (4)                Examples
```

```
320: (4)                --------
321: (4)                >>> G = nx.Graph([(0, 1), (0, 2)])
322: (4)                >>> H = nx.Graph([(3, 4)])
323: (4)                >>> R = nx.full_join(G, H, rename=("G", "H"))
324: (4)                >>> R.nodes
325: (4)                NodeView(('G0', 'G1', 'G2', 'H3', 'H4'))
326: (4)                >>> R.edges
327: (4)                EdgeView([('G0', 'G1'), ('G0', 'G2'), ('G0', 'H3'), ('G0', 'H4'), ('G1',
'H3'), ('G1', 'H4'), ('G2', 'H3'), ('G2', 'H4'), ('H3', 'H4')])
328: (4)                See Also
329: (4)                --------
330: (4)                union
331: (4)                disjoint_union
332: (4)                """
333: (4)                R = union(G, H, rename)
334: (4)                def add_prefix(graph, prefix):
335: (8)                    if prefix is None:
336: (12)                       return graph
337: (8)                    def label(x):
338: (12)                       return f"{prefix}{x}"
339: (8)                    return nx.relabel_nodes(graph, label)
340: (4)                G = add_prefix(G, rename[0])
341: (4)                H = add_prefix(H, rename[1])
342: (4)                for i in G:
343: (8)                    for j in H:
344: (12)                       R.add_edge(i, j)
345: (4)                if R.is_directed():
346: (8)                    for i in H:
347: (12)                       for j in G:
348: (16)                           R.add_edge(i, j)
349: (4)                return R

----------------------------------------

File 285 - product.py:

1: (0)              """
2: (0)              Graph products.
3: (0)              """
4: (0)              from itertools import product
5: (0)              import networkx as nx
6: (0)              from networkx.utils import not_implemented_for
7: (0)              __all__ = [
8: (4)                  "tensor_product",
9: (4)                  "cartesian_product",
10: (4)                 "lexicographic_product",
11: (4)                 "strong_product",
12: (4)                 "power",
13: (4)                 "rooted_product",
14: (4)                 "corona_product",
15: (4)                 "modular_product",
16: (0)             ]
17: (0)             _G_H = {"G": 0, "H": 1}
18: (0)             def _dict_product(d1, d2):
19: (4)                 return {k: (d1.get(k), d2.get(k)) for k in set(d1) | set(d2)}
20: (0)             def _node_product(G, H):
21: (4)                 for u, v in product(G, H):
22: (8)                     yield ((u, v), _dict_product(G.nodes[u], H.nodes[v]))
23: (0)             def _directed_edges_cross_edges(G, H):
24: (4)                 if not G.is_multigraph() and not H.is_multigraph():
25: (8)                     for u, v, c in G.edges(data=True):
26: (12)                        for x, y, d in H.edges(data=True):
27: (16)                            yield (u, x), (v, y), _dict_product(c, d)
28: (4)                 if not G.is_multigraph() and H.is_multigraph():
29: (8)                     for u, v, c in G.edges(data=True):
30: (12)                        for x, y, k, d in H.edges(data=True, keys=True):
31: (16)                            yield (u, x), (v, y), k, _dict_product(c, d)
32: (4)                 if G.is_multigraph() and not H.is_multigraph():
33: (8)                     for u, v, k, c in G.edges(data=True, keys=True):
```

```
34: (12)                        for x, y, d in H.edges(data=True):
35: (16)                            yield (u, x), (v, y), k, _dict_product(c, d)
36: (4)                 if G.is_multigraph() and H.is_multigraph():
37: (8)                     for u, v, j, c in G.edges(data=True, keys=True):
38: (12)                        for x, y, k, d in H.edges(data=True, keys=True):
39: (16)                            yield (u, x), (v, y), (j, k), _dict_product(c, d)
40: (0)             def _undirected_edges_cross_edges(G, H):
41: (4)                 if not G.is_multigraph() and not H.is_multigraph():
42: (8)                     for u, v, c in G.edges(data=True):
43: (12)                        for x, y, d in H.edges(data=True):
44: (16)                            yield (v, x), (u, y), _dict_product(c, d)
45: (4)                 if not G.is_multigraph() and H.is_multigraph():
46: (8)                     for u, v, c in G.edges(data=True):
47: (12)                        for x, y, k, d in H.edges(data=True, keys=True):
48: (16)                            yield (v, x), (u, y), k, _dict_product(c, d)
49: (4)                 if G.is_multigraph() and not H.is_multigraph():
50: (8)                     for u, v, k, c in G.edges(data=True, keys=True):
51: (12)                        for x, y, d in H.edges(data=True):
52: (16)                            yield (v, x), (u, y), k, _dict_product(c, d)
53: (4)                 if G.is_multigraph() and H.is_multigraph():
54: (8)                     for u, v, j, c in G.edges(data=True, keys=True):
55: (12)                        for x, y, k, d in H.edges(data=True, keys=True):
56: (16)                            yield (v, x), (u, y), (j, k), _dict_product(c, d)
57: (0)             def _edges_cross_nodes(G, H):
58: (4)                 if G.is_multigraph():
59: (8)                     for u, v, k, d in G.edges(data=True, keys=True):
60: (12)                        for x in H:
61: (16)                            yield (u, x), (v, x), k, d
62: (4)                 else:
63: (8)                     for u, v, d in G.edges(data=True):
64: (12)                        for x in H:
65: (16)                            if H.is_multigraph():
66: (20)                                yield (u, x), (v, x), None, d
67: (16)                            else:
68: (20)                                yield (u, x), (v, x), d
69: (0)             def _nodes_cross_edges(G, H):
70: (4)                 if H.is_multigraph():
71: (8)                     for x in G:
72: (12)                        for u, v, k, d in H.edges(data=True, keys=True):
73: (16)                            yield (x, u), (x, v), k, d
74: (4)                 else:
75: (8)                     for x in G:
76: (12)                        for u, v, d in H.edges(data=True):
77: (16)                            if G.is_multigraph():
78: (20)                                yield (x, u), (x, v), None, d
79: (16)                            else:
80: (20)                                yield (x, u), (x, v), d
81: (0)             def _edges_cross_nodes_and_nodes(G, H):
82: (4)                 if G.is_multigraph():
83: (8)                     for u, v, k, d in G.edges(data=True, keys=True):
84: (12)                        for x in H:
85: (16)                            for y in H:
86: (20)                                yield (u, x), (v, y), k, d
87: (4)                 else:
88: (8)                     for u, v, d in G.edges(data=True):
89: (12)                        for x in H:
90: (16)                            for y in H:
91: (20)                                if H.is_multigraph():
92: (24)                                    yield (u, x), (v, y), None, d
93: (20)                                else:
94: (24)                                    yield (u, x), (v, y), d
95: (0)             def _init_product_graph(G, H):
96: (4)                 if G.is_directed() != H.is_directed():
97: (8)                     msg = "G and H must be both directed or both undirected"
98: (8)                     raise nx.NetworkXError(msg)
99: (4)                 if G.is_multigraph() or H.is_multigraph():
100: (8)                    GH = nx.MultiGraph()
101: (4)                else:
102: (8)                    GH = nx.Graph()
```

```
(4)     if G.is_directed():
(8)         GH = GH.to_directed()
(4)     return GH
(0)@nx._dispatchable(graphs=_G_H, preserve_node_attrs=True, returns_graph=True)
(0)def tensor_product(G, H):
(4)     r"""Returns the tensor product of G and H.
(4)     The tensor product $P$ of the graphs $G$ and $H$ has a node set that
(4)     is the Cartesian product of the node sets, $V(P)=V(G) \times V(H)$.
(4)     $P$ has an edge $((u,v), (x,y))$ if and only if $(u,x)$ is an edge in $G$
(4)     and $(v,y)$ is an edge in $H$.
(4)     Tensor product is sometimes also referred to as the categorical product,
(4)     direct product, cardinal product or conjunction.
(4)     Parameters
(4)     ----------
(4)     G, H: graphs
(5)      Networkx graphs.
(4)     Returns
(4)     -------
(4)     P: NetworkX graph
(5)      The tensor product of G and H. P will be a multi-graph if either G
(5)      or H is a multi-graph, will be a directed if G and H are directed,
(5)      and undirected if G and H are undirected.
(4)     Raises
(4)     ------
(4)     NetworkXError
(5)      If G and H are not both directed or both undirected.
(4)     Notes
(4)     -----
(4)     Node attributes in P are two-tuple of the G and H node attributes.
(4)     Missing attributes are assigned None.
(4)     Examples
(4)     --------
(4)     >>> G = nx.Graph()
(4)     >>> H = nx.Graph()
(4)     >>> G.add_node(0, a1=True)
(4)     >>> H.add_node("a", a2="Spam")
(4)     >>> P = nx.tensor_product(G, H)
(4)     >>> list(P)
(4)     [(0, 'a')]
(4)     Edge attributes and edge keys (for multigraphs) are also copied to the
(4)     new product graph
(4)     """
(4)     GH = _init_product_graph(G, H)
(4)     GH.add_nodes_from(_node_product(G, H))
(4)     GH.add_edges_from(_directed_edges_cross_edges(G, H))
(4)     if not GH.is_directed():
(8)         GH.add_edges_from(_undirected_edges_cross_edges(G, H))
(4)     return GH
(0)@nx._dispatchable(graphs=_G_H, preserve_node_attrs=True, returns_graph=True)
(0)def cartesian_product(G, H):
(4)     r"""Returns the Cartesian product of G and H.
(4)     The Cartesian product $P$ of the graphs $G$ and $H$ has a node set that
(4)     is the Cartesian product of the node sets, $V(P)=V(G) \times V(H)$.
(4)     $P$ has an edge $((u,v),(x,y))$ if and only if either $u$ is equal to $x$
(4)     and both $v$ and $y$ are adjacent in $H$ or if $v$ is equal to $y$ and
(4)     both $u$ and $x$ are adjacent in $G$.
(4)     Parameters
(4)     ----------
(4)     G, H: graphs
(5)      Networkx graphs.
(4)     Returns
(4)     -------
(4)     P: NetworkX graph
(5)      The Cartesian product of G and H. P will be a multi-graph if either G
(5)      or H is a multi-graph. Will be a directed if G and H are directed,
(5)      and undirected if G and H are undirected.
(4)     Raises
(4)     ------
(4)     NetworkXError
```

```
(5)                 If G and H are not both directed or both undirected.
(4)                Notes
(4)                -----
(4)                Node attributes in P are two-tuple of the G and H node attributes.
(4)                Missing attributes are assigned None.
(4)                Examples
(4)                --------
(4)                >>> G = nx.Graph()
(4)                >>> H = nx.Graph()
(4)                >>> G.add_node(0, a1=True)
(4)                >>> H.add_node("a", a2="Spam")
(4)                >>> P = nx.cartesian_product(G, H)
(4)                >>> list(P)
(4)                [(0, 'a')]
(4)                Edge attributes and edge keys (for multigraphs) are also copied to the
(4)                new product graph
(4)                """
(4)                GH = _init_product_graph(G, H)
(4)                GH.add_nodes_from(_node_product(G, H))
(4)                GH.add_edges_from(_edges_cross_nodes(G, H))
(4)                GH.add_edges_from(_nodes_cross_edges(G, H))
(4)                return GH
(0)            @nx._dispatchable(graphs=_G_H, preserve_node_attrs=True, returns_graph=True)
(0)            def lexicographic_product(G, H):
(4)                r"""Returns the lexicographic product of G and H.
(4)                The lexicographical product $P$ of the graphs $G$ and $H$ has a node set
(4)                that is the Cartesian product of the node sets, $V(P)=V(G) \times V(H)$.
(4)                $P$ has an edge $((u,v), (x,y))$ if and only if $(u,v)$ is an edge in $G$
(4)                or $u==v$ and $(x,y)$ is an edge in $H$.
(4)                Parameters
(4)                ----------
(4)                G, H: graphs
(5)                 Networkx graphs.
(4)                Returns
(4)                -------
(4)                P: NetworkX graph
(5)                 The Cartesian product of G and H. P will be a multi-graph if either G
(5)                 or H is a multi-graph. Will be a directed if G and H are directed,
(5)                 and undirected if G and H are undirected.
(4)                Raises
(4)                ------
(4)                NetworkXError
(5)                 If G and H are not both directed or both undirected.
(4)                Notes
(4)                -----
(4)                Node attributes in P are two-tuple of the G and H node attributes.
(4)                Missing attributes are assigned None.
(4)                Examples
(4)                --------
(4)                >>> G = nx.Graph()
(4)                >>> H = nx.Graph()
(4)                >>> G.add_node(0, a1=True)
(4)                >>> H.add_node("a", a2="Spam")
(4)                >>> P = nx.lexicographic_product(G, H)
(4)                >>> list(P)
(4)                [(0, 'a')]
(4)                Edge attributes and edge keys (for multigraphs) are also copied to the
(4)                new product graph
(4)                """
(4)                GH = _init_product_graph(G, H)
(4)                GH.add_nodes_from(_node_product(G, H))
(4)                GH.add_edges_from(_edges_cross_nodes_and_nodes(G, H))
(4)                GH.add_edges_from(_nodes_cross_edges(G, H))
(4)                return GH
(0)            @nx._dispatchable(graphs=_G_H, preserve_node_attrs=True, returns_graph=True)
(0)            def strong_product(G, H):
(4)                r"""Returns the strong product of G and H.
(4)                The strong product $P$ of the graphs $G$ and $H$ has a node set that
(4)                is the Cartesian product of the node sets, $V(P)=V(G) \times V(H)$.
```

```
(4)                 $P$ has an edge $((u,v), (x,y))$ if and only if
(4)                 $u==v$ and $(x,y)$ is an edge in $H$, or
(4)                 $x==y$ and $(u,v)$ is an edge in $G$, or
(4)                 $(u,v)$ is an edge in $G$ and $(x,y)$ is an edge in $H$.
(4)                 Parameters
(4)                 ----------
(4)                 G, H: graphs
(5)                  Networkx graphs.
(4)                 Returns
(4)                 -------
(4)                 P: NetworkX graph
(5)                  The Cartesian product of G and H. P will be a multi-graph if either G
(5)                  or H is a multi-graph. Will be a directed if G and H are directed,
(5)                  and undirected if G and H are undirected.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(5)                  If G and H are not both directed or both undirected.
(4)                 Notes
(4)                 -----
(4)                 Node attributes in P are two-tuple of the G and H node attributes.
(4)                 Missing attributes are assigned None.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> H = nx.Graph()
(4)                 >>> G.add_node(0, a1=True)
(4)                 >>> H.add_node("a", a2="Spam")
(4)                 >>> P = nx.strong_product(G, H)
(4)                 >>> list(P)
(4)                 [(0, 'a')]
(4)                 Edge attributes and edge keys (for multigraphs) are also copied to the
(4)                 new product graph
(4)                 """
(4)                 GH = _init_product_graph(G, H)
(4)                 GH.add_nodes_from(_node_product(G, H))
(4)                 GH.add_edges_from(_nodes_cross_edges(G, H))
(4)                 GH.add_edges_from(_edges_cross_nodes(G, H))
(4)                 GH.add_edges_from(_directed_edges_cross_edges(G, H))
(4)                 if not GH.is_directed():
(8)                     GH.add_edges_from(_undirected_edges_cross_edges(G, H))
(4)                 return GH
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(returns_graph=True)
(0)             def power(G, k):
(4)                 """Returns the specified power of a graph.
(4)                 The $k$th power of a simple graph $G$, denoted $G^k$, is a
(4)                 graph on the same set of nodes in which two distinct nodes $u$ and
(4)                 $v$ are adjacent in $G^k$ if and only if the shortest path
(4)                 distance between $u$ and $v$ in $G$ is at most $k$.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX simple graph object.
(4)                 k : positive integer
(8)                     The power to which to raise the graph `G`.
(4)                 Returns
(4)                 -------
(4)                 NetworkX simple graph
(8)                     `G` to the power `k`.
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If the exponent `k` is not positive.
(4)                 NetworkXNotImplemented
(8)                     If `G` is not a simple graph.
(4)                 Examples
(4)                 --------
```

```
(4)                 The number of edges will never decrease when taking successive
(4)                 powers:
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> list(nx.power(G, 2).edges)
(4)                 [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
(4)                 >>> list(nx.power(G, 3).edges)
(4)                 [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
(4)                 The `k` th power of a cycle graph on *n* nodes is the complete graph
(4)                 on *n* nodes, if `k` is at least ``n // 2``:
(4)                 >>> G = nx.cycle_graph(5)
(4)                 >>> H = nx.complete_graph(5)
(4)                 >>> nx.is_isomorphic(nx.power(G, 2), H)
(4)                 True
(4)                 >>> G = nx.cycle_graph(8)
(4)                 >>> H = nx.complete_graph(8)
(4)                 >>> nx.is_isomorphic(nx.power(G, 4), H)
(4)                 True
(4)                 References
(4)                 ----------
(4)                 .. [1] J. A. Bondy, U. S. R. Murty, *Graph Theory*. Springer, 2008.
(4)                 Notes
(4)                 -----
(4)                 This definition of "power graph" comes from Exercise 3.1.6 of
(4)                 *Graph Theory* by Bondy and Murty [1]_.
(4)                 """
(4)                 if k <= 0:
(8)                     raise ValueError("k must be a positive integer")
(4)                 H = nx.Graph()
(4)                 H.add_nodes_from(G)
(4)                 for n in G:
(8)                     seen = {}  # level (number of hops) when seen in BFS
(8)                     level = 1  # the current level
(8)                     nextlevel = G[n]
(8)                     while nextlevel:
(12)                        thislevel = nextlevel  # advance to next level
(12)                        nextlevel = {}  # and start a new list (fringe)
(12)                        for v in thislevel:
(16)                            if v == n:  # avoid self loop
(20)                                continue
(16)                            if v not in seen:
(20)                                seen[v] = level  # set the level of vertex v
(20)                                nextlevel.update(G[v])  # add neighbors of v
(12)                        if k <= level:
(16)                            break
(12)                        level += 1
(8)                     H.add_edges_from((n, nbr) for nbr in seen)
(4)                 return H
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(graphs=_G_H, returns_graph=True)
(0)             def rooted_product(G, H, root):
(4)                 """Return the rooted product of graphs G and H rooted at root in H.
(4)                 A new graph is constructed representing the rooted product of
(4)                 the inputted graphs, G and H, with a root in H.
(4)                 A rooted product duplicates H for each nodes in G with the root
(4)                 of H corresponding to the node in G. Nodes are renamed as the direct
(4)                 product of G and H. The result is a subgraph of the cartesian product.
(4)                 Parameters
(4)                 ----------
(4)                 G,H : graph
(7)                    A NetworkX graph
(4)                 root : node
(7)                    A node in H
(4)                 Returns
(4)                 -------
(4)                 R : The rooted product of G and H with a specified root in H
(4)                 Notes
(4)                 -----
(4)                 The nodes of R are the Cartesian Product of the nodes of G and H.
(4)                 The nodes of G and H are not relabeled.
```

```
    """
    if root not in H:
        raise nx.NetworkXError("root must be a vertex in H")
    R = nx.Graph()
    R.add_nodes_from(product(G, H))
    R.add_edges_from(((e[0], root), (e[1], root)) for e in G.edges())
    R.add_edges_from(((g, e[0]), (g, e[1])) for g in G for e in H.edges())
    return R
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable(graphs=_G_H, returns_graph=True)
def corona_product(G, H):
    r"""Returns the Corona product of G and H.
    The corona product of $G$ and $H$ is the graph $C = G \circ H$ obtained by
    taking one copy of $G$, called the center graph, $|V(G)|$ copies of $H$,
    called the outer graph, and making the $i$-th vertex of $G$ adjacent to
    every vertex of the $i$-th copy of $H$, where $1 ≤ i ≤ |V(G)|$.
    Parameters
    ----------
    G, H: NetworkX graphs
        The graphs to take the carona product of.
        `G` is the center graph and `H` is the outer graph
    Returns
    -------
    C: NetworkX graph
        The Corona product of G and H.
    Raises
    ------
    NetworkXError
        If G and H are not both directed or both undirected.
    Examples
    --------
    >>> G = nx.cycle_graph(4)
    >>> H = nx.path_graph(2)
    >>> C = nx.corona_product(G, H)
    >>> list(C)
    [0, 1, 2, 3, (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1)]
    >>> print(C)
    Graph with 12 nodes and 16 edges
    References
    ----------
    [1] M. Tavakoli, F. Rahbarnia, and A. R. Ashrafi,
        "Studying the corona product of graphs under some graph invariants,"
        Transactions on Combinatorics, vol. 3, no. 3, pp. 43–49, Sep. 2014,
        doi: 10.22108/toc.2014.5542.
    [2] A. Faraji, "Corona Product in Graph Theory," Ali Faraji, May 11, 2021.
        https://blog.alifaraji.ir/math/graph-theory/corona-product.html (accessed Dec. 07, 2021).
    """
    GH = _init_product_graph(G, H)
    GH.add_nodes_from(G)
    GH.add_edges_from(G.edges)
    for G_node in G:
        GH.add_nodes_from((G_node, v) for v in H)
        GH.add_edges_from(
            ((G_node, e0), (G_node, e1), d) for e0, e1, d in H.edges.data()
        )
        GH.add_edges_from((G_node, (G_node, H_node)) for H_node in H)
    return GH
@nx._dispatchable(
    graphs=_G_H, preserve_edge_attrs=True, preserve_node_attrs=True, returns_graph=True
)
def modular_product(G, H):
    r"""Returns the Modular product of G and H.
    The modular product of `G` and `H` is the graph $M = G \nabla H$,
    consisting of the node set $V(M) = V(G) \times V(H)$ that is the Cartesian
    product of the node sets of `G` and `H`. Further, M contains an edge ((u,
```

```
v), (x, y)):
                - if u is adjacent to x in `G` and v is adjacent to y in `H`, or
                - if u is not adjacent to x in `G` and v is not adjacent to y in `H`.
                More formally::
                    E(M) = {((u, v), (x, y)) | ((u, x) in E(G) and (v, y) in E(H)) or
                                               ((u, x) not in E(G) and (v, y) not in
E(H))}
                Parameters
                ----------
                G, H: NetworkX graphs
                    The graphs to take the modular product of.
                Returns
                -------
                M: NetworkX graph
                    The Modular product of `G` and `H`.
                Raises
                ------
                NetworkXNotImplemented
                    If `G` is not a simple graph.
                Examples
                --------
                >>> G = nx.cycle_graph(4)
                >>> H = nx.path_graph(2)
                >>> M = nx.modular_product(G, H)
                >>> list(M)
                [(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1)]
                >>> print(M)
                Graph with 8 nodes and 8 edges
                Notes
                -----
                The *modular product* is defined in [1]_ and was first
                introduced as the *weak modular product*.
                The modular product reduces the problem of counting isomorphic subgraphs
                in `G` and `H` to the problem of counting cliques in M. The subgraphs of
                `G` and `H` that are induced by the nodes of a clique in M are
                isomorphic [2]_ [3]_.
                References
                ----------
                .. [1] R. Hammack, W. Imrich, and S. Klavžar,
                    "Handbook of Product Graphs", CRC Press, 2011.
                .. [2] H. G. Barrow and R. M. Burstall,
                    "Subgraph isomorphism, matching relational structures and maximal
                    cliques", Information Processing Letters, vol. 4, issue 4, pp. 83-84,
                    1976, https://doi.org/10.1016/0020-0190(76)90049-1.
                .. [3] V. G. Vizing, "Reduction of the problem of isomorphism and
isomorphic
                    entrance to the task of finding the nondensity of a graph." Proc.
Third
                    All-Union Conference on Problems of Theoretical Cybernetics. 1974.
                """
                if G.is_directed() or H.is_directed():
                    raise nx.NetworkXNotImplemented(
                        "Modular product not implemented for directed graphs"
                    )
                if G.is_multigraph() or H.is_multigraph():
                    raise nx.NetworkXNotImplemented(
                        "Modular product not implemented for multigraphs"
                    )
                GH = _init_product_graph(G, H)
                GH.add_nodes_from(_node_product(G, H))
                for u, v, c in G.edges(data=True):
                    for x, y, d in H.edges(data=True):
                        GH.add_edge((u, x), (v, y), **_dict_product(c, d))
                        GH.add_edge((v, x), (u, y), **_dict_product(c, d))
                G = nx.complement(G)
                H = nx.complement(H)
                for u, v, c in G.edges(data=True):
                    for x, y, d in H.edges(data=True):
                        GH.add_edge((u, x), (v, y), **_dict_product(c, d))
```

```
510: (12)                        GH.add_edge((v, x), (u, y), **_dict_product(c, d))
511: (4)                 return GH
```

File 286 - unary.py:

```
1: (0)               """Unary operations on graphs"""
2: (0)               import networkx as nx
3: (0)               __all__ = ["complement", "reverse"]
4: (0)               @nx._dispatchable(returns_graph=True)
5: (0)               def complement(G):
6: (4)                   """Returns the graph complement of G.
7: (4)                   Parameters
8: (4)                   ----------
9: (4)                   G : graph
10: (7)                     A NetworkX graph
11: (4)                  Returns
12: (4)                  -------
13: (4)                  GC : A new graph.
14: (4)                  Notes
15: (4)                  -----
16: (4)                  Note that `complement` does not create self-loops and also
17: (4)                  does not produce parallel edges for MultiGraphs.
18: (4)                  Graph, node, and edge data are not propagated to the new graph.
19: (4)                  Examples
20: (4)                  --------
21: (4)                  >>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (3, 4), (3, 5)])
22: (4)                  >>> G_complement = nx.complement(G)
23: (4)                  >>> G_complement.edges()  # This shows the edges of the complemented graph
24: (4)                  EdgeView([(1, 4), (1, 5), (2, 4), (2, 5), (4, 5)])
25: (4)                  """
26: (4)                  R = G.__class__()
27: (4)                  R.add_nodes_from(G)
28: (4)                  R.add_edges_from(
29: (8)                      ((n, n2) for n, nbrs in G.adjacency() for n2 in G if n2 not in nbrs if
n != n2)
30: (4)                  )
31: (4)                  return R
32: (0)               @nx._dispatchable(returns_graph=True)
33: (0)               def reverse(G, copy=True):
34: (4)                   """Returns the reverse directed graph of G.
35: (4)                   Parameters
36: (4)                   ----------
37: (4)                   G : directed graph
38: (8)                       A NetworkX directed graph
39: (4)                   copy : bool
40: (8)                       If True, then a new graph is returned. If False, then the graph is
41: (8)                       reversed in place.
42: (4)                   Returns
43: (4)                   -------
44: (4)                   H : directed graph
45: (8)                       The reversed G.
46: (4)                   Raises
47: (4)                   ------
48: (4)                   NetworkXError
49: (8)                       If graph is undirected.
50: (4)                   Examples
51: (4)                   --------
52: (4)                   >>> G = nx.DiGraph([(1, 2), (1, 3), (2, 3), (3, 4), (3, 5)])
53: (4)                   >>> G_reversed = nx.reverse(G)
54: (4)                   >>> G_reversed.edges()
55: (4)                   OutEdgeView([(2, 1), (3, 1), (3, 2), (4, 3), (5, 3)])
56: (4)                   """
57: (4)                   if not G.is_directed():
58: (8)                       raise nx.NetworkXError("Cannot reverse an undirected graph.")
59: (4)                   else:
60: (8)                       return G.reverse(copy=copy)
```

```
----------------------------------------

File 287 - test_all.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import edges_equal
4: (0)              def test_union_all_attributes():
5: (4)                  g = nx.Graph()
6: (4)                  g.add_node(0, x=4)
7: (4)                  g.add_node(1, x=5)
8: (4)                  g.add_edge(0, 1, size=5)
9: (4)                  g.graph["name"] = "g"
10: (4)                 h = g.copy()
11: (4)                 h.graph["name"] = "h"
12: (4)                 h.graph["attr"] = "attr"
13: (4)                 h.nodes[0]["x"] = 7
14: (4)                 j = g.copy()
15: (4)                 j.graph["name"] = "j"
16: (4)                 j.graph["attr"] = "attr"
17: (4)                 j.nodes[0]["x"] = 7
18: (4)                 ghj = nx.union_all([g, h, j], rename=("g", "h", "j"))
19: (4)                 assert set(ghj.nodes()) == {"h0", "h1", "g0", "g1", "j0", "j1"}
20: (4)                 for n in ghj:
21: (8)                     graph, node = n
22: (8)                     assert ghj.nodes[n] == eval(graph).nodes[int(node)]
23: (4)                 assert ghj.graph["attr"] == "attr"
24: (4)                 assert ghj.graph["name"] == "j"  # j graph attributes take precedent
25: (0)             def test_intersection_all():
26: (4)                 G = nx.Graph()
27: (4)                 H = nx.Graph()
28: (4)                 R = nx.Graph(awesome=True)
29: (4)                 G.add_nodes_from([1, 2, 3, 4])
30: (4)                 G.add_edge(1, 2)
31: (4)                 G.add_edge(2, 3)
32: (4)                 H.add_nodes_from([1, 2, 3, 4])
33: (4)                 H.add_edge(2, 3)
34: (4)                 H.add_edge(3, 4)
35: (4)                 R.add_nodes_from([1, 2, 3, 4])
36: (4)                 R.add_edge(2, 3)
37: (4)                 R.add_edge(4, 1)
38: (4)                 I = nx.intersection_all([G, H, R])
39: (4)                 assert set(I.nodes()) == {1, 2, 3, 4}
40: (4)                 assert sorted(I.edges()) == [(2, 3)]
41: (4)                 assert I.graph == {}
42: (0)             def test_intersection_all_different_node_sets():
43: (4)                 G = nx.Graph()
44: (4)                 H = nx.Graph()
45: (4)                 R = nx.Graph()
46: (4)                 G.add_nodes_from([1, 2, 3, 4, 6, 7])
47: (4)                 G.add_edge(1, 2)
48: (4)                 G.add_edge(2, 3)
49: (4)                 G.add_edge(6, 7)
50: (4)                 H.add_nodes_from([1, 2, 3, 4])
51: (4)                 H.add_edge(2, 3)
52: (4)                 H.add_edge(3, 4)
53: (4)                 R.add_nodes_from([1, 2, 3, 4, 8, 9])
54: (4)                 R.add_edge(2, 3)
55: (4)                 R.add_edge(4, 1)
56: (4)                 R.add_edge(8, 9)
57: (4)                 I = nx.intersection_all([G, H, R])
58: (4)                 assert set(I.nodes()) == {1, 2, 3, 4}
59: (4)                 assert sorted(I.edges()) == [(2, 3)]
60: (0)             def test_intersection_all_attributes():
61: (4)                 g = nx.Graph()
62: (4)                 g.add_node(0, x=4)
63: (4)                 g.add_node(1, x=5)
64: (4)                 g.add_edge(0, 1, size=5)
65: (4)                 g.graph["name"] = "g"
```

```
66: (4)                 h = g.copy()
67: (4)                 h.graph["name"] = "h"
68: (4)                 h.graph["attr"] = "attr"
69: (4)                 h.nodes[0]["x"] = 7
70: (4)                 gh = nx.intersection_all([g, h])
71: (4)                 assert set(gh.nodes()) == set(g.nodes())
72: (4)                 assert set(gh.nodes()) == set(h.nodes())
73: (4)                 assert sorted(gh.edges()) == sorted(g.edges())
74: (0)             def test_intersection_all_attributes_different_node_sets():
75: (4)                 g = nx.Graph()
76: (4)                 g.add_node(0, x=4)
77: (4)                 g.add_node(1, x=5)
78: (4)                 g.add_edge(0, 1, size=5)
79: (4)                 g.graph["name"] = "g"
80: (4)                 h = g.copy()
81: (4)                 g.add_node(2)
82: (4)                 h.graph["name"] = "h"
83: (4)                 h.graph["attr"] = "attr"
84: (4)                 h.nodes[0]["x"] = 7
85: (4)                 gh = nx.intersection_all([g, h])
86: (4)                 assert set(gh.nodes()) == set(h.nodes())
87: (4)                 assert sorted(gh.edges()) == sorted(g.edges())
88: (0)             def test_intersection_all_multigraph_attributes():
89: (4)                 g = nx.MultiGraph()
90: (4)                 g.add_edge(0, 1, key=0)
91: (4)                 g.add_edge(0, 1, key=1)
92: (4)                 g.add_edge(0, 1, key=2)
93: (4)                 h = nx.MultiGraph()
94: (4)                 h.add_edge(0, 1, key=0)
95: (4)                 h.add_edge(0, 1, key=3)
96: (4)                 gh = nx.intersection_all([g, h])
97: (4)                 assert set(gh.nodes()) == set(g.nodes())
98: (4)                 assert set(gh.nodes()) == set(h.nodes())
99: (4)                 assert sorted(gh.edges()) == [(0, 1)]
100: (4)                assert sorted(gh.edges(keys=True)) == [(0, 1, 0)]
101: (0)            def test_intersection_all_multigraph_attributes_different_node_sets():
102: (4)                g = nx.MultiGraph()
103: (4)                g.add_edge(0, 1, key=0)
104: (4)                g.add_edge(0, 1, key=1)
105: (4)                g.add_edge(0, 1, key=2)
106: (4)                g.add_edge(1, 2, key=1)
107: (4)                g.add_edge(1, 2, key=2)
108: (4)                h = nx.MultiGraph()
109: (4)                h.add_edge(0, 1, key=0)
110: (4)                h.add_edge(0, 1, key=2)
111: (4)                h.add_edge(0, 1, key=3)
112: (4)                gh = nx.intersection_all([g, h])
113: (4)                assert set(gh.nodes()) == set(h.nodes())
114: (4)                assert sorted(gh.edges()) == [(0, 1), (0, 1)]
115: (4)                assert sorted(gh.edges(keys=True)) == [(0, 1, 0), (0, 1, 2)]
116: (0)            def test_intersection_all_digraph():
117: (4)                g = nx.DiGraph()
118: (4)                g.add_edges_from([(1, 2), (2, 3)])
119: (4)                h = nx.DiGraph()
120: (4)                h.add_edges_from([(2, 1), (2, 3)])
121: (4)                gh = nx.intersection_all([g, h])
122: (4)                assert sorted(gh.edges()) == [(2, 3)]
123: (0)            def test_union_all_and_compose_all():
124: (4)                K3 = nx.complete_graph(3)
125: (4)                P3 = nx.path_graph(3)
126: (4)                G1 = nx.DiGraph()
127: (4)                G1.add_edge("A", "B")
128: (4)                G1.add_edge("A", "C")
129: (4)                G1.add_edge("A", "D")
130: (4)                G2 = nx.DiGraph()
131: (4)                G2.add_edge("1", "2")
132: (4)                G2.add_edge("1", "3")
133: (4)                G2.add_edge("1", "4")
134: (4)                G = nx.union_all([G1, G2])
```

```
(4)                 H = nx.compose_all([G1, G2])
(4)                 assert edges_equal(G.edges(), H.edges())
(4)                 assert not G.has_edge("A", "1")
(4)                 pytest.raises(nx.NetworkXError, nx.union, K3, P3)
(4)                 H1 = nx.union_all([H, G1], rename=("H", "G1"))
(4)                 assert sorted(H1.nodes()) == [
(8)                     "G1A",
(8)                     "G1B",
(8)                     "G1C",
(8)                     "G1D",
(8)                     "H1",
(8)                     "H2",
(8)                     "H3",
(8)                     "H4",
(8)                     "HA",
(8)                     "HB",
(8)                     "HC",
(8)                     "HD",
(4)                 ]
(4)                 H2 = nx.union_all([H, G2], rename=("H", ""))
(4)                 assert sorted(H2.nodes()) == [
(8)                     "1",
(8)                     "2",
(8)                     "3",
(8)                     "4",
(8)                     "H1",
(8)                     "H2",
(8)                     "H3",
(8)                     "H4",
(8)                     "HA",
(8)                     "HB",
(8)                     "HC",
(8)                     "HD",
(4)                 ]
(4)                 assert not H1.has_edge("NB", "NA")
(4)                 G = nx.compose_all([G, G])
(4)                 assert edges_equal(G.edges(), H.edges())
(4)                 G2 = nx.union_all([G2, G2], rename=("", "copy"))
(4)                 assert sorted(G2.nodes()) == [
(8)                     "1",
(8)                     "2",
(8)                     "3",
(8)                     "4",
(8)                     "copy1",
(8)                     "copy2",
(8)                     "copy3",
(8)                     "copy4",
(4)                 ]
(4)                 assert sorted(G2.neighbors("copy4")) == []
(4)                 assert sorted(G2.neighbors("copy1")) == ["copy2", "copy3", "copy4"]
(4)                 assert len(G) == 8
(4)                 assert nx.number_of_edges(G) == 6
(4)                 E = nx.disjoint_union_all([G, G])
(4)                 assert len(E) == 16
(4)                 assert nx.number_of_edges(E) == 12
(4)                 E = nx.disjoint_union_all([G1, G2])
(4)                 assert sorted(E.nodes()) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
(4)                 G1 = nx.DiGraph()
(4)                 G1.add_edge("A", "B")
(4)                 G2 = nx.DiGraph()
(4)                 G2.add_edge(1, 2)
(4)                 G3 = nx.DiGraph()
(4)                 G3.add_edge(11, 22)
(4)                 G4 = nx.union_all([G1, G2, G3], rename=("G1", "G2", "G3"))
(4)                 assert sorted(G4.nodes()) == ["G1A", "G1B", "G21", "G22", "G311", "G322"]
(0)             def test_union_all_multigraph():
(4)                 G = nx.MultiGraph()
(4)                 G.add_edge(1, 2, key=0)
(4)                 G.add_edge(1, 2, key=1)
```

```
(4)                 H = nx.MultiGraph()
(4)                 H.add_edge(3, 4, key=0)
(4)                 H.add_edge(3, 4, key=1)
(4)                 GH = nx.union_all([G, H])
(4)                 assert set(GH) == set(G) | set(H)
(4)                 assert set(GH.edges(keys=True)) == set(G.edges(keys=True)) |
set(H.edges(keys=True))
(0)             def test_input_output():
(4)                 l = [nx.Graph([(1, 2)]), nx.Graph([(3, 4)], awesome=True)]
(4)                 U = nx.disjoint_union_all(l)
(4)                 assert len(l) == 2
(4)                 assert U.graph["awesome"]
(4)                 C = nx.compose_all(l)
(4)                 assert len(l) == 2
(4)                 l = [nx.Graph([(1, 2)]), nx.Graph([(1, 2)])]
(4)                 R = nx.intersection_all(l)
(4)                 assert len(l) == 2
(0)             def test_mixed_type_union():
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     G = nx.Graph()
(8)                     H = nx.MultiGraph()
(8)                     I = nx.Graph()
(8)                     U = nx.union_all([G, H, I])
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     X = nx.Graph()
(8)                     Y = nx.DiGraph()
(8)                     XY = nx.union_all([X, Y])
(0)             def test_mixed_type_disjoint_union():
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     G = nx.Graph()
(8)                     H = nx.MultiGraph()
(8)                     I = nx.Graph()
(8)                     U = nx.disjoint_union_all([G, H, I])
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     X = nx.Graph()
(8)                     Y = nx.DiGraph()
(8)                     XY = nx.disjoint_union_all([X, Y])
(0)             def test_mixed_type_intersection():
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     G = nx.Graph()
(8)                     H = nx.MultiGraph()
(8)                     I = nx.Graph()
(8)                     U = nx.intersection_all([G, H, I])
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     X = nx.Graph()
(8)                     Y = nx.DiGraph()
(8)                     XY = nx.intersection_all([X, Y])
(0)             def test_mixed_type_compose():
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     G = nx.Graph()
(8)                     H = nx.MultiGraph()
(8)                     I = nx.Graph()
(8)                     U = nx.compose_all([G, H, I])
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     X = nx.Graph()
(8)                     Y = nx.DiGraph()
(8)                     XY = nx.compose_all([X, Y])
(0)             def test_empty_union():
(4)                 with pytest.raises(ValueError):
(8)                     nx.union_all([])
(0)             def test_empty_disjoint_union():
(4)                 with pytest.raises(ValueError):
(8)                     nx.disjoint_union_all([])
(0)             def test_empty_compose_all():
(4)                 with pytest.raises(ValueError):
(8)                     nx.compose_all([])
(0)             def test_empty_intersection_all():
(4)                 with pytest.raises(ValueError):
(8)                     nx.intersection_all([])
```

```
----------------------------------------

File 288 - __init__.py:

1: (0)

----------------------------------------

File 289 - test_binary.py:

1: (0)              import os
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.classes.tests import dispatch_interface
5: (0)              from networkx.utils import edges_equal
6: (0)              def test_union_attributes():
7: (4)                  g = nx.Graph()
8: (4)                  g.add_node(0, x=4)
9: (4)                  g.add_node(1, x=5)
10: (4)                 g.add_edge(0, 1, size=5)
11: (4)                 g.graph["name"] = "g"
12: (4)                 h = g.copy()
13: (4)                 h.graph["name"] = "h"
14: (4)                 h.graph["attr"] = "attr"
15: (4)                 h.nodes[0]["x"] = 7
16: (4)                 gh = nx.union(g, h, rename=("g", "h"))
17: (4)                 assert set(gh.nodes()) == {"h0", "h1", "g0", "g1"}
18: (4)                 for n in gh:
19: (8)                     graph, node = n
20: (8)                     assert gh.nodes[n] == eval(graph).nodes[int(node)]
21: (4)                 assert gh.graph["attr"] == "attr"
22: (4)                 assert gh.graph["name"] == "h"  # h graph attributes take precedent
23: (0)             def test_intersection():
24: (4)                 G = nx.Graph()
25: (4)                 H = nx.Graph()
26: (4)                 G.add_nodes_from([1, 2, 3, 4])
27: (4)                 G.add_edge(1, 2)
28: (4)                 G.add_edge(2, 3)
29: (4)                 H.add_nodes_from([1, 2, 3, 4])
30: (4)                 H.add_edge(2, 3)
31: (4)                 H.add_edge(3, 4)
32: (4)                 I = nx.intersection(G, H)
33: (4)                 assert set(I.nodes()) == {1, 2, 3, 4}
34: (4)                 assert sorted(I.edges()) == [(2, 3)]
35: (4)                 G2 = dispatch_interface.convert(G)
36: (4)                 H2 = dispatch_interface.convert(H)
37: (4)                 I2 = nx.intersection(G2, H2)
38: (4)                 assert set(I2.nodes()) == {1, 2, 3, 4}
39: (4)                 assert sorted(I2.edges()) == [(2, 3)]
40: (4)                 if not nx.config["backend_priority"]:
41: (8)                     with pytest.raises(TypeError):
42: (12)                        nx.intersection(G2, H)
43: (8)                     with pytest.raises(TypeError):
44: (12)                        nx.intersection(G, H2)
45: (0)             def test_intersection_node_sets_different():
46: (4)                 G = nx.Graph()
47: (4)                 H = nx.Graph()
48: (4)                 G.add_nodes_from([1, 2, 3, 4, 7])
49: (4)                 G.add_edge(1, 2)
50: (4)                 G.add_edge(2, 3)
51: (4)                 H.add_nodes_from([1, 2, 3, 4, 5, 6])
52: (4)                 H.add_edge(2, 3)
53: (4)                 H.add_edge(3, 4)
54: (4)                 H.add_edge(5, 6)
55: (4)                 I = nx.intersection(G, H)
56: (4)                 assert set(I.nodes()) == {1, 2, 3, 4}
57: (4)                 assert sorted(I.edges()) == [(2, 3)]
58: (0)             def test_intersection_attributes():
```

```
(4)                 g = nx.Graph()
(4)                 g.add_node(0, x=4)
(4)                 g.add_node(1, x=5)
(4)                 g.add_edge(0, 1, size=5)
(4)                 g.graph["name"] = "g"
(4)                 h = g.copy()
(4)                 h.graph["name"] = "h"
(4)                 h.graph["attr"] = "attr"
(4)                 h.nodes[0]["x"] = 7
(4)                 gh = nx.intersection(g, h)
(4)                 assert set(gh.nodes()) == set(g.nodes())
(4)                 assert set(gh.nodes()) == set(h.nodes())
(4)                 assert sorted(gh.edges()) == sorted(g.edges())
(0)             def test_intersection_attributes_node_sets_different():
(4)                 g = nx.Graph()
(4)                 g.add_node(0, x=4)
(4)                 g.add_node(1, x=5)
(4)                 g.add_node(2, x=3)
(4)                 g.add_edge(0, 1, size=5)
(4)                 g.graph["name"] = "g"
(4)                 h = g.copy()
(4)                 h.graph["name"] = "h"
(4)                 h.graph["attr"] = "attr"
(4)                 h.nodes[0]["x"] = 7
(4)                 h.remove_node(2)
(4)                 gh = nx.intersection(g, h)
(4)                 assert set(gh.nodes()) == set(h.nodes())
(4)                 assert sorted(gh.edges()) == sorted(g.edges())
(0)             def test_intersection_multigraph_attributes():
(4)                 g = nx.MultiGraph()
(4)                 g.add_edge(0, 1, key=0)
(4)                 g.add_edge(0, 1, key=1)
(4)                 g.add_edge(0, 1, key=2)
(4)                 h = nx.MultiGraph()
(4)                 h.add_edge(0, 1, key=0)
(4)                 h.add_edge(0, 1, key=3)
(4)                 gh = nx.intersection(g, h)
(4)                 assert set(gh.nodes()) == set(g.nodes())
(4)                 assert set(gh.nodes()) == set(h.nodes())
(4)                 assert sorted(gh.edges()) == [(0, 1)]
(4)                 assert sorted(gh.edges(keys=True)) == [(0, 1, 0)]
(0)             def test_intersection_multigraph_attributes_node_set_different():
(4)                 g = nx.MultiGraph()
(4)                 g.add_edge(0, 1, key=0)
(4)                 g.add_edge(0, 1, key=1)
(4)                 g.add_edge(0, 1, key=2)
(4)                 g.add_edge(0, 2, key=2)
(4)                 g.add_edge(0, 2, key=1)
(4)                 h = nx.MultiGraph()
(4)                 h.add_edge(0, 1, key=0)
(4)                 h.add_edge(0, 1, key=3)
(4)                 gh = nx.intersection(g, h)
(4)                 assert set(gh.nodes()) == set(h.nodes())
(4)                 assert sorted(gh.edges()) == [(0, 1)]
(4)                 assert sorted(gh.edges(keys=True)) == [(0, 1, 0)]
(0)             def test_difference():
(4)                 G = nx.Graph()
(4)                 H = nx.Graph()
(4)                 G.add_nodes_from([1, 2, 3, 4])
(4)                 G.add_edge(1, 2)
(4)                 G.add_edge(2, 3)
(4)                 H.add_nodes_from([1, 2, 3, 4])
(4)                 H.add_edge(2, 3)
(4)                 H.add_edge(3, 4)
(4)                 D = nx.difference(G, H)
(4)                 assert set(D.nodes()) == {1, 2, 3, 4}
(4)                 assert sorted(D.edges()) == [(1, 2)]
(4)                 D = nx.difference(H, G)
(4)                 assert set(D.nodes()) == {1, 2, 3, 4}
```

```
(4)             assert sorted(D.edges()) == [(3, 4)]
(4)             D = nx.symmetric_difference(G, H)
(4)             assert set(D.nodes()) == {1, 2, 3, 4}
(4)             assert sorted(D.edges()) == [(1, 2), (3, 4)]
(0)         def test_difference2():
(4)             G = nx.Graph()
(4)             H = nx.Graph()
(4)             G.add_nodes_from([1, 2, 3, 4])
(4)             H.add_nodes_from([1, 2, 3, 4])
(4)             G.add_edge(1, 2)
(4)             H.add_edge(1, 2)
(4)             G.add_edge(2, 3)
(4)             D = nx.difference(G, H)
(4)             assert set(D.nodes()) == {1, 2, 3, 4}
(4)             assert sorted(D.edges()) == [(2, 3)]
(4)             D = nx.difference(H, G)
(4)             assert set(D.nodes()) == {1, 2, 3, 4}
(4)             assert sorted(D.edges()) == []
(4)             H.add_edge(3, 4)
(4)             D = nx.difference(H, G)
(4)             assert set(D.nodes()) == {1, 2, 3, 4}
(4)             assert sorted(D.edges()) == [(3, 4)]
(0)         def test_difference_attributes():
(4)             g = nx.Graph()
(4)             g.add_node(0, x=4)
(4)             g.add_node(1, x=5)
(4)             g.add_edge(0, 1, size=5)
(4)             g.graph["name"] = "g"
(4)             h = g.copy()
(4)             h.graph["name"] = "h"
(4)             h.graph["attr"] = "attr"
(4)             h.nodes[0]["x"] = 7
(4)             gh = nx.difference(g, h)
(4)             assert set(gh.nodes()) == set(g.nodes())
(4)             assert set(gh.nodes()) == set(h.nodes())
(4)             assert sorted(gh.edges()) == []
(4)             assert gh.nodes.data() != g.nodes.data()
(4)             assert gh.graph != g.graph
(0)         def test_difference_multigraph_attributes():
(4)             g = nx.MultiGraph()
(4)             g.add_edge(0, 1, key=0)
(4)             g.add_edge(0, 1, key=1)
(4)             g.add_edge(0, 1, key=2)
(4)             h = nx.MultiGraph()
(4)             h.add_edge(0, 1, key=0)
(4)             h.add_edge(0, 1, key=3)
(4)             gh = nx.difference(g, h)
(4)             assert set(gh.nodes()) == set(g.nodes())
(4)             assert set(gh.nodes()) == set(h.nodes())
(4)             assert sorted(gh.edges()) == [(0, 1), (0, 1)]
(4)             assert sorted(gh.edges(keys=True)) == [(0, 1, 1), (0, 1, 2)]
(0)         def test_difference_raise():
(4)             G = nx.path_graph(4)
(4)             H = nx.path_graph(3)
(4)             pytest.raises(nx.NetworkXError, nx.difference, G, H)
(4)             pytest.raises(nx.NetworkXError, nx.symmetric_difference, G, H)
(0)         def test_symmetric_difference_multigraph():
(4)             g = nx.MultiGraph()
(4)             g.add_edge(0, 1, key=0)
(4)             g.add_edge(0, 1, key=1)
(4)             g.add_edge(0, 1, key=2)
(4)             h = nx.MultiGraph()
(4)             h.add_edge(0, 1, key=0)
(4)             h.add_edge(0, 1, key=3)
(4)             gh = nx.symmetric_difference(g, h)
(4)             assert set(gh.nodes()) == set(g.nodes())
(4)             assert set(gh.nodes()) == set(h.nodes())
(4)             assert sorted(gh.edges()) == 3 * [(0, 1)]
(4)             assert sorted(sorted(e) for e in gh.edges(keys=True)) == [
```

```
(8)                     [0, 1, 1],
(8)                     [0, 1, 2],
(8)                     [0, 1, 3],
(4)                 ]
(0)             def test_union_and_compose():
(4)                 K3 = nx.complete_graph(3)
(4)                 P3 = nx.path_graph(3)
(4)                 G1 = nx.DiGraph()
(4)                 G1.add_edge("A", "B")
(4)                 G1.add_edge("A", "C")
(4)                 G1.add_edge("A", "D")
(4)                 G2 = nx.DiGraph()
(4)                 G2.add_edge("1", "2")
(4)                 G2.add_edge("1", "3")
(4)                 G2.add_edge("1", "4")
(4)                 G = nx.union(G1, G2)
(4)                 H = nx.compose(G1, G2)
(4)                 assert edges_equal(G.edges(), H.edges())
(4)                 assert not G.has_edge("A", 1)
(4)                 pytest.raises(nx.NetworkXError, nx.union, K3, P3)
(4)                 H1 = nx.union(H, G1, rename=("H", "G1"))
(4)                 assert sorted(H1.nodes()) == [
(8)                     "G1A",
(8)                     "G1B",
(8)                     "G1C",
(8)                     "G1D",
(8)                     "H1",
(8)                     "H2",
(8)                     "H3",
(8)                     "H4",
(8)                     "HA",
(8)                     "HB",
(8)                     "HC",
(8)                     "HD",
(4)                 ]
(4)                 H2 = nx.union(H, G2, rename=("H", ""))
(4)                 assert sorted(H2.nodes()) == [
(8)                     "1",
(8)                     "2",
(8)                     "3",
(8)                     "4",
(8)                     "H1",
(8)                     "H2",
(8)                     "H3",
(8)                     "H4",
(8)                     "HA",
(8)                     "HB",
(8)                     "HC",
(8)                     "HD",
(4)                 ]
(4)                 assert not H1.has_edge("NB", "NA")
(4)                 G = nx.compose(G, G)
(4)                 assert edges_equal(G.edges(), H.edges())
(4)                 G2 = nx.union(G2, G2, rename=("", "copy"))
(4)                 assert sorted(G2.nodes()) == [
(8)                     "1",
(8)                     "2",
(8)                     "3",
(8)                     "4",
(8)                     "copy1",
(8)                     "copy2",
(8)                     "copy3",
(8)                     "copy4",
(4)                 ]
(4)                 assert sorted(G2.neighbors("copy4")) == []
(4)                 assert sorted(G2.neighbors("copy1")) == ["copy2", "copy3", "copy4"]
(4)                 assert len(G) == 8
(4)                 assert nx.number_of_edges(G) == 6
(4)                 E = nx.disjoint_union(G, G)
```

```
266: (4)                assert len(E) == 16
267: (4)                assert nx.number_of_edges(E) == 12
268: (4)                E = nx.disjoint_union(G1, G2)
269: (4)                assert sorted(E.nodes()) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
270: (4)                G = nx.Graph()
271: (4)                H = nx.Graph()
272: (4)                G.add_nodes_from([(1, {"a1": 1})])
273: (4)                H.add_nodes_from([(1, {"b1": 1})])
274: (4)                R = nx.compose(G, H)
275: (4)                assert R.nodes == {1: {"a1": 1, "b1": 1}}
276: (0)            def test_union_multigraph():
277: (4)                G = nx.MultiGraph()
278: (4)                G.add_edge(1, 2, key=0)
279: (4)                G.add_edge(1, 2, key=1)
280: (4)                H = nx.MultiGraph()
281: (4)                H.add_edge(3, 4, key=0)
282: (4)                H.add_edge(3, 4, key=1)
283: (4)                GH = nx.union(G, H)
284: (4)                assert set(GH) == set(G) | set(H)
285: (4)                assert set(GH.edges(keys=True)) == set(G.edges(keys=True)) |
set(H.edges(keys=True))
286: (0)            def test_disjoint_union_multigraph():
287: (4)                G = nx.MultiGraph()
288: (4)                G.add_edge(0, 1, key=0)
289: (4)                G.add_edge(0, 1, key=1)
290: (4)                H = nx.MultiGraph()
291: (4)                H.add_edge(2, 3, key=0)
292: (4)                H.add_edge(2, 3, key=1)
293: (4)                GH = nx.disjoint_union(G, H)
294: (4)                assert set(GH) == set(G) | set(H)
295: (4)                assert set(GH.edges(keys=True)) == set(G.edges(keys=True)) |
set(H.edges(keys=True))
296: (0)            def test_compose_multigraph():
297: (4)                G = nx.MultiGraph()
298: (4)                G.add_edge(1, 2, key=0)
299: (4)                G.add_edge(1, 2, key=1)
300: (4)                H = nx.MultiGraph()
301: (4)                H.add_edge(3, 4, key=0)
302: (4)                H.add_edge(3, 4, key=1)
303: (4)                GH = nx.compose(G, H)
304: (4)                assert set(GH) == set(G) | set(H)
305: (4)                assert set(GH.edges(keys=True)) == set(G.edges(keys=True)) |
set(H.edges(keys=True))
306: (4)                H.add_edge(1, 2, key=2)
307: (4)                GH = nx.compose(G, H)
308: (4)                assert set(GH) == set(G) | set(H)
309: (4)                assert set(GH.edges(keys=True)) == set(G.edges(keys=True)) |
set(H.edges(keys=True))
310: (0)            def test_full_join_graph():
311: (4)                G = nx.Graph()
312: (4)                G.add_node(0)
313: (4)                G.add_edge(1, 2)
314: (4)                H = nx.Graph()
315: (4)                H.add_edge(3, 4)
316: (4)                U = nx.full_join(G, H)
317: (4)                assert set(U) == set(G) | set(H)
318: (4)                assert len(U) == len(G) + len(H)
319: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
320: (4)                U = nx.full_join(G, H, rename=("g", "h"))
321: (4)                assert set(U) == {"g0", "g1", "g2", "h3", "h4"}
322: (4)                assert len(U) == len(G) + len(H)
323: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
324: (4)                G = nx.Graph()
325: (4)                G.add_node("a")
326: (4)                G.add_edge("b", "c")
327: (4)                H = nx.Graph()
328: (4)                H.add_edge("d", "e")
329: (4)                U = nx.full_join(G, H, rename=("g", "h"))
330: (4)                assert set(U) == {"ga", "gb", "gc", "hd", "he"}
```

```
331: (4)                assert len(U) == len(G) + len(H)
332: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
333: (4)                G = nx.DiGraph()
334: (4)                G.add_node(0)
335: (4)                G.add_edge(1, 2)
336: (4)                H = nx.DiGraph()
337: (4)                H.add_edge(3, 4)
338: (4)                U = nx.full_join(G, H)
339: (4)                assert set(U) == set(G) | set(H)
340: (4)                assert len(U) == len(G) + len(H)
341: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
* 2
342: (4)                U = nx.full_join(G, H, rename=("g", "h"))
343: (4)                assert set(U) == {"g0", "g1", "g2", "h3", "h4"}
344: (4)                assert len(U) == len(G) + len(H)
345: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
* 2
346: (0)            def test_full_join_multigraph():
347: (4)                G = nx.MultiGraph()
348: (4)                G.add_node(0)
349: (4)                G.add_edge(1, 2)
350: (4)                H = nx.MultiGraph()
351: (4)                H.add_edge(3, 4)
352: (4)                U = nx.full_join(G, H)
353: (4)                assert set(U) == set(G) | set(H)
354: (4)                assert len(U) == len(G) + len(H)
355: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
356: (4)                U = nx.full_join(G, H, rename=("g", "h"))
357: (4)                assert set(U) == {"g0", "g1", "g2", "h3", "h4"}
358: (4)                assert len(U) == len(G) + len(H)
359: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
360: (4)                G = nx.MultiDiGraph()
361: (4)                G.add_node(0)
362: (4)                G.add_edge(1, 2)
363: (4)                H = nx.MultiDiGraph()
364: (4)                H.add_edge(3, 4)
365: (4)                U = nx.full_join(G, H)
366: (4)                assert set(U) == set(G) | set(H)
367: (4)                assert len(U) == len(G) + len(H)
368: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
* 2
369: (4)                U = nx.full_join(G, H, rename=("g", "h"))
370: (4)                assert set(U) == {"g0", "g1", "g2", "h3", "h4"}
371: (4)                assert len(U) == len(G) + len(H)
372: (4)                assert len(U.edges()) == len(G.edges()) + len(H.edges()) + len(G) * len(H)
* 2
373: (0)            def test_mixed_type_union():
374: (4)                G = nx.Graph()
375: (4)                H = nx.MultiGraph()
376: (4)                pytest.raises(nx.NetworkXError, nx.union, G, H)
377: (4)                pytest.raises(nx.NetworkXError, nx.disjoint_union, G, H)
378: (4)                pytest.raises(nx.NetworkXError, nx.intersection, G, H)
379: (4)                pytest.raises(nx.NetworkXError, nx.difference, G, H)
380: (4)                pytest.raises(nx.NetworkXError, nx.symmetric_difference, G, H)
381: (4)                pytest.raises(nx.NetworkXError, nx.compose, G, H)

----------------------------------------

File 290 - test_product.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import edges_equal
4: (0)              def test_tensor_product_raises():
5: (4)                  with pytest.raises(nx.NetworkXError):
6: (8)                      P = nx.tensor_product(nx.DiGraph(), nx.Graph())
7: (0)              def test_tensor_product_null():
8: (4)                  null = nx.null_graph()
9: (4)                  empty10 = nx.empty_graph(10)
```

```
(4)                 K3 = nx.complete_graph(3)
(4)                 K10 = nx.complete_graph(10)
(4)                 P3 = nx.path_graph(3)
(4)                 P10 = nx.path_graph(10)
(4)                 G = nx.tensor_product(null, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(null, empty10)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(null, K3)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(null, K10)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(null, P3)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(null, P10)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(empty10, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(K3, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(K10, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(P3, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.tensor_product(P10, null)
(4)                 assert nx.is_isomorphic(G, null)
(0)             def test_tensor_product_size():
(4)                 P5 = nx.path_graph(5)
(4)                 K3 = nx.complete_graph(3)
(4)                 K5 = nx.complete_graph(5)
(4)                 G = nx.tensor_product(P5, K3)
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.tensor_product(K3, K5)
(4)                 assert nx.number_of_nodes(G) == 3 * 5
(0)             def test_tensor_product_combinations():
(4)                 P5 = nx.path_graph(5)
(4)                 K3 = nx.complete_graph(3)
(4)                 G = nx.tensor_product(P5, K3)
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.tensor_product(P5, nx.MultiGraph(K3))
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.tensor_product(nx.MultiGraph(P5), K3)
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.tensor_product(nx.MultiGraph(P5), nx.MultiGraph(K3))
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.tensor_product(nx.DiGraph(P5), nx.DiGraph(K3))
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(0)             def test_tensor_product_classic_result():
(4)                 K2 = nx.complete_graph(2)
(4)                 G = nx.petersen_graph()
(4)                 G = nx.tensor_product(G, K2)
(4)                 assert nx.is_isomorphic(G, nx.desargues_graph())
(4)                 G = nx.cycle_graph(5)
(4)                 G = nx.tensor_product(G, K2)
(4)                 assert nx.is_isomorphic(G, nx.cycle_graph(10))
(4)                 G = nx.tetrahedral_graph()
(4)                 G = nx.tensor_product(G, K2)
(4)                 assert nx.is_isomorphic(G, nx.cubical_graph())
(0)             def test_tensor_product_random():
(4)                 G = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)                 H = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)                 GH = nx.tensor_product(G, H)
(4)                 for u_G, u_H in GH.nodes():
(8)                     for v_G, v_H in GH.nodes():
(12)                        if H.has_edge(u_H, v_H) and G.has_edge(u_G, v_G):
(16)                            assert GH.has_edge((u_G, u_H), (v_G, v_H))
(12)                        else:
(16)                            assert not GH.has_edge((u_G, u_H), (v_G, v_H))
(0)             def test_cartesian_product_multigraph():
```

```
 79: (4)                 G = nx.MultiGraph()
 80: (4)                 G.add_edge(1, 2, key=0)
 81: (4)                 G.add_edge(1, 2, key=1)
 82: (4)                 H = nx.MultiGraph()
 83: (4)                 H.add_edge(3, 4, key=0)
 84: (4)                 H.add_edge(3, 4, key=1)
 85: (4)                 GH = nx.cartesian_product(G, H)
 86: (4)                 assert set(GH) == {(1, 3), (2, 3), (2, 4), (1, 4)}
 87: (4)                 assert {(frozenset([u, v]), k) for u, v, k in GH.edges(keys=True)} == {
 88: (8)                     (frozenset([u, v]), k)
 89: (8)                     for u, v, k in [
 90: (12)                        ((1, 3), (2, 3), 0),
 91: (12)                        ((1, 3), (2, 3), 1),
 92: (12)                        ((1, 3), (1, 4), 0),
 93: (12)                        ((1, 3), (1, 4), 1),
 94: (12)                        ((2, 3), (2, 4), 0),
 95: (12)                        ((2, 3), (2, 4), 1),
 96: (12)                        ((2, 4), (1, 4), 0),
 97: (12)                        ((2, 4), (1, 4), 1),
 98: (8)                     ]
 99: (4)                 }
 100: (0)            def test_cartesian_product_raises():
 101: (4)                with pytest.raises(nx.NetworkXError):
 102: (8)                    P = nx.cartesian_product(nx.DiGraph(), nx.Graph())
 103: (0)            def test_cartesian_product_null():
 104: (4)                null = nx.null_graph()
 105: (4)                empty10 = nx.empty_graph(10)
 106: (4)                K3 = nx.complete_graph(3)
 107: (4)                K10 = nx.complete_graph(10)
 108: (4)                P3 = nx.path_graph(3)
 109: (4)                P10 = nx.path_graph(10)
 110: (4)                G = nx.cartesian_product(null, null)
 111: (4)                assert nx.is_isomorphic(G, null)
 112: (4)                G = nx.cartesian_product(null, empty10)
 113: (4)                assert nx.is_isomorphic(G, null)
 114: (4)                G = nx.cartesian_product(null, K3)
 115: (4)                assert nx.is_isomorphic(G, null)
 116: (4)                G = nx.cartesian_product(null, K10)
 117: (4)                assert nx.is_isomorphic(G, null)
 118: (4)                G = nx.cartesian_product(null, P3)
 119: (4)                assert nx.is_isomorphic(G, null)
 120: (4)                G = nx.cartesian_product(null, P10)
 121: (4)                assert nx.is_isomorphic(G, null)
 122: (4)                G = nx.cartesian_product(empty10, null)
 123: (4)                assert nx.is_isomorphic(G, null)
 124: (4)                G = nx.cartesian_product(K3, null)
 125: (4)                assert nx.is_isomorphic(G, null)
 126: (4)                G = nx.cartesian_product(K10, null)
 127: (4)                assert nx.is_isomorphic(G, null)
 128: (4)                G = nx.cartesian_product(P3, null)
 129: (4)                assert nx.is_isomorphic(G, null)
 130: (4)                G = nx.cartesian_product(P10, null)
 131: (4)                assert nx.is_isomorphic(G, null)
 132: (0)            def test_cartesian_product_size():
 133: (4)                K5 = nx.complete_graph(5)
 134: (4)                P5 = nx.path_graph(5)
 135: (4)                K3 = nx.complete_graph(3)
 136: (4)                G = nx.cartesian_product(P5, K3)
 137: (4)                assert nx.number_of_nodes(G) == 5 * 3
 138: (4)                assert nx.number_of_edges(G) == nx.number_of_edges(P5) *
nx.number_of_nodes(
 139: (8)                    K3
 140: (4)                ) + nx.number_of_edges(K3) * nx.number_of_nodes(P5)
 141: (4)                G = nx.cartesian_product(K3, K5)
 142: (4)                assert nx.number_of_nodes(G) == 3 * 5
 143: (4)                assert nx.number_of_edges(G) == nx.number_of_edges(K5) *
nx.number_of_nodes(
 144: (8)                    K3
 145: (4)                ) + nx.number_of_edges(K3) * nx.number_of_nodes(K5)
```

```
(0)  def test_cartesian_product_classic():
(4)      P2 = nx.path_graph(2)
(4)      P3 = nx.path_graph(3)
(4)      G = nx.cartesian_product(P2, P2)
(4)      G = nx.cartesian_product(P2, G)
(4)      assert nx.is_isomorphic(G, nx.cubical_graph())
(4)      G = nx.cartesian_product(P3, P3)
(4)      assert nx.is_isomorphic(G, nx.grid_2d_graph(3, 3))
(0)  def test_cartesian_product_random():
(4)      G = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)      H = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)      GH = nx.cartesian_product(G, H)
(4)      for u_G, u_H in GH.nodes():
(8)          for v_G, v_H in GH.nodes():
(12)             if (u_G == v_G and H.has_edge(u_H, v_H)) or (
(16)                 u_H == v_H and G.has_edge(u_G, v_G)
(12)             ):
(16)                 assert GH.has_edge((u_G, u_H), (v_G, v_H))
(12)             else:
(16)                 assert not GH.has_edge((u_G, u_H), (v_G, v_H))
(0)  def test_lexicographic_product_raises():
(4)      with pytest.raises(nx.NetworkXError):
(8)          P = nx.lexicographic_product(nx.DiGraph(), nx.Graph())
(0)  def test_lexicographic_product_null():
(4)      null = nx.null_graph()
(4)      empty10 = nx.empty_graph(10)
(4)      K3 = nx.complete_graph(3)
(4)      K10 = nx.complete_graph(10)
(4)      P3 = nx.path_graph(3)
(4)      P10 = nx.path_graph(10)
(4)      G = nx.lexicographic_product(null, null)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(null, empty10)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(null, K3)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(null, K10)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(null, P3)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(null, P10)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(empty10, null)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(K3, null)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(K10, null)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(P3, null)
(4)      assert nx.is_isomorphic(G, null)
(4)      G = nx.lexicographic_product(P10, null)
(4)      assert nx.is_isomorphic(G, null)
(0)  def test_lexicographic_product_size():
(4)      K5 = nx.complete_graph(5)
(4)      P5 = nx.path_graph(5)
(4)      K3 = nx.complete_graph(3)
(4)      G = nx.lexicographic_product(P5, K3)
(4)      assert nx.number_of_nodes(G) == 5 * 3
(4)      G = nx.lexicographic_product(K3, K5)
(4)      assert nx.number_of_nodes(G) == 3 * 5
(0)  def test_lexicographic_product_combinations():
(4)      P5 = nx.path_graph(5)
(4)      K3 = nx.complete_graph(3)
(4)      G = nx.lexicographic_product(P5, K3)
(4)      assert nx.number_of_nodes(G) == 5 * 3
(4)      G = nx.lexicographic_product(nx.MultiGraph(P5), K3)
(4)      assert nx.number_of_nodes(G) == 5 * 3
(4)      G = nx.lexicographic_product(P5, nx.MultiGraph(K3))
(4)      assert nx.number_of_nodes(G) == 5 * 3
```

```
(4)                 G = nx.lexicographic_product(nx.MultiGraph(P5), nx.MultiGraph(K3))
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(0)             def test_lexicographic_product_random():
(4)                 G = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)                 H = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)                 GH = nx.lexicographic_product(G, H)
(4)                 for u_G, u_H in GH.nodes():
(8)                     for v_G, v_H in GH.nodes():
(12)                        if G.has_edge(u_G, v_G) or (u_G == v_G and H.has_edge(u_H, v_H)):
(16)                            assert GH.has_edge((u_G, u_H), (v_G, v_H))
(12)                        else:
(16)                            assert not GH.has_edge((u_G, u_H), (v_G, v_H))
(0)             def test_strong_product_raises():
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     P = nx.strong_product(nx.DiGraph(), nx.Graph())
(0)             def test_strong_product_null():
(4)                 null = nx.null_graph()
(4)                 empty10 = nx.empty_graph(10)
(4)                 K3 = nx.complete_graph(3)
(4)                 K10 = nx.complete_graph(10)
(4)                 P3 = nx.path_graph(3)
(4)                 P10 = nx.path_graph(10)
(4)                 G = nx.strong_product(null, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(null, empty10)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(null, K3)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(null, K10)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(null, P3)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(null, P10)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(empty10, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(K3, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(K10, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(P3, null)
(4)                 assert nx.is_isomorphic(G, null)
(4)                 G = nx.strong_product(P10, null)
(4)                 assert nx.is_isomorphic(G, null)
(0)             def test_strong_product_size():
(4)                 K5 = nx.complete_graph(5)
(4)                 P5 = nx.path_graph(5)
(4)                 K3 = nx.complete_graph(3)
(4)                 G = nx.strong_product(P5, K3)
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.strong_product(K3, K5)
(4)                 assert nx.number_of_nodes(G) == 3 * 5
(0)             def test_strong_product_combinations():
(4)                 P5 = nx.path_graph(5)
(4)                 K3 = nx.complete_graph(3)
(4)                 G = nx.strong_product(P5, K3)
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.strong_product(nx.MultiGraph(P5), K3)
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.strong_product(P5, nx.MultiGraph(K3))
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(4)                 G = nx.strong_product(nx.MultiGraph(P5), nx.MultiGraph(K3))
(4)                 assert nx.number_of_nodes(G) == 5 * 3
(0)             def test_strong_product_random():
(4)                 G = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)                 H = nx.erdos_renyi_graph(10, 2 / 10.0)
(4)                 GH = nx.strong_product(G, H)
(4)                 for u_G, u_H in GH.nodes():
(8)                     for v_G, v_H in GH.nodes():
```

```
(12)                        if (
(16)                            (u_G == v_G and H.has_edge(u_H, v_H))
(16)                            or (u_H == v_H and G.has_edge(u_G, v_G))
(16)                            or (G.has_edge(u_G, v_G) and H.has_edge(u_H, v_H))
(12)                        ):
(16)                            assert GH.has_edge((u_G, u_H), (v_G, v_H))
(12)                        else:
(16)                            assert not GH.has_edge((u_G, u_H), (v_G, v_H))
(0)             def test_graph_power_raises():
(4)                 with pytest.raises(nx.NetworkXNotImplemented):
(8)                     nx.power(nx.MultiDiGraph(), 2)
(0)             def test_graph_power():
(4)                 G = nx.cycle_graph(7)
(4)                 G.add_edge(6, 7)
(4)                 G.add_edge(7, 8)
(4)                 G.add_edge(8, 9)
(4)                 G.add_edge(9, 2)
(4)                 H = nx.power(G, 2)
(4)                 assert edges_equal(
(8)                     list(H.edges()),
(8)                     [
(12)                        (0, 1),
(12)                        (0, 2),
(12)                        (0, 5),
(12)                        (0, 6),
(12)                        (0, 7),
(12)                        (1, 9),
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (1, 6),
(12)                        (2, 3),
(12)                        (2, 4),
(12)                        (2, 8),
(12)                        (2, 9),
(12)                        (3, 4),
(12)                        (3, 5),
(12)                        (3, 9),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (5, 6),
(12)                        (5, 7),
(12)                        (6, 7),
(12)                        (6, 8),
(12)                        (7, 8),
(12)                        (7, 9),
(12)                        (8, 9),
(8)                     ],
(4)                 )
(0)             def test_graph_power_negative():
(4)                 with pytest.raises(ValueError):
(8)                     nx.power(nx.Graph(), -1)
(0)             def test_rooted_product_raises():
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     nx.rooted_product(nx.Graph(), nx.path_graph(2), 10)
(0)             def test_rooted_product():
(4)                 G = nx.cycle_graph(5)
(4)                 H = nx.Graph()
(4)                 H.add_edges_from([("a", "b"), ("b", "c"), ("b", "d")])
(4)                 R = nx.rooted_product(G, H, "a")
(4)                 assert len(R) == len(G) * len(H)
(4)                 assert R.size() == G.size() + len(G) * H.size()
(0)             def test_corona_product():
(4)                 G = nx.cycle_graph(3)
(4)                 H = nx.path_graph(2)
(4)                 C = nx.corona_product(G, H)
(4)                 assert len(C) == (len(G) * len(H)) + len(G)
(4)                 assert C.size() == G.size() + len(G) * H.size() + len(G) * len(H)
(0)             def test_modular_product():
(4)                 G = nx.path_graph(3)
```

```
353: (4)                H = nx.path_graph(4)
354: (4)                M = nx.modular_product(G, H)
355: (4)                assert len(M) == len(G) * len(H)
356: (4)                assert edges_equal(
357: (8)                    list(M.edges()),
358: (8)                    [
359: (12)                       ((0, 0), (1, 1)),
360: (12)                       ((0, 0), (2, 2)),
361: (12)                       ((0, 0), (2, 3)),
362: (12)                       ((0, 1), (1, 0)),
363: (12)                       ((0, 1), (1, 2)),
364: (12)                       ((0, 1), (2, 3)),
365: (12)                       ((0, 2), (1, 1)),
366: (12)                       ((0, 2), (1, 3)),
367: (12)                       ((0, 2), (2, 0)),
368: (12)                       ((0, 3), (1, 2)),
369: (12)                       ((0, 3), (2, 0)),
370: (12)                       ((0, 3), (2, 1)),
371: (12)                       ((1, 0), (2, 1)),
372: (12)                       ((1, 1), (2, 0)),
373: (12)                       ((1, 1), (2, 2)),
374: (12)                       ((1, 2), (2, 1)),
375: (12)                       ((1, 2), (2, 3)),
376: (12)                       ((1, 3), (2, 2)),
377: (8)                    ],
378: (4)                )
379: (0)            def test_modular_product_raises():
380: (4)                G = nx.Graph([(0, 1), (1, 2), (2, 0)])
381: (4)                H = nx.Graph([(0, 1), (1, 2), (2, 0)])
382: (4)                DG = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
383: (4)                DH = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
384: (4)                with pytest.raises(nx.NetworkXNotImplemented):
385: (8)                    nx.modular_product(G, DH)
386: (4)                with pytest.raises(nx.NetworkXNotImplemented):
387: (8)                    nx.modular_product(DG, H)
388: (4)                with pytest.raises(nx.NetworkXNotImplemented):
389: (8)                    nx.modular_product(DG, DH)
390: (4)                MG = nx.MultiGraph([(0, 1), (1, 2), (2, 0), (0, 1)])
391: (4)                MH = nx.MultiGraph([(0, 1), (1, 2), (2, 0), (0, 1)])
392: (4)                with pytest.raises(nx.NetworkXNotImplemented):
393: (8)                    nx.modular_product(G, MH)
394: (4)                with pytest.raises(nx.NetworkXNotImplemented):
395: (8)                    nx.modular_product(MG, H)
396: (4)                with pytest.raises(nx.NetworkXNotImplemented):
397: (8)                    nx.modular_product(MG, MH)
398: (4)                with pytest.raises(nx.NetworkXNotImplemented):
399: (8)                    nx.modular_product(nx.MultiGraph(G), H)

----------------------------------------

File 291 - test_unary.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_complement():
4: (4)                  null = nx.null_graph()
5: (4)                  empty1 = nx.empty_graph(1)
6: (4)                  empty10 = nx.empty_graph(10)
7: (4)                  K3 = nx.complete_graph(3)
8: (4)                  K5 = nx.complete_graph(5)
9: (4)                  K10 = nx.complete_graph(10)
10: (4)                 P2 = nx.path_graph(2)
11: (4)                 P3 = nx.path_graph(3)
12: (4)                 P5 = nx.path_graph(5)
13: (4)                 P10 = nx.path_graph(10)
14: (4)                 G = nx.complement(K3)
15: (4)                 assert nx.is_isomorphic(G, nx.empty_graph(3))
16: (4)                 G = nx.complement(K5)
17: (4)                 assert nx.is_isomorphic(G, nx.empty_graph(5))
```

```
18: (4)                 P3cc = nx.complement(nx.complement(P3))
19: (4)                 assert nx.is_isomorphic(P3, P3cc)
20: (4)                 nullcc = nx.complement(nx.complement(null))
21: (4)                 assert nx.is_isomorphic(null, nullcc)
22: (4)                 b = nx.bull_graph()
23: (4)                 bcc = nx.complement(nx.complement(b))
24: (4)                 assert nx.is_isomorphic(b, bcc)
25: (0)             def test_complement_2():
26: (4)                 G1 = nx.DiGraph()
27: (4)                 G1.add_edge("A", "B")
28: (4)                 G1.add_edge("A", "C")
29: (4)                 G1.add_edge("A", "D")
30: (4)                 G1C = nx.complement(G1)
31: (4)                 assert sorted(G1C.edges()) == [
32: (8)                     ("B", "A"),
33: (8)                     ("B", "C"),
34: (8)                     ("B", "D"),
35: (8)                     ("C", "A"),
36: (8)                     ("C", "B"),
37: (8)                     ("C", "D"),
38: (8)                     ("D", "A"),
39: (8)                     ("D", "B"),
40: (8)                     ("D", "C"),
41: (4)                 ]
42: (0)             def test_reverse1():
43: (4)                 G1 = nx.Graph()
44: (4)                 pytest.raises(nx.NetworkXError, nx.reverse, G1)

----------------------------------------

File 292 - __init__.py:

1: (0)              from networkx.algorithms.shortest_paths.generic import *
2: (0)              from networkx.algorithms.shortest_paths.unweighted import *
3: (0)              from networkx.algorithms.shortest_paths.weighted import *
4: (0)              from networkx.algorithms.shortest_paths.astar import *
5: (0)              from networkx.algorithms.shortest_paths.dense import *

----------------------------------------

File 293 - astar.py:

1: (0)              """Shortest paths and path lengths using the A* ("A star") algorithm.
2: (0)              """
3: (0)              from heapq import heappop, heappush
4: (0)              from itertools import count
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.shortest_paths.weighted import _weight_function
7: (0)              __all__ = ["astar_path", "astar_path_length"]
8: (0)              @nx._dispatchable(edge_attrs="weight", preserve_node_attrs="heuristic")
9: (0)              def astar_path(G, source, target, heuristic=None, weight="weight", *,
cutoff=None):
10: (4)                 """Returns a list of nodes in a shortest path between source and target
11: (4)                 using the A* ("A-star") algorithm.
12: (4)                 There may be more than one shortest path.  This returns only one.
13: (4)                 Parameters
14: (4)                 ----------
15: (4)                 G : NetworkX graph
16: (4)                 source : node
17: (7)                    Starting node for path
18: (4)                 target : node
19: (7)                    Ending node for path
20: (4)                 heuristic : function
21: (7)                    A function to evaluate the estimate of the distance
22: (7)                    from the a node to the target.  The function takes
23: (7)                    two nodes arguments and must return a number.
24: (7)                    If the heuristic is inadmissible (if it might
25: (7)                    overestimate the cost of reaching the goal from a node),
26: (7)                    the result may not be a shortest path.
```

```
(7)                 The algorithm does not support updating heuristic
(7)                 values for the same node due to caching the first
(7)                 heuristic calculation per node.
(4)              weight : string or function
(7)                 If this is a string, then edge weights will be accessed via the
(7)                 edge attribute with this key (that is, the weight of the edge
(7)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(7)                 such edge attribute exists, the weight of the edge is assumed to
(7)                 be one.
(7)                 If this is a function, the weight of an edge is the value
(7)                 returned by the function. The function must accept exactly three
(7)                 positional arguments: the two endpoints of an edge and the
(7)                 dictionary of edge attributes for that edge. The function must
(7)                 return a number or None to indicate a hidden edge.
(4)              cutoff : float, optional
(7)                 If this is provided, the search will be bounded to this value. I.e. if
(7)                 the evaluation function surpasses this value for a node n, the node
will not
(7)                 be expanded further and will be ignored. More formally, let h'(n) be
the
(7)                 heuristic function, and g(n) be the cost of reaching n from the source
node. Then,
(7)                 if g(n) + h'(n) > cutoff, the node will not be explored further.
(7)                 Note that if the heuristic is inadmissible, it is possible that paths
(7)                 are ignored even though they satisfy the cutoff.
(4)              Raises
(4)              ------
(4)              NetworkXNoPath
(8)                  If no path exists between source and target.
(4)              Examples
(4)              --------
(4)              >>> G = nx.path_graph(5)
(4)              >>> print(nx.astar_path(G, 0, 4))
(4)              [0, 1, 2, 3, 4]
(4)              >>> G = nx.grid_graph(dim=[3, 3])  # nodes are two-tuples (x,y)
(4)              >>> nx.set_edge_attributes(G, {e: e[1][0] * 2 for e in G.edges()}, "cost")
(4)              >>> def dist(a, b):
(4)              ...     (x1, y1) = a
(4)              ...     (x2, y2) = b
(4)              ...     return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
(4)              >>> print(nx.astar_path(G, (0, 0), (2, 2), heuristic=dist, weight="cost"))
(4)              [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]
(4)              Notes
(4)              -----
(4)              Edge weight attributes must be numerical.
(4)              Distances are calculated as sums of weighted edges traversed.
(4)              The weight function can be used to hide edges by returning None.
(4)              So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
(4)              will find the shortest red path.
(4)              See Also
(4)              --------
(4)              shortest_path, dijkstra_path
(4)              """
(4)              if source not in G or target not in G:
(8)                  msg = f"Either source {source} or target {target} is not in G"
(8)                  raise nx.NodeNotFound(msg)
(4)              if heuristic is None:
(8)                  def heuristic(u, v):
(12)                     return 0
(4)              push = heappush
(4)              pop = heappop
(4)              weight = _weight_function(G, weight)
(4)              G_succ = G._adj  # For speed-up (and works for both directed and
undirected graphs)
(4)              c = count()
(4)              queue = [(0, next(c), source, 0, None)]
(4)              enqueued = {}
(4)              explored = {}
(4)              while queue:
```

```
92: (8)                     _, __, curnode, dist, parent = pop(queue)
93: (8)                     if curnode == target:
94: (12)                        path = [curnode]
95: (12)                        node = parent
96: (12)                        while node is not None:
97: (16)                            path.append(node)
98: (16)                            node = explored[node]
99: (12)                        path.reverse()
100: (12)                       return path
101: (8)                    if curnode in explored:
102: (12)                       if explored[curnode] is None:
103: (16)                           continue
104: (12)                       qcost, h = enqueued[curnode]
105: (12)                       if qcost < dist:
106: (16)                           continue
107: (8)                    explored[curnode] = parent
108: (8)                    for neighbor, w in G_succ[curnode].items():
109: (12)                       cost = weight(curnode, neighbor, w)
110: (12)                       if cost is None:
111: (16)                           continue
112: (12)                       ncost = dist + cost
113: (12)                       if neighbor in enqueued:
114: (16)                           qcost, h = enqueued[neighbor]
115: (16)                           if qcost <= ncost:
116: (20)                               continue
117: (12)                       else:
118: (16)                           h = heuristic(neighbor, target)
119: (12)                       if cutoff and ncost + h > cutoff:
120: (16)                           continue
121: (12)                       enqueued[neighbor] = ncost, h
122: (12)                       push(queue, (ncost + h, next(c), neighbor, ncost, curnode))
123: (4)                raise nx.NetworkXNoPath(f"Node {target} not reachable from {source}")
124: (0)            @nx._dispatchable(edge_attrs="weight", preserve_node_attrs="heuristic")
125: (0)            def astar_path_length(
126: (4)                G, source, target, heuristic=None, weight="weight", *, cutoff=None
127: (0)            ):
128: (4)                """Returns the length of the shortest path between source and target using
129: (4)                the A* ("A-star") algorithm.
130: (4)                Parameters
131: (4)                ----------
132: (4)                G : NetworkX graph
133: (4)                source : node
134: (7)                   Starting node for path
135: (4)                target : node
136: (7)                   Ending node for path
137: (4)                heuristic : function
138: (7)                   A function to evaluate the estimate of the distance
139: (7)                   from the a node to the target.  The function takes
140: (7)                   two nodes arguments and must return a number.
141: (7)                   If the heuristic is inadmissible (if it might
142: (7)                   overestimate the cost of reaching the goal from a node),
143: (7)                   the result may not be a shortest path.
144: (7)                   The algorithm does not support updating heuristic
145: (7)                   values for the same node due to caching the first
146: (7)                   heuristic calculation per node.
147: (4)                weight : string or function
148: (7)                   If this is a string, then edge weights will be accessed via the
149: (7)                   edge attribute with this key (that is, the weight of the edge
150: (7)                   joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
151: (7)                   such edge attribute exists, the weight of the edge is assumed to
152: (7)                   be one.
153: (7)                   If this is a function, the weight of an edge is the value
154: (7)                   returned by the function. The function must accept exactly three
155: (7)                   positional arguments: the two endpoints of an edge and the
156: (7)                   dictionary of edge attributes for that edge. The function must
157: (7)                   return a number or None to indicate a hidden edge.
158: (4)                cutoff : float, optional
159: (7)                   If this is provided, the search will be bounded to this value. I.e. if
160: (7)                   the evaluation function surpasses this value for a node n, the node
```

```
will not
161: (7)                   be expanded further and will be ignored. More formally, let h'(n) be
the
162: (7)                   heuristic function, and g(n) be the cost of reaching n from the source
node. Then,
163: (7)                   if g(n) + h'(n) > cutoff, the node will not be explored further.
164: (7)                   Note that if the heuristic is inadmissible, it is possible that paths
165: (7)                   are ignored even though they satisfy the cutoff.
166: (4)                Raises
167: (4)                ------
168: (4)                NetworkXNoPath
169: (8)                    If no path exists between source and target.
170: (4)                See Also
171: (4)                --------
172: (4)                astar_path
173: (4)                """
174: (4)                if source not in G or target not in G:
175: (8)                    msg = f"Either source {source} or target {target} is not in G"
176: (8)                    raise nx.NodeNotFound(msg)
177: (4)                weight = _weight_function(G, weight)
178: (4)                path = astar_path(G, source, target, heuristic, weight, cutoff=cutoff)
179: (4)                return sum(weight(u, v, G[u][v]) for u, v in zip(path[:-1], path[1:]))

----------------------------------------

File 294 - dense.py:

1: (0)              """Floyd-Warshall algorithm for shortest paths.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              __all__ = [
5: (4)                  "floyd_warshall",
6: (4)                  "floyd_warshall_predecessor_and_distance",
7: (4)                  "reconstruct_path",
8: (4)                  "floyd_warshall_numpy",
9: (0)              ]
10: (0)             @nx._dispatchable(edge_attrs="weight")
11: (0)             def floyd_warshall_numpy(G, nodelist=None, weight="weight"):
12: (4)                 """Find all-pairs shortest path lengths using Floyd's algorithm.
13: (4)                 This algorithm for finding shortest paths takes advantage of
14: (4)                 matrix representations of a graph and works well for dense
15: (4)                 graphs where all-pairs shortest path lengths are desired.
16: (4)                 The results are returned as a NumPy array, distance[i, j],
17: (4)                 where i and j are the indexes of two nodes in nodelist.
18: (4)                 The entry distance[i, j] is the distance along a shortest
19: (4)                 path from i to j. If no path exists the distance is Inf.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : NetworkX graph
23: (4)                 nodelist : list, optional (default=G.nodes)
24: (7)                    The rows and columns are ordered by the nodes in nodelist.
25: (7)                    If nodelist is None then the ordering is produced by G.nodes.
26: (7)                    Nodelist should include all nodes in G.
27: (4)                 weight: string, optional (default='weight')
28: (7)                    Edge data key corresponding to the edge weight.
29: (4)                 Returns
30: (4)                 -------
31: (4)                 distance : 2D numpy.ndarray
32: (8)                     A numpy array of shortest path distances between nodes.
33: (8)                     If there is no path between two nodes the value is Inf.
34: (4)                 Examples
35: (4)                 --------
36: (4)                 >>> G = nx.DiGraph()
37: (4)                 >>> G.add_weighted_edges_from([(0, 1, 5), (1, 2, 2), (2, 3, -3), (1, 3,
10), (3, 2, 8)])
38: (4)                 >>> nx.floyd_warshall_numpy(G)
39: (4)                 array([[ 0.,  5.,  7.,  4.],
40: (11)                       [inf,  0.,  2., -1.],
41: (11)                       [inf, inf,  0., -3.],
```

```
(11)                    [inf, inf,  8.,  0.]])
(4)                 Notes
(4)                 -----
(4)                 Floyd's algorithm is appropriate for finding shortest paths in
(4)                 dense graphs or graphs with negative weights when Dijkstra's
(4)                 algorithm fails. This algorithm can still fail if there are negative
(4)                 cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If nodelist is not a list of the nodes in G.
(4)                 """
(4)                 import numpy as np
(4)                 if nodelist is not None:
(8)                     if not (len(nodelist) == len(G) == len(set(nodelist))):
(12)                        raise nx.NetworkXError(
(16)                            "nodelist must contain every node in G with no repeats."
(16)                            "If you wanted a subgraph of G use G.subgraph(nodelist)"
(12)                        )
(4)                 A = nx.to_numpy_array(
(8)                     G, nodelist, multigraph_weight=min, weight=weight, nonedge=np.inf
(4)                 )
(4)                 n, m = A.shape
(4)                 np.fill_diagonal(A, 0)  # diagonal elements should be zero
(4)                 for i in range(n):
(8)                     A = np.minimum(A, A[i, :][np.newaxis, :] + A[:, i][:, np.newaxis])
(4)                 return A
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def floyd_warshall_predecessor_and_distance(G, weight="weight"):
(4)                 """Find all-pairs shortest path lengths using Floyd's algorithm.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 weight: string, optional (default= 'weight')
(7)                    Edge data key corresponding to the edge weight.
(4)                 Returns
(4)                 -------
(4)                 predecessor,distance : dictionaries
(7)                    Dictionaries, keyed by source and target, of predecessors and distances
(7)                    in the shortest path.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_weighted_edges_from(
(4)                 ...     [
(4)                 ...         ("s", "u", 10),
(4)                 ...         ("s", "x", 5),
(4)                 ...         ("u", "v", 1),
(4)                 ...         ("u", "x", 2),
(4)                 ...         ("v", "y", 1),
(4)                 ...         ("x", "u", 3),
(4)                 ...         ("x", "v", 5),
(4)                 ...         ("x", "y", 2),
(4)                 ...         ("y", "s", 7),
(4)                 ...         ("y", "v", 6),
(4)                 ...     ]
(4)                 ... )
(4)                 >>> predecessors, _ = nx.floyd_warshall_predecessor_and_distance(G)
(4)                 >>> print(nx.reconstruct_path("s", "v", predecessors))
(4)                 ['s', 'x', 'u', 'v']
(4)                 Notes
(4)                 -----
(4)                 Floyd's algorithm is appropriate for finding shortest paths
(4)                 in dense graphs or graphs with negative weights when Dijkstra's algorithm
(4)                 fails.  This algorithm can still fail if there are negative cycles.
(4)                 It has running time $O(n^3)$ with running space of $O(n^2)$.
(4)                 See Also
(4)                 --------
(4)                 floyd_warshall
```

```
(4)                 floyd_warshall_numpy
(4)                 all_pairs_shortest_path
(4)                 all_pairs_shortest_path_length
(4)                 """
(4)                 from collections import defaultdict
(4)                 dist = defaultdict(lambda: defaultdict(lambda: float("inf")))
(4)                 for u in G:
(8)                     dist[u][u] = 0
(4)                 pred = defaultdict(dict)
(4)                 undirected = not G.is_directed()
(4)                 for u, v, d in G.edges(data=True):
(8)                     e_weight = d.get(weight, 1.0)
(8)                     dist[u][v] = min(e_weight, dist[u][v])
(8)                     pred[u][v] = u
(8)                     if undirected:
(12)                        dist[v][u] = min(e_weight, dist[v][u])
(12)                        pred[v][u] = v
(4)                 for w in G:
(8)                     dist_w = dist[w]  # save recomputation
(8)                     for u in G:
(12)                        dist_u = dist[u]  # save recomputation
(12)                        for v in G:
(16)                            d = dist_u[w] + dist_w[v]
(16)                            if dist_u[v] > d:
(20)                                dist_u[v] = d
(20)                                pred[u][v] = pred[w][v]
(4)                 return dict(pred), dict(dist)
(0)             @nx._dispatchable(graphs=None)
(0)             def reconstruct_path(source, target, predecessors):
(4)                 """Reconstruct a path from source to target using the predecessors
(4)                 dict as returned by floyd_warshall_predecessor_and_distance
(4)                 Parameters
(4)                 ----------
(4)                 source : node
(7)                    Starting node for path
(4)                 target : node
(7)                    Ending node for path
(4)                 predecessors: dictionary
(7)                    Dictionary, keyed by source and target, of predecessors in the
(7)                    shortest path, as returned by floyd_warshall_predecessor_and_distance
(4)                 Returns
(4)                 -------
(4)                 path : list
(7)                    A list of nodes containing the shortest path from source to target
(7)                    If source and target are the same, an empty list is returned
(4)                 Notes
(4)                 -----
(4)                 This function is meant to give more applicability to the
(4)                 floyd_warshall_predecessor_and_distance function
(4)                 See Also
(4)                 --------
(4)                 floyd_warshall_predecessor_and_distance
(4)                 """
(4)                 if source == target:
(8)                     return []
(4)                 prev = predecessors[source]
(4)                 curr = prev[target]
(4)                 path = [target, curr]
(4)                 while curr != source:
(8)                     curr = prev[curr]
(8)                     path.append(curr)
(4)                 return list(reversed(path))
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def floyd_warshall(G, weight="weight"):
(4)                 """Find all-pairs shortest path lengths using Floyd's algorithm.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 weight: string, optional (default= 'weight')
```

```
180: (7)                   Edge data key corresponding to the edge weight.
181: (4)                Returns
182: (4)                -------
183: (4)                distance : dict
184: (7)                   A dictionary,  keyed by source and target, of shortest paths distances
185: (7)                   between nodes.
186: (4)                Examples
187: (4)                --------
188: (4)                >>> G = nx.DiGraph()
189: (4)                >>> G.add_weighted_edges_from([(0, 1, 5), (1, 2, 2), (2, 3, -3), (1, 3,
10), (3, 2, 8)])
190: (4)                >>> fw = nx.floyd_warshall(G, weight="weight")
191: (4)                >>> results = {a: dict(b) for a, b in fw.items()}
192: (4)                >>> print(results)
193: (4)                {0: {0: 0, 1: 5, 2: 7, 3: 4}, 1: {1: 0, 2: 2, 3: -1, 0: inf}, 2: {2: 0, 3:
-3, 0: inf, 1: inf}, 3: {3: 0, 2: 8, 0: inf, 1: inf}}
194: (4)                Notes
195: (4)                -----
196: (4)                Floyd's algorithm is appropriate for finding shortest paths
197: (4)                in dense graphs or graphs with negative weights when Dijkstra's algorithm
198: (4)                fails.  This algorithm can still fail if there are negative cycles.
199: (4)                It has running time $O(n^3)$ with running space of $O(n^2)$.
200: (4)                See Also
201: (4)                --------
202: (4)                floyd_warshall_predecessor_and_distance
203: (4)                floyd_warshall_numpy
204: (4)                all_pairs_shortest_path
205: (4)                all_pairs_shortest_path_length
206: (4)                """
207: (4)                return floyd_warshall_predecessor_and_distance(G, weight=weight)[1]
```

File 295 - generic.py:

```
1: (0)              """
2: (0)              Compute the shortest paths and path lengths between nodes in the graph.
3: (0)              These algorithms work with undirected and directed graphs.
4: (0)              """
5: (0)              import warnings
6: (0)              import networkx as nx
7: (0)              __all__ = [
8: (4)                  "shortest_path",
9: (4)                  "all_shortest_paths",
10: (4)                 "single_source_all_shortest_paths",
11: (4)                 "all_pairs_all_shortest_paths",
12: (4)                 "shortest_path_length",
13: (4)                 "average_shortest_path_length",
14: (4)                 "has_path",
15: (0)             ]
16: (0)             @nx._dispatchable
17: (0)             def has_path(G, source, target):
18: (4)                 """Returns *True* if *G* has a path from *source* to *target*.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : NetworkX graph
22: (4)                 source : node
23: (7)                    Starting node for path
24: (4)                 target : node
25: (7)                    Ending node for path
26: (4)                 """
27: (4)                 try:
28: (8)                     nx.shortest_path(G, source, target)
29: (4)                 except nx.NetworkXNoPath:
30: (8)                     return False
31: (4)                 return True
32: (0)             @nx._dispatchable(edge_attrs="weight")
33: (0)             def shortest_path(G, source=None, target=None, weight=None,
method="dijkstra"):
```

```
    """Compute shortest paths in the graph.
    Parameters
    ----------
    G : NetworkX graph
    source : node, optional
        Starting node for path. If not specified, compute shortest
        paths for each possible starting node.
    target : node, optional
        Ending node for path. If not specified, compute shortest
        paths to all possible nodes.
    weight : None, string or function, optional (default = None)
        If None, every edge has weight/distance/cost 1.
        If a string, use this edge attribute as the edge weight.
        Any edge attribute not present defaults to 1.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly
        three positional arguments: the two endpoints of an edge and
        the dictionary of edge attributes for that edge.
        The function must return a number.
    method : string, optional (default = 'dijkstra')
        The algorithm to use to compute the path.
        Supported options: 'dijkstra', 'bellman-ford'.
        Other inputs produce a ValueError.
        If `weight` is None, unweighted graph methods are used, and this
        suggestion is ignored.
    Returns
    -------
    path: list or dictionary
        All returned paths include both the source and target in the path.
        If the source and target are both specified, return a single list
        of nodes in a shortest path from the source to the target.
        If only the source is specified, return a dictionary keyed by
        targets with a list of nodes in a shortest path from the source
        to one of the targets.
        If only the target is specified, return a dictionary keyed by
        sources with a list of nodes in a shortest path from one of the
        sources to the target.
        If neither the source nor target are specified return a dictionary
        of dictionaries with path[source][target]=[list of nodes in path].
    Raises
    ------
    NodeNotFound
        If `source` is not in `G`.
    ValueError
        If `method` is not among the supported options.
    Examples
    --------
    >>> G = nx.path_graph(5)
    >>> print(nx.shortest_path(G, source=0, target=4))
    [0, 1, 2, 3, 4]
    >>> p = nx.shortest_path(G, source=0)  # target not specified
    >>> p[3]  # shortest path from source=0 to target=3
    [0, 1, 2, 3]
    >>> p = nx.shortest_path(G, target=4)  # source not specified
    >>> p[1]  # shortest path from source=1 to target=4
    [1, 2, 3, 4]
    >>> p = dict(nx.shortest_path(G))  # source, target not specified
    >>> p[2][4]  # shortest path from source=2 to target=4
    [2, 3, 4]
    Notes
    -----
    There may be more than one shortest path between a source and target.
    This returns only one of them.
    See Also
    --------
    all_pairs_shortest_path
    all_pairs_dijkstra_path
    all_pairs_bellman_ford_path
    single_source_shortest_path
```

```
103: (4)                single_source_dijkstra_path
104: (4)                single_source_bellman_ford_path
105: (4)                """
106: (4)                if method not in ("dijkstra", "bellman-ford"):
107: (8)                    raise ValueError(f"method not supported: {method}")
108: (4)                method = "unweighted" if weight is None else method
109: (4)                if source is None:
110: (8)                    if target is None:
111: (12)                       warnings.warn(
112: (16)                           (
113: (20)                               "\n\nshortest_path will return an iterator that yields\n"
114: (20)                               "(node, path) pairs instead of a dictionary when source\n"
115: (20)                               "and target are unspecified beginning in version 3.5\n\n"
116: (20)                               "To keep the current behavior, use:\n\n"
117: (20)                               "\tdict(nx.shortest_path(G))"
118: (16)                           ),
119: (16)                           FutureWarning,
120: (16)                           stacklevel=3,
121: (12)                       )
122: (12)                       if method == "unweighted":
123: (16)                           paths = dict(nx.all_pairs_shortest_path(G))
124: (12)                       elif method == "dijkstra":
125: (16)                           paths = dict(nx.all_pairs_dijkstra_path(G, weight=weight))
126: (12)                       else:  # method == 'bellman-ford':
127: (16)                           paths = dict(nx.all_pairs_bellman_ford_path(G, weight=weight))
128: (8)                    else:
129: (12)                       if G.is_directed():
130: (16)                           G = G.reverse(copy=False)
131: (12)                       if method == "unweighted":
132: (16)                           paths = nx.single_source_shortest_path(G, target)
133: (12)                       elif method == "dijkstra":
134: (16)                           paths = nx.single_source_dijkstra_path(G, target,
weight=weight)
135: (12)                       else:  # method == 'bellman-ford':
136: (16)                           paths = nx.single_source_bellman_ford_path(G, target,
weight=weight)
137: (12)                       for target in paths:
138: (16)                           paths[target] = list(reversed(paths[target]))
139: (4)                else:
140: (8)                    if target is None:
141: (12)                       if method == "unweighted":
142: (16)                           paths = nx.single_source_shortest_path(G, source)
143: (12)                       elif method == "dijkstra":
144: (16)                           paths = nx.single_source_dijkstra_path(G, source,
weight=weight)
145: (12)                       else:  # method == 'bellman-ford':
146: (16)                           paths = nx.single_source_bellman_ford_path(G, source,
weight=weight)
147: (8)                    else:
148: (12)                       if method == "unweighted":
149: (16)                           paths = nx.bidirectional_shortest_path(G, source, target)
150: (12)                       elif method == "dijkstra":
151: (16)                           _, paths = nx.bidirectional_dijkstra(G, source, target,
weight)
152: (12)                       else:  # method == 'bellman-ford':
153: (16)                           paths = nx.bellman_ford_path(G, source, target, weight)
154: (4)                return paths
155: (0)            @nx._dispatchable(edge_attrs="weight")
156: (0)            def shortest_path_length(G, source=None, target=None, weight=None,
method="dijkstra"):
157: (4)                """Compute shortest path lengths in the graph.
158: (4)                Parameters
159: (4)                ----------
160: (4)                G : NetworkX graph
161: (4)                source : node, optional
162: (8)                    Starting node for path.
163: (8)                    If not specified, compute shortest path lengths using all nodes as
164: (8)                    source nodes.
165: (4)                target : node, optional
```

```
(8)                 Ending node for path.
(8)                 If not specified, compute shortest path lengths using all nodes as
(8)                 target nodes.
(4)             weight : None, string or function, optional (default = None)
(8)                 If None, every edge has weight/distance/cost 1.
(8)                 If a string, use this edge attribute as the edge weight.
(8)                 Any edge attribute not present defaults to 1.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly
(8)                 three positional arguments: the two endpoints of an edge and
(8)                 the dictionary of edge attributes for that edge.
(8)                 The function must return a number.
(4)             method : string, optional (default = 'dijkstra')
(8)                 The algorithm to use to compute the path length.
(8)                 Supported options: 'dijkstra', 'bellman-ford'.
(8)                 Other inputs produce a ValueError.
(8)                 If `weight` is None, unweighted graph methods are used, and this
(8)                 suggestion is ignored.
(4)             Returns
(4)             -------
(4)             length: int or iterator
(8)                 If the source and target are both specified, return the length of
(8)                 the shortest path from the source to the target.
(8)                 If only the source is specified, return a dict keyed by target
(8)                 to the shortest path length from the source to that target.
(8)                 If only the target is specified, return a dict keyed by source
(8)                 to the shortest path length from that source to the target.
(8)                 If neither the source nor target are specified, return an iterator
(8)                 over (source, dictionary) where dictionary is keyed by target to
(8)                 shortest path length from source to that target.
(4)             Raises
(4)             ------
(4)             NodeNotFound
(8)                 If `source` is not in `G`.
(4)             NetworkXNoPath
(8)                 If no path exists between source and target.
(4)             ValueError
(8)                 If `method` is not among the supported options.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> nx.shortest_path_length(G, source=0, target=4)
(4)             4
(4)             >>> p = nx.shortest_path_length(G, source=0)  # target not specified
(4)             >>> p[4]
(4)             4
(4)             >>> p = nx.shortest_path_length(G, target=4)  # source not specified
(4)             >>> p[0]
(4)             4
(4)             >>> p = dict(nx.shortest_path_length(G))  # source,target not specified
(4)             >>> p[0][4]
(4)             4
(4)             Notes
(4)             -----
(4)             The length of the path is always 1 less than the number of nodes involved
(4)             in the path since the length measures the number of edges followed.
(4)             For digraphs this returns the shortest directed path length. To find path
(4)             lengths in the reverse direction use G.reverse(copy=False) first to flip
(4)             the edge orientation.
(4)             See Also
(4)             --------
(4)             all_pairs_shortest_path_length
(4)             all_pairs_dijkstra_path_length
(4)             all_pairs_bellman_ford_path_length
(4)             single_source_shortest_path_length
(4)             single_source_dijkstra_path_length
(4)             single_source_bellman_ford_path_length
(4)             """
(4)             if method not in ("dijkstra", "bellman-ford"):
```

```
(8)                     raise ValueError(f"method not supported: {method}")
(4)                 method = "unweighted" if weight is None else method
(4)                 if source is None:
(8)                     if target is None:
(12)                        if method == "unweighted":
(16)                            paths = nx.all_pairs_shortest_path_length(G)
(12)                        elif method == "dijkstra":
(16)                            paths = nx.all_pairs_dijkstra_path_length(G, weight=weight)
(12)                        else:  # method == 'bellman-ford':
(16)                            paths = nx.all_pairs_bellman_ford_path_length(G, weight=weight)
(8)                     else:
(12)                        if G.is_directed():
(16)                            G = G.reverse(copy=False)
(12)                        if method == "unweighted":
(16)                            path_length = nx.single_source_shortest_path_length
(16)                            paths = path_length(G, target)
(12)                        elif method == "dijkstra":
(16)                            path_length = nx.single_source_dijkstra_path_length
(16)                            paths = path_length(G, target, weight=weight)
(12)                        else:  # method == 'bellman-ford':
(16)                            path_length = nx.single_source_bellman_ford_path_length
(16)                            paths = path_length(G, target, weight=weight)
(4)                 else:
(8)                     if target is None:
(12)                        if method == "unweighted":
(16)                            paths = nx.single_source_shortest_path_length(G, source)
(12)                        elif method == "dijkstra":
(16)                            path_length = nx.single_source_dijkstra_path_length
(16)                            paths = path_length(G, source, weight=weight)
(12)                        else:  # method == 'bellman-ford':
(16)                            path_length = nx.single_source_bellman_ford_path_length
(16)                            paths = path_length(G, source, weight=weight)
(8)                     else:
(12)                        if method == "unweighted":
(16)                            p = nx.bidirectional_shortest_path(G, source, target)
(16)                            paths = len(p) - 1
(12)                        elif method == "dijkstra":
(16)                            paths = nx.dijkstra_path_length(G, source, target, weight)
(12)                        else:  # method == 'bellman-ford':
(16)                            paths = nx.bellman_ford_path_length(G, source, target, weight)
(4)                 return paths
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def average_shortest_path_length(G, weight=None, method=None):
(4)                 r"""Returns the average shortest path length.
(4)                 The average shortest path length is
(4)                 .. math::
(7)                    a =\sum_{\substack{s,t \in V \\ s\neq t}} \frac{d(s, t)}{n(n-1)}
(4)                 where `V` is the set of nodes in `G`,
(4)                 `d(s, t)` is the shortest path from `s` to `t`,
(4)                 and `n` is the number of nodes in `G`.
(4)                 .. versionchanged:: 3.0
(7)                    An exception is raised for directed graphs that are not strongly
(7)                    connected.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 weight : None, string or function, optional (default = None)
(8)                     If None, every edge has weight/distance/cost 1.
(8)                     If a string, use this edge attribute as the edge weight.
(8)                     Any edge attribute not present defaults to 1.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly
(8)                     three positional arguments: the two endpoints of an edge and
(8)                     the dictionary of edge attributes for that edge.
(8)                     The function must return a number.
(4)                 method : string, optional (default = 'unweighted' or 'dijkstra')
(8)                     The algorithm to use to compute the path lengths.
(8)                     Supported options are 'unweighted', 'dijkstra', 'bellman-ford',
```

```
(8)                     'floyd-warshall' and 'floyd-warshall-numpy'.
(8)                     Other method values produce a ValueError.
(8)                     The default method is 'unweighted' if `weight` is None,
(8)                     otherwise the default method is 'dijkstra'.
(4)                 Raises
(4)                 ------
(4)                 NetworkXPointlessConcept
(8)                     If `G` is the null graph (that is, the graph on zero nodes).
(4)                 NetworkXError
(8)                     If `G` is not connected (or not strongly connected, in the case
(8)                     of a directed graph).
(4)                 ValueError
(8)                     If `method` is not among the supported options.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> nx.average_shortest_path_length(G)
(4)                 2.0
(4)                 For disconnected graphs, you can compute the average shortest path
(4)                 length for each component
(4)                 >>> G = nx.Graph([(1, 2), (3, 4)])
(4)                 >>> for C in (G.subgraph(c).copy() for c in nx.connected_components(G)):
(4)                 ...     print(nx.average_shortest_path_length(C))
(4)                 1.0
(4)                 1.0
(4)                 """
(4)                 single_source_methods = ["unweighted", "dijkstra", "bellman-ford"]
(4)                 all_pairs_methods = ["floyd-warshall", "floyd-warshall-numpy"]
(4)                 supported_methods = single_source_methods + all_pairs_methods
(4)                 if method is None:
(8)                     method = "unweighted" if weight is None else "dijkstra"
(4)                 if method not in supported_methods:
(8)                     raise ValueError(f"method not supported: {method}")
(4)                 n = len(G)
(4)                 if n == 0:
(8)                     msg = (
(12)                        "the null graph has no paths, thus there is no average "
(12)                        "shortest path length"
(8)                     )
(8)                     raise nx.NetworkXPointlessConcept(msg)
(4)                 if n == 1:
(8)                     return 0
(4)                 if G.is_directed() and not nx.is_strongly_connected(G):
(8)                     raise nx.NetworkXError("Graph is not strongly connected.")
(4)                 if not G.is_directed() and not nx.is_connected(G):
(8)                     raise nx.NetworkXError("Graph is not connected.")
(4)                 def path_length(v):
(8)                     if method == "unweighted":
(12)                        return nx.single_source_shortest_path_length(G, v)
(8)                     elif method == "dijkstra":
(12)                        return nx.single_source_dijkstra_path_length(G, v, weight=weight)
(8)                     elif method == "bellman-ford":
(12)                        return nx.single_source_bellman_ford_path_length(G, v,
weight=weight)
(4)                 if method in single_source_methods:
(8)                     s = sum(l for u in G for l in path_length(u).values())
(4)                 else:
(8)                     if method == "floyd-warshall":
(12)                        all_pairs = nx.floyd_warshall(G, weight=weight)
(12)                        s = sum(sum(t.values()) for t in all_pairs.values())
(8)                     elif method == "floyd-warshall-numpy":
(12)                        s = float(nx.floyd_warshall_numpy(G, weight=weight).sum())
(4)                 return s / (n * (n - 1))
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def all_shortest_paths(G, source, target, weight=None, method="dijkstra"):
(4)                 """Compute all shortest simple paths in the graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
```

```
(4)                 source : node
(7)                    Starting node for path.
(4)                 target : node
(7)                    Ending node for path.
(4)                 weight : None, string or function, optional (default = None)
(8)                     If None, every edge has weight/distance/cost 1.
(8)                     If a string, use this edge attribute as the edge weight.
(8)                     Any edge attribute not present defaults to 1.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly
(8)                     three positional arguments: the two endpoints of an edge and
(8)                     the dictionary of edge attributes for that edge.
(8)                     The function must return a number.
(4)                 method : string, optional (default = 'dijkstra')
(7)                    The algorithm to use to compute the path lengths.
(7)                    Supported options: 'dijkstra', 'bellman-ford'.
(7)                    Other inputs produce a ValueError.
(7)                    If `weight` is None, unweighted graph methods are used, and this
(7)                    suggestion is ignored.
(4)                 Returns
(4)                 -------
(4)                 paths : generator of lists
(8)                     A generator of all paths between source and target.
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If `method` is not among the supported options.
(4)                 NetworkXNoPath
(8)                     If `target` cannot be reached from `source`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> nx.add_path(G, [0, 1, 2])
(4)                 >>> nx.add_path(G, [0, 10, 2])
(4)                 >>> print([p for p in nx.all_shortest_paths(G, source=0, target=2)])
(4)                 [[0, 1, 2], [0, 10, 2]]
(4)                 Notes
(4)                 -----
(4)                 There may be many shortest paths between the source and target.  If G
(4)                 contains zero-weight cycles, this function will not produce all shortest
(4)                 paths because doing so would produce infinitely many paths of unbounded
(4)                 length -- instead, we only produce the shortest simple paths.
(4)                 See Also
(4)                 --------
(4)                 shortest_path
(4)                 single_source_shortest_path
(4)                 all_pairs_shortest_path
(4)                 """
(4)                 method = "unweighted" if weight is None else method
(4)                 if method == "unweighted":
(8)                     pred = nx.predecessor(G, source)
(4)                 elif method == "dijkstra":
(8)                     pred, dist = nx.dijkstra_predecessor_and_distance(G, source,
weight=weight)
(4)                 elif method == "bellman-ford":
(8)                     pred, dist = nx.bellman_ford_predecessor_and_distance(G, source,
weight=weight)
(4)                 else:
(8)                     raise ValueError(f"method not supported: {method}")
(4)                 return _build_paths_from_predecessors({source}, target, pred)
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def single_source_all_shortest_paths(G, source, weight=None,
method="dijkstra"):
(4)                 """Compute all shortest simple paths from the given source in the graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node
(7)                    Starting node for path.
```

```
    weight : None, string or function, optional (default = None)
        If None, every edge has weight/distance/cost 1.
        If a string, use this edge attribute as the edge weight.
        Any edge attribute not present defaults to 1.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly
        three positional arguments: the two endpoints of an edge and
        the dictionary of edge attributes for that edge.
        The function must return a number.
    method : string, optional (default = 'dijkstra')
       The algorithm to use to compute the path lengths.
       Supported options: 'dijkstra', 'bellman-ford'.
       Other inputs produce a ValueError.
       If `weight` is None, unweighted graph methods are used, and this
       suggestion is ignored.
    Returns
    -------
    paths : generator of dictionary
        A generator of all paths between source and all nodes in the graph.
    Raises
    ------
    ValueError
        If `method` is not among the supported options.
    Examples
    --------
    >>> G = nx.Graph()
    >>> nx.add_path(G, [0, 1, 2, 3, 0])
    >>> dict(nx.single_source_all_shortest_paths(G, source=0))
    {0: [[0]], 1: [[0, 1]], 2: [[0, 1, 2], [0, 3, 2]], 3: [[0, 3]]}
    Notes
    -----
    There may be many shortest paths between the source and target.  If G
    contains zero-weight cycles, this function will not produce all shortest
    paths because doing so would produce infinitely many paths of unbounded
    length -- instead, we only produce the shortest simple paths.
    See Also
    --------
    shortest_path
    all_shortest_paths
    single_source_shortest_path
    all_pairs_shortest_path
    all_pairs_all_shortest_paths
    """
    method = "unweighted" if weight is None else method
    if method == "unweighted":
        pred = nx.predecessor(G, source)
    elif method == "dijkstra":
        pred, dist = nx.dijkstra_predecessor_and_distance(G, source, weight=weight)
    elif method == "bellman-ford":
        pred, dist = nx.bellman_ford_predecessor_and_distance(G, source, weight=weight)
    else:
        raise ValueError(f"method not supported: {method}")
    for n in G:
        try:
            yield n, list(_build_paths_from_predecessors({source}, n, pred))
        except nx.NetworkXNoPath:
            pass
@nx._dispatchable(edge_attrs="weight")
def all_pairs_all_shortest_paths(G, weight=None, method="dijkstra"):
    """Compute all shortest paths between all nodes.
    Parameters
    ----------
    G : NetworkX graph
    weight : None, string or function, optional (default = None)
        If None, every edge has weight/distance/cost 1.
        If a string, use this edge attribute as the edge weight.
        Any edge attribute not present defaults to 1.
```

```
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly
(8)                     three positional arguments: the two endpoints of an edge and
(8)                     the dictionary of edge attributes for that edge.
(8)                     The function must return a number.
(4)                 method : string, optional (default = 'dijkstra')
(7)                    The algorithm to use to compute the path lengths.
(7)                    Supported options: 'dijkstra', 'bellman-ford'.
(7)                    Other inputs produce a ValueError.
(7)                    If `weight` is None, unweighted graph methods are used, and this
(7)                    suggestion is ignored.
(4)                 Returns
(4)                 -------
(4)                 paths : generator of dictionary
(8)                     Dictionary of arrays, keyed by source and target, of all shortest
paths.
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If `method` is not among the supported options.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.cycle_graph(4)
(4)                 >>> dict(nx.all_pairs_all_shortest_paths(G))[0][2]
(4)                 [[0, 1, 2], [0, 3, 2]]
(4)                 >>> dict(nx.all_pairs_all_shortest_paths(G))[0][3]
(4)                 [[0, 3]]
(4)                 Notes
(4)                 -----
(4)                 There may be multiple shortest paths with equal lengths. Unlike
(4)                 all_pairs_shortest_path, this method returns all shortest paths.
(4)                 See Also
(4)                 --------
(4)                 all_pairs_shortest_path
(4)                 single_source_all_shortest_paths
(4)                 """
(4)                 for n in G:
(8)                     yield (
(12)                        n,
(12)                        dict(single_source_all_shortest_paths(G, n, weight=weight,
method=method)),
(8)                     )
(0)             def _build_paths_from_predecessors(sources, target, pred):
(4)                 """Compute all simple paths to target, given the predecessors found in
(4)                 pred, terminating when any source in sources is found.
(4)                 Parameters
(4)                 ----------
(4)                 sources : set
(7)                    Starting nodes for path.
(4)                 target : node
(7)                    Ending node for path.
(4)                 pred : dict
(7)                    A dictionary of predecessor lists, keyed by node
(4)                 Returns
(4)                 -------
(4)                 paths : generator of lists
(8)                     A generator of all paths between source and target.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNoPath
(8)                     If `target` cannot be reached from `source`.
(4)                 Notes
(4)                 -----
(4)                 There may be many paths between the sources and target.  If there are
(4)                 cycles among the predecessors, this function will not produce all
(4)                 possible paths because doing so would produce infinitely many paths
(4)                 of unbounded length -- instead, we only produce simple paths.
(4)                 See Also
(4)                 --------
```

```
571: (4)                shortest_path
572: (4)                single_source_shortest_path
573: (4)                all_pairs_shortest_path
574: (4)                all_shortest_paths
575: (4)                bellman_ford_path
576: (4)                """
577: (4)                if target not in pred:
578: (8)                    raise nx.NetworkXNoPath(f"Target {target} cannot be reached from given
sources")
579: (4)                seen = {target}
580: (4)                stack = [[target, 0]]
581: (4)                top = 0
582: (4)                while top >= 0:
583: (8)                    node, i = stack[top]
584: (8)                    if node in sources:
585: (12)                       yield [p for p, n in reversed(stack[: top + 1])]
586: (8)                    if len(pred[node]) > i:
587: (12)                       stack[top][1] = i + 1
588: (12)                       next = pred[node][i]
589: (12)                       if next in seen:
590: (16)                           continue
591: (12)                       else:
592: (16)                           seen.add(next)
593: (12)                       top += 1
594: (12)                       if top == len(stack):
595: (16)                           stack.append([next, 0])
596: (12)                       else:
597: (16)                           stack[top][:] = [next, 0]
598: (8)                    else:
599: (12)                       seen.discard(node)
600: (12)                       top -= 1
```

File 296 - unweighted.py:

```
1: (0)              """
2: (0)              Shortest path algorithms for unweighted graphs.
3: (0)              """
4: (0)              import warnings
5: (0)              import networkx as nx
6: (0)              __all__ = [
7: (4)                  "bidirectional_shortest_path",
8: (4)                  "single_source_shortest_path",
9: (4)                  "single_source_shortest_path_length",
10: (4)                 "single_target_shortest_path",
11: (4)                 "single_target_shortest_path_length",
12: (4)                 "all_pairs_shortest_path",
13: (4)                 "all_pairs_shortest_path_length",
14: (4)                 "predecessor",
15: (0)             ]
16: (0)             @nx._dispatchable
17: (0)             def single_source_shortest_path_length(G, source, cutoff=None):
18: (4)                 """Compute the shortest path lengths from source to all reachable nodes.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : NetworkX graph
22: (4)                 source : node
23: (7)                    Starting node for path
24: (4)                 cutoff : integer, optional
25: (8)                     Depth to stop the search. Only paths of length <= cutoff are returned.
26: (4)                 Returns
27: (4)                 -------
28: (4)                 lengths : dict
29: (8)                     Dict keyed by node to shortest path length to source.
30: (4)                 Examples
31: (4)                 --------
32: (4)                 >>> G = nx.path_graph(5)
33: (4)                 >>> length = nx.single_source_shortest_path_length(G, 0)
```

```
(4)                 >>> length[4]
(4)                 4
(4)                 >>> for node in length:
(4)                 ...     print(f"{node}: {length[node]}")
(4)                 0: 0
(4)                 1: 1
(4)                 2: 2
(4)                 3: 3
(4)                 4: 4
(4)                 See Also
(4)                 --------
(4)                 shortest_path_length
(4)                 """
(4)                 if source not in G:
(8)                     raise nx.NodeNotFound(f"Source {source} is not in G")
(4)                 if cutoff is None:
(8)                     cutoff = float("inf")
(4)                 nextlevel = [source]
(4)                 return dict(_single_shortest_path_length(G._adj, nextlevel, cutoff))
(0)             def _single_shortest_path_length(adj, firstlevel, cutoff):
(4)                 """Yields (node, level) in a breadth first search
(4)                 Shortest Path Length helper function
(4)                 Parameters
(4)                 ----------
(8)                     adj : dict
(12)                        Adjacency dict or view
(8)                     firstlevel : list
(12)                        starting nodes, e.g. [source] or [target]
(8)                     cutoff : int or float
(12)                        level at which we stop the process
(4)                 """
(4)                 seen = set(firstlevel)
(4)                 nextlevel = firstlevel
(4)                 level = 0
(4)                 n = len(adj)
(4)                 for v in nextlevel:
(8)                     yield (v, level)
(4)                 while nextlevel and cutoff > level:
(8)                     level += 1
(8)                     thislevel = nextlevel
(8)                     nextlevel = []
(8)                     for v in thislevel:
(12)                        for w in adj[v]:
(16)                            if w not in seen:
(20)                                seen.add(w)
(20)                                nextlevel.append(w)
(20)                                yield (w, level)
(12)                        if len(seen) == n:
(16)                            return
(0)             @nx._dispatchable
(0)             def single_target_shortest_path_length(G, target, cutoff=None):
(4)                 """Compute the shortest path lengths to target from all reachable nodes.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 target : node
(7)                    Target node for path
(4)                 cutoff : integer, optional
(8)                     Depth to stop the search. Only paths of length <= cutoff are returned.
(4)                 Returns
(4)                 -------
(4)                 lengths : iterator
(8)                     (source, shortest path length) iterator
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5, create_using=nx.DiGraph())
(4)                 >>> length = dict(nx.single_target_shortest_path_length(G, 4))
(4)                 >>> length[0]
(4)                 4
```

```
(4)                 >>> for node in range(5):
(4)                 ...     print(f"{node}: {length[node]}")
(4)                 0: 4
(4)                 1: 3
(4)                 2: 2
(4)                 3: 1
(4)                 4: 0
(4)                 See Also
(4)                 --------
(4)                 single_source_shortest_path_length, shortest_path_length
(4)                 """
(4)                 if target not in G:
(8)                     raise nx.NodeNotFound(f"Target {target} is not in G")
(4)                 warnings.warn(
(8)                     (
(12)                        "\n\nsingle_target_shortest_path_length will return a dict instead of"
(12)                        "\nan iterator in version 3.5"
(8)                     ),
(8)                     FutureWarning,
(8)                     stacklevel=3,
(4)                 )
(4)                 if cutoff is None:
(8)                     cutoff = float("inf")
(4)                 adj = G._pred if G.is_directed() else G._adj
(4)                 nextlevel = [target]
(4)                 return _single_shortest_path_length(adj, nextlevel, cutoff)
(0)             @nx._dispatchable
(0)             def all_pairs_shortest_path_length(G, cutoff=None):
(4)                 """Computes the shortest path lengths between all nodes in `G`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 cutoff : integer, optional
(8)                     Depth at which to stop the search. Only paths of length at most
(8)                     `cutoff` are returned.
(4)                 Returns
(4)                 -------
(4)                 lengths : iterator
(8)                     (source, dictionary) iterator with dictionary keyed by target and
(8)                     shortest path length as the key value.
(4)                 Notes
(4)                 -----
(4)                 The iterator returned only has reachable node pairs.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> length = dict(nx.all_pairs_shortest_path_length(G))
(4)                 >>> for node in [0, 1, 2, 3, 4]:
(4)                 ...     print(f"1 - {node}: {length[1][node]}")
(4)                 1 - 0: 1
(4)                 1 - 1: 0
(4)                 1 - 2: 1
(4)                 1 - 3: 2
(4)                 1 - 4: 3
(4)                 >>> length[3][2]
(4)                 1
(4)                 >>> length[2][2]
(4)                 0
(4)                 """
(4)                 length = single_source_shortest_path_length
(4)                 for n in G:
(8)                     yield (n, length(G, n, cutoff=cutoff))
(0)             @nx._dispatchable
(0)             def bidirectional_shortest_path(G, source, target):
(4)                 """Returns a list of nodes in a shortest path between source and target.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
```

```
(4)                 source : node label
(7)                    starting node for path
(4)                 target : node label
(7)                    ending node for path
(4)                 Returns
(4)                 -------
(4)                 path: list
(7)                    List of nodes in a path from source to target.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNoPath
(7)                    If no path exists between source and target.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> nx.add_path(G, [0, 1, 2, 3, 0, 4, 5, 6, 7, 4])
(4)                 >>> nx.bidirectional_shortest_path(G, 2, 6)
(4)                 [2, 1, 0, 4, 5, 6]
(4)                 See Also
(4)                 --------
(4)                 shortest_path
(4)                 Notes
(4)                 -----
(4)                 This algorithm is used by shortest_path(G, source, target).
(4)                 """
(4)                 if source not in G or target not in G:
(8)                     msg = f"Either source {source} or target {target} is not in G"
(8)                     raise nx.NodeNotFound(msg)
(4)                 results = _bidirectional_pred_succ(G, source, target)
(4)                 pred, succ, w = results
(4)                 path = []
(4)                 while w is not None:
(8)                     path.append(w)
(8)                     w = pred[w]
(4)                 path.reverse()
(4)                 w = succ[path[-1]]
(4)                 while w is not None:
(8)                     path.append(w)
(8)                     w = succ[w]
(4)                 return path
(0)             def _bidirectional_pred_succ(G, source, target):
(4)                 """Bidirectional shortest path helper.
(4)                 Returns (pred, succ, w) where
(4)                 pred is a dictionary of predecessors from w to the source, and
(4)                 succ is a dictionary of successors from w to the target.
(4)                 """
(4)                 if target == source:
(8)                     return ({target: None}, {source: None}, source)
(4)                 if G.is_directed():
(8)                     Gpred = G.pred
(8)                     Gsucc = G.succ
(4)                 else:
(8)                     Gpred = G.adj
(8)                     Gsucc = G.adj
(4)                 pred = {source: None}
(4)                 succ = {target: None}
(4)                 forward_fringe = [source]
(4)                 reverse_fringe = [target]
(4)                 while forward_fringe and reverse_fringe:
(8)                     if len(forward_fringe) <= len(reverse_fringe):
(12)                        this_level = forward_fringe
(12)                        forward_fringe = []
(12)                        for v in this_level:
(16)                            for w in Gsucc[v]:
(20)                                if w not in pred:
(24)                                    forward_fringe.append(w)
(24)                                    pred[w] = v
(20)                                if w in succ:  # path found
(24)                                    return pred, succ, w
```

```
240: (8)                    else:
241: (12)                       this_level = reverse_fringe
242: (12)                       reverse_fringe = []
243: (12)                       for v in this_level:
244: (16)                           for w in Gpred[v]:
245: (20)                               if w not in succ:
246: (24)                                   succ[w] = v
247: (24)                                   reverse_fringe.append(w)
248: (20)                               if w in pred:  # found path
249: (24)                                   return pred, succ, w
250: (4)                raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
251: (0)            @nx._dispatchable
252: (0)            def single_source_shortest_path(G, source, cutoff=None):
253: (4)                """Compute shortest path between source
254: (4)                and all other nodes reachable from source.
255: (4)                Parameters
256: (4)                ----------
257: (4)                G : NetworkX graph
258: (4)                source : node label
259: (7)                   Starting node for path
260: (4)                cutoff : integer, optional
261: (8)                    Depth to stop the search. Only paths of length <= cutoff are returned.
262: (4)                Returns
263: (4)                -------
264: (4)                paths : dictionary
265: (8)                    Dictionary, keyed by target, of shortest paths.
266: (4)                Examples
267: (4)                --------
268: (4)                >>> G = nx.path_graph(5)
269: (4)                >>> path = nx.single_source_shortest_path(G, 0)
270: (4)                >>> path[4]
271: (4)                [0, 1, 2, 3, 4]
272: (4)                Notes
273: (4)                -----
274: (4)                The shortest path is not necessarily unique. So there can be multiple
275: (4)                paths between the source and each target node, all of which have the
276: (4)                same 'shortest' length. For each target node, this function returns
277: (4)                only one of those paths.
278: (4)                See Also
279: (4)                --------
280: (4)                shortest_path
281: (4)                """
282: (4)                if source not in G:
283: (8)                    raise nx.NodeNotFound(f"Source {source} not in G")
284: (4)                def join(p1, p2):
285: (8)                    return p1 + p2
286: (4)                if cutoff is None:
287: (8)                    cutoff = float("inf")
288: (4)                nextlevel = {source: 1}  # list of nodes to check at next level
289: (4)                paths = {source: [source]}  # paths dictionary  (paths to key from source)
290: (4)                return dict(_single_shortest_path(G.adj, nextlevel, paths, cutoff, join))
291: (0)            def _single_shortest_path(adj, firstlevel, paths, cutoff, join):
292: (4)                """Returns shortest paths
293: (4)                Shortest Path helper function
294: (4)                Parameters
295: (4)                ----------
296: (8)                    adj : dict
297: (12)                       Adjacency dict or view
298: (8)                    firstlevel : dict
299: (12)                       starting nodes, e.g. {source: 1} or {target: 1}
300: (8)                    paths : dict
301: (12)                       paths for starting nodes, e.g. {source: [source]}
302: (8)                    cutoff : int or float
303: (12)                       level at which we stop the process
304: (8)                    join : function
305: (12)                       function to construct a path from two partial paths. Requires two
306: (12)                       list inputs `p1` and `p2`, and returns a list. Usually returns
307: (12)                       `p1 + p2` (forward from source) or `p2 + p1` (backward from
target)
```

```
    """
    level = 0  # the current level
    nextlevel = firstlevel
    while nextlevel and cutoff > level:
        thislevel = nextlevel
        nextlevel = {}
        for v in thislevel:
            for w in adj[v]:
                if w not in paths:
                    paths[w] = join(paths[v], [w])
                    nextlevel[w] = 1
        level += 1
    return paths
@nx._dispatchable
def single_target_shortest_path(G, target, cutoff=None):
    """Compute shortest path to target from all nodes that reach target.
    Parameters
    ----------
    G : NetworkX graph
    target : node label
       Target node for path
    cutoff : integer, optional
        Depth to stop the search. Only paths of length <= cutoff are returned.
    Returns
    -------
    paths : dictionary
        Dictionary, keyed by target, of shortest paths.
    Examples
    --------
    >>> G = nx.path_graph(5, create_using=nx.DiGraph())
    >>> path = nx.single_target_shortest_path(G, 4)
    >>> path[0]
    [0, 1, 2, 3, 4]
    Notes
    -----
    The shortest path is not necessarily unique. So there can be multiple
    paths between the source and each target node, all of which have the
    same 'shortest' length. For each target node, this function returns
    only one of those paths.
    See Also
    --------
    shortest_path, single_source_shortest_path
    """
    if target not in G:
        raise nx.NodeNotFound(f"Target {target} not in G")
    def join(p1, p2):
        return p2 + p1
    adj = G.pred if G.is_directed() else G.adj
    if cutoff is None:
        cutoff = float("inf")
    nextlevel = {target: 1}  # list of nodes to check at next level
    paths = {target: [target]}  # paths dictionary  (paths to key from source)
    return dict(_single_shortest_path(adj, nextlevel, paths, cutoff, join))
@nx._dispatchable
def all_pairs_shortest_path(G, cutoff=None):
    """Compute shortest paths between all nodes.
    Parameters
    ----------
    G : NetworkX graph
    cutoff : integer, optional
        Depth at which to stop the search. Only paths of length at most
        `cutoff` are returned.
    Returns
    -------
    paths : iterator
        Dictionary, keyed by source and target, of shortest paths.
    Examples
    --------
    >>> G = nx.path_graph(5)
```

```
    >>> path = dict(nx.all_pairs_shortest_path(G))
    >>> print(path[0][4])
    [0, 1, 2, 3, 4]
    Notes
    -----
    There may be multiple shortest paths with the same length between
    two nodes. For each pair, this function returns only one of those paths.
    See Also
    --------
    floyd_warshall
    all_pairs_all_shortest_paths
    """
    for n in G:
        yield (n, single_source_shortest_path(G, n, cutoff=cutoff))
@nx._dispatchable
def predecessor(G, source, target=None, cutoff=None, return_seen=None):
    """Returns dict of predecessors for the path from source to all nodes in G.
    Parameters
    ----------
    G : NetworkX graph
    source : node label
       Starting node for path
    target : node label, optional
       Ending node for path. If provided only predecessors between
       source and target are returned
    cutoff : integer, optional
        Depth to stop the search. Only paths of length <= cutoff are returned.
    return_seen : bool, optional (default=None)
        Whether to return a dictionary, keyed by node, of the level (number of
        hops) to reach the node (as seen during breadth-first-search).
    Returns
    -------
    pred : dictionary
        Dictionary, keyed by node, of predecessors in the shortest path.
    (pred, seen): tuple of dictionaries
        If `return_seen` argument is set to `True`, then a tuple of dictionaries
        is returned. The first element is the dictionary, keyed by node, of
        predecessors in the shortest path. The second element is the dictionary,
        keyed by node, of the level (number of hops) to reach the node (as seen
        during breadth-first-search).
    Examples
    --------
    >>> G = nx.path_graph(4)
    >>> list(G)
    [0, 1, 2, 3]
    >>> nx.predecessor(G, 0)
    {0: [], 1: [0], 2: [1], 3: [2]}
    >>> nx.predecessor(G, 0, return_seen=True)
    ({0: [], 1: [0], 2: [1], 3: [2]}, {0: 0, 1: 1, 2: 2, 3: 3})
    """
    if source not in G:
        raise nx.NodeNotFound(f"Source {source} not in G")
    level = 0  # the current level
    nextlevel = [source]  # list of nodes to check at next level
    seen = {source: level}  # level (number of hops) when seen in BFS
    pred = {source: []}  # predecessor dictionary
    while nextlevel:
        level = level + 1
        thislevel = nextlevel
        nextlevel = []
        for v in thislevel:
            for w in G[v]:
                if w not in seen:
                    pred[w] = [v]
                    seen[w] = level
```

```
442: (20)                                nextlevel.append(w)
443: (16)                            elif seen[w] == level:  # add v to predecessor list if it
444: (20)                                pred[w].append(v)  # is at the correct level
445: (8)                     if cutoff and cutoff <= level:
446: (12)                        break
447: (4)                 if target is not None:
448: (8)                     if return_seen:
449: (12)                        if target not in pred:
450: (16)                            return ([], -1)  # No predecessor
451: (12)                        return (pred[target], seen[target])
452: (8)                     else:
453: (12)                        if target not in pred:
454: (16)                            return []  # No predecessor
455: (12)                        return pred[target]
456: (4)                 else:
457: (8)                     if return_seen:
458: (12)                        return (pred, seen)
459: (8)                     else:
460: (12)                        return pred
```

File 297 - weighted.py:

```
1: (0)              """
2: (0)              Shortest path algorithms for weighted graphs.
3: (0)              """
4: (0)              from collections import deque
5: (0)              from heapq import heappop, heappush
6: (0)              from itertools import count
7: (0)              import networkx as nx
8: (0)              from networkx.algorithms.shortest_paths.generic import
_build_paths_from_predecessors
9: (0)              __all__ = [
10: (4)                 "dijkstra_path",
11: (4)                 "dijkstra_path_length",
12: (4)                 "bidirectional_dijkstra",
13: (4)                 "single_source_dijkstra",
14: (4)                 "single_source_dijkstra_path",
15: (4)                 "single_source_dijkstra_path_length",
16: (4)                 "multi_source_dijkstra",
17: (4)                 "multi_source_dijkstra_path",
18: (4)                 "multi_source_dijkstra_path_length",
19: (4)                 "all_pairs_dijkstra",
20: (4)                 "all_pairs_dijkstra_path",
21: (4)                 "all_pairs_dijkstra_path_length",
22: (4)                 "dijkstra_predecessor_and_distance",
23: (4)                 "bellman_ford_path",
24: (4)                 "bellman_ford_path_length",
25: (4)                 "single_source_bellman_ford",
26: (4)                 "single_source_bellman_ford_path",
27: (4)                 "single_source_bellman_ford_path_length",
28: (4)                 "all_pairs_bellman_ford_path",
29: (4)                 "all_pairs_bellman_ford_path_length",
30: (4)                 "bellman_ford_predecessor_and_distance",
31: (4)                 "negative_edge_cycle",
32: (4)                 "find_negative_cycle",
33: (4)                 "goldberg_radzik",
34: (4)                 "johnson",
35: (0)             ]
36: (0)             def _weight_function(G, weight):
37: (4)                 """Returns a function that returns the weight of an edge.
38: (4)                 The returned function is specifically suitable for input to
39: (4)                 functions :func:`_dijkstra` and :func:`_bellman_ford_relaxation`.
40: (4)                 Parameters
41: (4)                 ----------
42: (4)                 G : NetworkX graph.
43: (4)                 weight : string or function
44: (8)                     If it is callable, `weight` itself is returned. If it is a string,
```

```
(8)                 it is assumed to be the name of the edge attribute that represents
(8)                 the weight of an edge. In that case, a function is returned that
(8)                 gets the edge weight according to the specified edge attribute.
(4)             Returns
(4)             -------
(4)             function
(8)                 This function returns a callable that accepts exactly three inputs:
(8)                 a node, an node adjacent to the first one, and the edge attribute
(8)                 dictionary for the eedge joining those nodes. That function returns
(8)                 a number representing the weight of an edge.
(4)             If `G` is a multigraph, and `weight` is not callable, the
(4)             minimum edge weight over all parallel edges is returned. If any edge
(4)             does not have an attribute with key `weight`, it is assumed to
(4)             have weight one.
(4)             """
(4)             if callable(weight):
(8)                 return weight
(4)             if G.is_multigraph():
(8)                 return lambda u, v, d: min(attr.get(weight, 1) for attr in d.values())
(4)             return lambda u, v, data: data.get(weight, 1)
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def dijkstra_path(G, source, target, weight="weight"):
(4)             """Returns the shortest weighted path from source to target in G.
(4)             Uses Dijkstra's Method to compute the shortest weighted path
(4)             between two nodes in a graph.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source : node
(8)                 Starting node
(4)             target : node
(8)                 Ending node
(4)             weight : string or function
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number or None to indicate a hidden edge.
(4)             Returns
(4)             -------
(4)             path : list
(8)                 List of nodes in a shortest path.
(4)             Raises
(4)             ------
(4)             NodeNotFound
(8)                 If `source` is not in `G`.
(4)             NetworkXNoPath
(8)                 If no path exists between source and target.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> print(nx.dijkstra_path(G, 0, 4))
(4)             [0, 1, 2, 3, 4]
(4)             Find edges of shortest path in Multigraph
(4)             >>> G = nx.MultiDiGraph()
(4)             >>> G.add_weighted_edges_from([(1, 2, 0.75), (1, 2, 0.5), (2, 3, 0.5), (1,
3, 1.5)])
(4)             >>> nodes = nx.dijkstra_path(G, 1, 3)
(4)             >>> edges = nx.utils.pairwise(nodes)
(4)             >>> list(
(4)             ...     (u, v, min(G[u][v], key=lambda k: G[u][v][k].get("weight", 1)))
(4)             ...     for u, v in edges
(4)             ... )
(4)             [(1, 2, 1), (2, 3, 0)]
```

```
113: (4)                Notes
114: (4)                -----
115: (4)                Edge weight attributes must be numerical.
116: (4)                Distances are calculated as sums of weighted edges traversed.
117: (4)                The weight function can be used to hide edges by returning None.
118: (4)                So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
119: (4)                will find the shortest red path.
120: (4)                The weight function can be used to include node weights.
121: (4)                >>> def func(u, v, d):
122: (4)                ...     node_u_wt = G.nodes[u].get("node_weight", 1)
123: (4)                ...     node_v_wt = G.nodes[v].get("node_weight", 1)
124: (4)                ...     edge_wt = d.get("weight", 1)
125: (4)                ...     return node_u_wt / 2 + node_v_wt / 2 + edge_wt
126: (4)                In this example we take the average of start and end node
127: (4)                weights of an edge and add it to the weight of the edge.
128: (4)                The function :func:`single_source_dijkstra` computes both
129: (4)                path and length-of-path if you need both, use that.
130: (4)                See Also
131: (4)                --------
132: (4)                bidirectional_dijkstra
133: (4)                bellman_ford_path
134: (4)                single_source_dijkstra
135: (4)                """
136: (4)                (length, path) = single_source_dijkstra(G, source, target=target,
weight=weight)
137: (4)                return path
138: (0)            @nx._dispatchable(edge_attrs="weight")
139: (0)            def dijkstra_path_length(G, source, target, weight="weight"):
140: (4)                """Returns the shortest weighted path length in G from source to target.
141: (4)                Uses Dijkstra's Method to compute the shortest weighted path length
142: (4)                between two nodes in a graph.
143: (4)                Parameters
144: (4)                ----------
145: (4)                G : NetworkX graph
146: (4)                source : node label
147: (8)                    starting node for path
148: (4)                target : node label
149: (8)                    ending node for path
150: (4)                weight : string or function
151: (8)                    If this is a string, then edge weights will be accessed via the
152: (8)                    edge attribute with this key (that is, the weight of the edge
153: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
154: (8)                    such edge attribute exists, the weight of the edge is assumed to
155: (8)                    be one.
156: (8)                    If this is a function, the weight of an edge is the value
157: (8)                    returned by the function. The function must accept exactly three
158: (8)                    positional arguments: the two endpoints of an edge and the
159: (8)                    dictionary of edge attributes for that edge. The function must
160: (8)                    return a number or None to indicate a hidden edge.
161: (4)                Returns
162: (4)                -------
163: (4)                length : number
164: (8)                    Shortest path length.
165: (4)                Raises
166: (4)                ------
167: (4)                NodeNotFound
168: (8)                    If `source` is not in `G`.
169: (4)                NetworkXNoPath
170: (8)                    If no path exists between source and target.
171: (4)                Examples
172: (4)                --------
173: (4)                >>> G = nx.path_graph(5)
174: (4)                >>> nx.dijkstra_path_length(G, 0, 4)
175: (4)                4
176: (4)                Notes
177: (4)                -----
178: (4)                Edge weight attributes must be numerical.
179: (4)                Distances are calculated as sums of weighted edges traversed.
180: (4)                The weight function can be used to hide edges by returning None.
```

```
181: (4)                So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
182: (4)                will find the shortest red path.
183: (4)                The function :func:`single_source_dijkstra` computes both
184: (4)                path and length-of-path if you need both, use that.
185: (4)                See Also
186: (4)                --------
187: (4)                bidirectional_dijkstra
188: (4)                bellman_ford_path_length
189: (4)                single_source_dijkstra
190: (4)                """
191: (4)                if source not in G:
192: (8)                    raise nx.NodeNotFound(f"Node {source} not found in graph")
193: (4)                if source == target:
194: (8)                    return 0
195: (4)                weight = _weight_function(G, weight)
196: (4)                length = _dijkstra(G, source, weight, target=target)
197: (4)                try:
198: (8)                    return length[target]
199: (4)                except KeyError as err:
200: (8)                    raise nx.NetworkXNoPath(f"Node {target} not reachable from {source}")
from err
201: (0)            @nx._dispatchable(edge_attrs="weight")
202: (0)            def single_source_dijkstra_path(G, source, cutoff=None, weight="weight"):
203: (4)                """Find shortest weighted paths in G from a source node.
204: (4)                Compute shortest path between source and all other reachable
205: (4)                nodes for a weighted graph.
206: (4)                Parameters
207: (4)                ----------
208: (4)                G : NetworkX graph
209: (4)                source : node
210: (8)                    Starting node for path.
211: (4)                cutoff : integer or float, optional
212: (8)                    Length (sum of edge weights) at which the search is stopped.
213: (8)                    If cutoff is provided, only return paths with summed weight <= cutoff.
214: (4)                weight : string or function
215: (8)                    If this is a string, then edge weights will be accessed via the
216: (8)                    edge attribute with this key (that is, the weight of the edge
217: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
218: (8)                    such edge attribute exists, the weight of the edge is assumed to
219: (8)                    be one.
220: (8)                    If this is a function, the weight of an edge is the value
221: (8)                    returned by the function. The function must accept exactly three
222: (8)                    positional arguments: the two endpoints of an edge and the
223: (8)                    dictionary of edge attributes for that edge. The function must
224: (8)                    return a number or None to indicate a hidden edge.
225: (4)                Returns
226: (4)                -------
227: (4)                paths : dictionary
228: (8)                    Dictionary of shortest path lengths keyed by target.
229: (4)                Raises
230: (4)                ------
231: (4)                NodeNotFound
232: (8)                    If `source` is not in `G`.
233: (4)                Examples
234: (4)                --------
235: (4)                >>> G = nx.path_graph(5)
236: (4)                >>> path = nx.single_source_dijkstra_path(G, 0)
237: (4)                >>> path[4]
238: (4)                [0, 1, 2, 3, 4]
239: (4)                Notes
240: (4)                -----
241: (4)                Edge weight attributes must be numerical.
242: (4)                Distances are calculated as sums of weighted edges traversed.
243: (4)                The weight function can be used to hide edges by returning None.
244: (4)                So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
245: (4)                will find the shortest red path.
246: (4)                See Also
247: (4)                --------
248: (4)                single_source_dijkstra, single_source_bellman_ford
```

```
    """
    return multi_source_dijkstra_path(G, {source}, cutoff=cutoff, weight=weight)
@nx._dispatchable(edge_attrs="weight")
def single_source_dijkstra_path_length(G, source, cutoff=None, weight="weight"):
    """Find shortest weighted path lengths in G from a source node.
    Compute the shortest path length between source and all other
    reachable nodes for a weighted graph.
    Parameters
    ----------
    G : NetworkX graph
    source : node label
        Starting node for path
    cutoff : integer or float, optional
        Length (sum of edge weights) at which the search is stopped.
        If cutoff is provided, only return paths with summed weight <= cutoff.
    weight : string or function
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly three
        positional arguments: the two endpoints of an edge and the
        dictionary of edge attributes for that edge. The function must
        return a number or None to indicate a hidden edge.
    Returns
    -------
    length : dict
        Dict keyed by node to shortest path length from source.
    Raises
    ------
    NodeNotFound
        If `source` is not in `G`.
    Examples
    --------
    >>> G = nx.path_graph(5)
    >>> length = nx.single_source_dijkstra_path_length(G, 0)
    >>> length[4]
    4
    >>> for node in [0, 1, 2, 3, 4]:
    ...     print(f"{node}: {length[node]}")
    0: 0
    1: 1
    2: 2
    3: 3
    4: 4
    Notes
    -----
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    The weight function can be used to hide edges by returning None.
    So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
    will find the shortest red path.
    See Also
    --------
    single_source_dijkstra, single_source_bellman_ford_path_length
    """
    return multi_source_dijkstra_path_length(G, {source}, cutoff=cutoff, weight=weight)
@nx._dispatchable(edge_attrs="weight")
def single_source_dijkstra(G, source, target=None, cutoff=None, weight="weight"):
    """Find shortest weighted paths and lengths from a source node.
    Compute the shortest path length between source and all other
    reachable nodes for a weighted graph.
    Uses Dijkstra's algorithm to compute shortest paths and lengths
```

```
(4)                 between a source and all other reachable nodes in a weighted graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node label
(8)                     Starting node for path
(4)                 target : node label, optional
(8)                     Ending node for path
(4)                 cutoff : integer or float, optional
(8)                     Length (sum of edge weights) at which the search is stopped.
(8)                     If cutoff is provided, only return paths with summed weight <= cutoff.
(4)                 weight : string or function
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number or None to indicate a hidden edge.
(4)                 Returns
(4)                 -------
(4)                 distance, path : pair of dictionaries, or numeric and list.
(8)                     If target is None, paths and lengths to all nodes are computed.
(8)                     The return value is a tuple of two dictionaries keyed by target nodes.
(8)                     The first dictionary stores distance to each target node.
(8)                     The second stores the path to each target node.
(8)                     If target is not None, returns a tuple (distance, path), where
(8)                     distance is the distance from source to target and path is a list
(8)                     representing the path from source to target.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(8)                     If `source` is not in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> length, path = nx.single_source_dijkstra(G, 0)
(4)                 >>> length[4]
(4)                 4
(4)                 >>> for node in [0, 1, 2, 3, 4]:
(4)                 ...     print(f"{node}: {length[node]}")
(4)                 0: 0
(4)                 1: 1
(4)                 2: 2
(4)                 3: 3
(4)                 4: 4
(4)                 >>> path[4]
(4)                 [0, 1, 2, 3, 4]
(4)                 >>> length, path = nx.single_source_dijkstra(G, 0, 1)
(4)                 >>> length
(4)                 1
(4)                 >>> path
(4)                 [0, 1]
(4)                 Notes
(4)                 -----
(4)                 Edge weight attributes must be numerical.
(4)                 Distances are calculated as sums of weighted edges traversed.
(4)                 The weight function can be used to hide edges by returning None.
(4)                 So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
(4)                 will find the shortest red path.
(4)                 Based on the Python cookbook recipe (119466) at
(4)                 https://code.activestate.com/recipes/119466/
(4)                 This algorithm is not guaranteed to work if edge weights
(4)                 are negative or are floating point numbers
(4)                 (overflows and roundoff errors can cause problems).
(4)                 See Also
```

```
383: (4)                --------
384: (4)                single_source_dijkstra_path
385: (4)                single_source_dijkstra_path_length
386: (4)                single_source_bellman_ford
387: (4)                """
388: (4)                return multi_source_dijkstra(
389: (8)                    G, {source}, cutoff=cutoff, target=target, weight=weight
390: (4)                )
391: (0)            @nx._dispatchable(edge_attrs="weight")
392: (0)            def multi_source_dijkstra_path(G, sources, cutoff=None, weight="weight"):
393: (4)                """Find shortest weighted paths in G from a given set of source
394: (4)                nodes.
395: (4)                Compute shortest path between any of the source nodes and all other
396: (4)                reachable nodes for a weighted graph.
397: (4)                Parameters
398: (4)                ----------
399: (4)                G : NetworkX graph
400: (4)                sources : non-empty set of nodes
401: (8)                    Starting nodes for paths. If this is just a set containing a
402: (8)                    single node, then all paths computed by this function will start
403: (8)                    from that node. If there are two or more nodes in the set, the
404: (8)                    computed paths may begin from any one of the start nodes.
405: (4)                cutoff : integer or float, optional
406: (8)                    Length (sum of edge weights) at which the search is stopped.
407: (8)                    If cutoff is provided, only return paths with summed weight <= cutoff.
408: (4)                weight : string or function
409: (8)                    If this is a string, then edge weights will be accessed via the
410: (8)                    edge attribute with this key (that is, the weight of the edge
411: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
412: (8)                    such edge attribute exists, the weight of the edge is assumed to
413: (8)                    be one.
414: (8)                    If this is a function, the weight of an edge is the value
415: (8)                    returned by the function. The function must accept exactly three
416: (8)                    positional arguments: the two endpoints of an edge and the
417: (8)                    dictionary of edge attributes for that edge. The function must
418: (8)                    return a number or None to indicate a hidden edge.
419: (4)                Returns
420: (4)                -------
421: (4)                paths : dictionary
422: (8)                    Dictionary of shortest paths keyed by target.
423: (4)                Examples
424: (4)                --------
425: (4)                >>> G = nx.path_graph(5)
426: (4)                >>> path = nx.multi_source_dijkstra_path(G, {0, 4})
427: (4)                >>> path[1]
428: (4)                [0, 1]
429: (4)                >>> path[3]
430: (4)                [4, 3]
431: (4)                Notes
432: (4)                -----
433: (4)                Edge weight attributes must be numerical.
434: (4)                Distances are calculated as sums of weighted edges traversed.
435: (4)                The weight function can be used to hide edges by returning None.
436: (4)                So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
437: (4)                will find the shortest red path.
438: (4)                Raises
439: (4)                ------
440: (4)                ValueError
441: (8)                    If `sources` is empty.
442: (4)                NodeNotFound
443: (8)                    If any of `sources` is not in `G`.
444: (4)                See Also
445: (4)                --------
446: (4)                multi_source_dijkstra, multi_source_bellman_ford
447: (4)                """
448: (4)                length, path = multi_source_dijkstra(G, sources, cutoff=cutoff,
weight=weight)
449: (4)                return path
450: (0)            @nx._dispatchable(edge_attrs="weight")
```

```
(0)  def multi_source_dijkstra_path_length(G, sources, cutoff=None,
weight="weight"):
(4)      """Find shortest weighted path lengths in G from a given set of
(4)      source nodes.
(4)      Compute the shortest path length between any of the source nodes and
(4)      all other reachable nodes for a weighted graph.
(4)      Parameters
(4)      ----------
(4)      G : NetworkX graph
(4)      sources : non-empty set of nodes
(8)          Starting nodes for paths. If this is just a set containing a
(8)          single node, then all paths computed by this function will start
(8)          from that node. If there are two or more nodes in the set, the
(8)          computed paths may begin from any one of the start nodes.
(4)      cutoff : integer or float, optional
(8)          Length (sum of edge weights) at which the search is stopped.
(8)          If cutoff is provided, only return paths with summed weight <= cutoff.
(4)      weight : string or function
(8)          If this is a string, then edge weights will be accessed via the
(8)          edge attribute with this key (that is, the weight of the edge
(8)          joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)          such edge attribute exists, the weight of the edge is assumed to
(8)          be one.
(8)          If this is a function, the weight of an edge is the value
(8)          returned by the function. The function must accept exactly three
(8)          positional arguments: the two endpoints of an edge and the
(8)          dictionary of edge attributes for that edge. The function must
(8)          return a number or None to indicate a hidden edge.
(4)      Returns
(4)      -------
(4)      length : dict
(8)          Dict keyed by node to shortest path length to nearest source.
(4)      Examples
(4)      --------
(4)      >>> G = nx.path_graph(5)
(4)      >>> length = nx.multi_source_dijkstra_path_length(G, {0, 4})
(4)      >>> for node in [0, 1, 2, 3, 4]:
(4)      ...     print(f"{node}: {length[node]}")
(4)      0: 0
(4)      1: 1
(4)      2: 2
(4)      3: 1
(4)      4: 0
(4)      Notes
(4)      -----
(4)      Edge weight attributes must be numerical.
(4)      Distances are calculated as sums of weighted edges traversed.
(4)      The weight function can be used to hide edges by returning None.
(4)      So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
(4)      will find the shortest red path.
(4)      Raises
(4)      ------
(4)      ValueError
(8)          If `sources` is empty.
(4)      NodeNotFound
(8)          If any of `sources` is not in `G`.
(4)      See Also
(4)      --------
(4)      multi_source_dijkstra
(4)      """
(4)      if not sources:
(8)          raise ValueError("sources must not be empty")
(4)      for s in sources:
(8)          if s not in G:
(12)             raise nx.NodeNotFound(f"Node {s} not found in graph")
(4)      weight = _weight_function(G, weight)
(4)      return _dijkstra_multisource(G, sources, weight, cutoff=cutoff)
(0)  @nx._dispatchable(edge_attrs="weight")
(0)  def multi_source_dijkstra(G, sources, target=None, cutoff=None,
```

```
weight="weight"):
    """Find shortest weighted paths and lengths from a given set of
    source nodes.
    Uses Dijkstra's algorithm to compute the shortest paths and lengths
    between one of the source nodes and the given `target`, or all other
    reachable nodes if not specified, for a weighted graph.
    Parameters
    ----------
    G : NetworkX graph
    sources : non-empty set of nodes
        Starting nodes for paths. If this is just a set containing a
        single node, then all paths computed by this function will start
        from that node. If there are two or more nodes in the set, the
        computed paths may begin from any one of the start nodes.
    target : node label, optional
        Ending node for path
    cutoff : integer or float, optional
        Length (sum of edge weights) at which the search is stopped.
        If cutoff is provided, only return paths with summed weight <= cutoff.
    weight : string or function
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly three
        positional arguments: the two endpoints of an edge and the
        dictionary of edge attributes for that edge. The function must
        return a number or None to indicate a hidden edge.
    Returns
    -------
    distance, path : pair of dictionaries, or numeric and list
        If target is None, returns a tuple of two dictionaries keyed by node.
        The first dictionary stores distance from one of the source nodes.
        The second stores the path from one of the sources to that node.
        If target is not None, returns a tuple of (distance, path) where
        distance is the distance from source to target and path is a list
        representing the path from source to target.
    Examples
    --------
    >>> G = nx.path_graph(5)
    >>> length, path = nx.multi_source_dijkstra(G, {0, 4})
    >>> for node in [0, 1, 2, 3, 4]:
    ...     print(f"{node}: {length[node]}")
    0: 0
    1: 1
    2: 2
    3: 1
    4: 0
    >>> path[1]
    [0, 1]
    >>> path[3]
    [4, 3]
    >>> length, path = nx.multi_source_dijkstra(G, {0, 4}, 1)
    >>> length
    1
    >>> path
    [0, 1]
    Notes
    -----
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    The weight function can be used to hide edges by returning None.
    So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
    will find the shortest red path.
    Based on the Python cookbook recipe (119466) at
    https://code.activestate.com/recipes/119466/
    This algorithm is not guaranteed to work if edge weights
```

```
(4)                 are negative or are floating point numbers
(4)                 (overflows and roundoff errors can cause problems).
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If `sources` is empty.
(4)                 NodeNotFound
(8)                     If any of `sources` is not in `G`.
(4)                 See Also
(4)                 --------
(4)                 multi_source_dijkstra_path
(4)                 multi_source_dijkstra_path_length
(4)                 """
(4)                 if not sources:
(8)                     raise ValueError("sources must not be empty")
(4)                 for s in sources:
(8)                     if s not in G:
(12)                        raise nx.NodeNotFound(f"Node {s} not found in graph")
(4)                 if target in sources:
(8)                     return (0, [target])
(4)                 weight = _weight_function(G, weight)
(4)                 paths = {source: [source] for source in sources}  # dictionary of paths
(4)                 dist = _dijkstra_multisource(
(8)                     G, sources, weight, paths=paths, cutoff=cutoff, target=target
(4)                 )
(4)                 if target is None:
(8)                     return (dist, paths)
(4)                 try:
(8)                     return (dist[target], paths[target])
(4)                 except KeyError as err:
(8)                     raise nx.NetworkXNoPath(f"No path to {target}.") from err
(0)             def _dijkstra(G, source, weight, pred=None, paths=None, cutoff=None,
target=None):
(4)                 """Uses Dijkstra's algorithm to find shortest weighted paths from a
(4)                 single source.
(4)                 This is a convenience function for :func:`_dijkstra_multisource`
(4)                 with all the arguments the same, except the keyword argument
(4)                 `sources` set to ``[source]``.
(4)                 """
(4)                 return _dijkstra_multisource(
(8)                     G, [source], weight, pred=pred, paths=paths, cutoff=cutoff,
target=target
(4)                 )
(0)             def _dijkstra_multisource(
(4)                 G, sources, weight, pred=None, paths=None, cutoff=None, target=None
(0)             ):
(4)                 """Uses Dijkstra's algorithm to find shortest weighted paths
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 sources : non-empty iterable of nodes
(8)                     Starting nodes for paths. If this is just an iterable containing
(8)                     a single node, then all paths computed by this function will
(8)                     start from that node. If there are two or more nodes in this
(8)                     iterable, the computed paths may begin from any one of the start
(8)                     nodes.
(4)                 weight: function
(8)                     Function with (u, v, data) input that returns that edge's weight
(8)                     or None to indicate a hidden edge
(4)                 pred: dict of lists, optional(default=None)
(8)                     dict to store a list of predecessors keyed by that node
(8)                     If None, predecessors are not stored.
(4)                 paths: dict, optional (default=None)
(8)                     dict to store the path list from source to each node, keyed by node.
(8)                     If None, paths are not stored.
(4)                 target : node label, optional
(8)                     Ending node for path. Search is halted when target is found.
(4)                 cutoff : integer or float, optional
(8)                     Length (sum of edge weights) at which the search is stopped.
```

```
654: (8)                    If cutoff is provided, only return paths with summed weight <= cutoff.
655: (4)                Returns
656: (4)                -------
657: (4)                distance : dictionary
658: (8)                    A mapping from node to shortest distance to that node from one
659: (8)                    of the source nodes.
660: (4)                Raises
661: (4)                ------
662: (4)                NodeNotFound
663: (8)                    If any of `sources` is not in `G`.
664: (4)                Notes
665: (4)                -----
666: (4)                The optional predecessor and path dictionaries can be accessed by
667: (4)                the caller through the original pred and paths objects passed
668: (4)                as arguments. No need to explicitly return pred or paths.
669: (4)                """
670: (4)                G_succ = G._adj  # For speed-up (and works for both directed and
undirected graphs)
671: (4)                push = heappush
672: (4)                pop = heappop
673: (4)                dist = {}  # dictionary of final distances
674: (4)                seen = {}
675: (4)                c = count()
676: (4)                fringe = []
677: (4)                for source in sources:
678: (8)                    seen[source] = 0
679: (8)                    push(fringe, (0, next(c), source))
680: (4)                while fringe:
681: (8)                    (d, _, v) = pop(fringe)
682: (8)                    if v in dist:
683: (12)                       continue  # already searched this node.
684: (8)                    dist[v] = d
685: (8)                    if v == target:
686: (12)                       break
687: (8)                    for u, e in G_succ[v].items():
688: (12)                       cost = weight(v, u, e)
689: (12)                       if cost is None:
690: (16)                           continue
691: (12)                       vu_dist = dist[v] + cost
692: (12)                       if cutoff is not None:
693: (16)                           if vu_dist > cutoff:
694: (20)                               continue
695: (12)                       if u in dist:
696: (16)                           u_dist = dist[u]
697: (16)                           if vu_dist < u_dist:
698: (20)                               raise ValueError("Contradictory paths found:", "negative
weights?")
699: (16)                           elif pred is not None and vu_dist == u_dist:
700: (20)                               pred[u].append(v)
701: (12)                       elif u not in seen or vu_dist < seen[u]:
702: (16)                           seen[u] = vu_dist
703: (16)                           push(fringe, (vu_dist, next(c), u))
704: (16)                           if paths is not None:
705: (20)                               paths[u] = paths[v] + [u]
706: (16)                           if pred is not None:
707: (20)                               pred[u] = [v]
708: (12)                       elif vu_dist == seen[u]:
709: (16)                           if pred is not None:
710: (20)                               pred[u].append(v)
711: (4)                return dist
712: (0)            @nx._dispatchable(edge_attrs="weight")
713: (0)            def dijkstra_predecessor_and_distance(G, source, cutoff=None,
weight="weight"):
714: (4)                """Compute weighted shortest path length and predecessors.
715: (4)                Uses Dijkstra's Method to obtain the shortest weighted paths
716: (4)                and return dictionaries of predecessors for each node and
717: (4)                distance for each node from the `source`.
718: (4)                Parameters
719: (4)                ----------
```

```
    G : NetworkX graph
    source : node label
        Starting node for path
    cutoff : integer or float, optional
        Length (sum of edge weights) at which the search is stopped.
        If cutoff is provided, only return paths with summed weight <= cutoff.
    weight : string or function
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly three
        positional arguments: the two endpoints of an edge and the
        dictionary of edge attributes for that edge. The function must
        return a number or None to indicate a hidden edge.
    Returns
    -------
    pred, distance : dictionaries
        Returns two dictionaries representing a list of predecessors
        of a node and the distance to each node.
    Raises
    ------
    NodeNotFound
        If `source` is not in `G`.
    Notes
    -----
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    The list of predecessors contains more than one element only when
    there are more than one shortest paths to the key node.
    Examples
    --------
    >>> G = nx.path_graph(5, create_using=nx.DiGraph())
    >>> pred, dist = nx.dijkstra_predecessor_and_distance(G, 0)
    >>> sorted(pred.items())
    [(0, []), (1, [0]), (2, [1]), (3, [2]), (4, [3])]
    >>> sorted(dist.items())
    [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
    >>> pred, dist = nx.dijkstra_predecessor_and_distance(G, 0, 1)
    >>> sorted(pred.items())
    [(0, []), (1, [0])]
    >>> sorted(dist.items())
    [(0, 0), (1, 1)]
    """
    if source not in G:
        raise nx.NodeNotFound(f"Node {source} is not found in the graph")
    weight = _weight_function(G, weight)
    pred = {source: []}  # dictionary of predecessors
    return (pred, _dijkstra(G, source, weight, pred=pred, cutoff=cutoff))
@nx._dispatchable(edge_attrs="weight")
def all_pairs_dijkstra(G, cutoff=None, weight="weight"):
    """Find shortest weighted paths and lengths between all nodes.
    Parameters
    ----------
    G : NetworkX graph
    cutoff : integer or float, optional
        Length (sum of edge weights) at which the search is stopped.
        If cutoff is provided, only return paths with summed weight <= cutoff.
    weight : string or function
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edge[u][v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly three
        positional arguments: the two endpoints of an edge and the
```

```
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number or None to indicate a hidden edge.
(4)             Yields
(4)             ------
(4)             (node, (distance, path)) : (node obj, (dict, dict))
(8)                 Each source node has two associated dicts. The first holds distance
(8)                 keyed by target and the second holds paths keyed by target.
(8)                 (See single_source_dijkstra for the source/target node terminology.)
(8)                 If desired you can apply `dict()` to this function to create a dict
(8)                 keyed by source node to the two dicts.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> len_path = dict(nx.all_pairs_dijkstra(G))
(4)             >>> len_path[3][0][1]
(4)             2
(4)             >>> for node in [0, 1, 2, 3, 4]:
(4)             ...     print(f"3 - {node}: {len_path[3][0][node]}")
(4)             3 - 0: 3
(4)             3 - 1: 2
(4)             3 - 2: 1
(4)             3 - 3: 0
(4)             3 - 4: 1
(4)             >>> len_path[3][1][1]
(4)             [3, 2, 1]
(4)             >>> for n, (dist, path) in nx.all_pairs_dijkstra(G):
(4)             ...     print(path[1])
(4)             [0, 1]
(4)             [1]
(4)             [2, 1]
(4)             [3, 2, 1]
(4)             [4, 3, 2, 1]
(4)             Notes
(4)             -----
(4)             Edge weight attributes must be numerical.
(4)             Distances are calculated as sums of weighted edges traversed.
(4)             The yielded dicts only have keys for reachable nodes.
(4)             """
(4)             for n in G:
(8)                 dist, path = single_source_dijkstra(G, n, cutoff=cutoff,
weight=weight)
(8)                 yield (n, (dist, path))
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def all_pairs_dijkstra_path_length(G, cutoff=None, weight="weight"):
(4)             """Compute shortest path lengths between all nodes in a weighted graph.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             cutoff : integer or float, optional
(8)                 Length (sum of edge weights) at which the search is stopped.
(8)                 If cutoff is provided, only return paths with summed weight <= cutoff.
(4)             weight : string or function
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number or None to indicate a hidden edge.
(4)             Returns
(4)             -------
(4)             distance : iterator
(8)                 (source, dictionary) iterator with dictionary keyed by target and
(8)                 shortest path length as the key value.
(4)             Examples
(4)             --------
```

```
857: (4)                >>> G = nx.path_graph(5)
858: (4)                >>> length = dict(nx.all_pairs_dijkstra_path_length(G))
859: (4)                >>> for node in [0, 1, 2, 3, 4]:
860: (4)                ...     print(f"1 - {node}: {length[1][node]}")
861: (4)                1 - 0: 1
862: (4)                1 - 1: 0
863: (4)                1 - 2: 1
864: (4)                1 - 3: 2
865: (4)                1 - 4: 3
866: (4)                >>> length[3][2]
867: (4)                1
868: (4)                >>> length[2][2]
869: (4)                0
870: (4)                Notes
871: (4)                -----
872: (4)                Edge weight attributes must be numerical.
873: (4)                Distances are calculated as sums of weighted edges traversed.
874: (4)                The dictionary returned only has keys for reachable node pairs.
875: (4)                """
876: (4)                length = single_source_dijkstra_path_length
877: (4)                for n in G:
878: (8)                    yield (n, length(G, n, cutoff=cutoff, weight=weight))
879: (0)            @nx._dispatchable(edge_attrs="weight")
880: (0)            def all_pairs_dijkstra_path(G, cutoff=None, weight="weight"):
881: (4)                """Compute shortest paths between all nodes in a weighted graph.
882: (4)                Parameters
883: (4)                ----------
884: (4)                G : NetworkX graph
885: (4)                cutoff : integer or float, optional
886: (8)                    Length (sum of edge weights) at which the search is stopped.
887: (8)                    If cutoff is provided, only return paths with summed weight <= cutoff.
888: (4)                weight : string or function
889: (8)                    If this is a string, then edge weights will be accessed via the
890: (8)                    edge attribute with this key (that is, the weight of the edge
891: (8)                    joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
892: (8)                    such edge attribute exists, the weight of the edge is assumed to
893: (8)                    be one.
894: (8)                    If this is a function, the weight of an edge is the value
895: (8)                    returned by the function. The function must accept exactly three
896: (8)                    positional arguments: the two endpoints of an edge and the
897: (8)                    dictionary of edge attributes for that edge. The function must
898: (8)                    return a number or None to indicate a hidden edge.
899: (4)                Returns
900: (4)                -------
901: (4)                paths : iterator
902: (8)                    (source, dictionary) iterator with dictionary keyed by target and
903: (8)                    shortest path as the key value.
904: (4)                Examples
905: (4)                --------
906: (4)                >>> G = nx.path_graph(5)
907: (4)                >>> path = dict(nx.all_pairs_dijkstra_path(G))
908: (4)                >>> path[0][4]
909: (4)                [0, 1, 2, 3, 4]
910: (4)                Notes
911: (4)                -----
912: (4)                Edge weight attributes must be numerical.
913: (4)                Distances are calculated as sums of weighted edges traversed.
914: (4)                See Also
915: (4)                --------
916: (4)                floyd_warshall, all_pairs_bellman_ford_path
917: (4)                """
918: (4)                path = single_source_dijkstra_path
919: (4)                for n in G:
920: (8)                    yield (n, path(G, n, cutoff=cutoff, weight=weight))
921: (0)            @nx._dispatchable(edge_attrs="weight")
922: (0)            def bellman_ford_predecessor_and_distance(
923: (4)                G, source, target=None, weight="weight", heuristic=False
924: (0)            ):
925: (4)                """Compute shortest path lengths and predecessors on shortest paths
```

```
(4)                 in weighted graphs.
(4)                 The algorithm has a running time of $O(mn)$ where $n$ is the number of
(4)                 nodes and $m$ is the number of edges.  It is slower than Dijkstra but
(4)                 can handle negative edge weights.
(4)                 If a negative cycle is detected, you can use :func:`find_negative_cycle`
(4)                 to return the cycle and examine it. Shortest paths are not defined when
(4)                 a negative cycle exists because once reached, the path can cycle forever
(4)                 to build up arbitrarily low weights.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     The algorithm works for all types of graphs, including directed
(8)                     graphs and multigraphs.
(4)                 source: node label
(8)                     Starting node for path
(4)                 target : node label, optional
(8)                     Ending node for path
(4)                 weight : string or function
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number.
(4)                 heuristic : bool
(8)                     Determines whether to use a heuristic to early detect negative
(8)                     cycles at a hopefully negligible cost.
(4)                 Returns
(4)                 -------
(4)                 pred, dist : dictionaries
(8)                     Returns two dictionaries keyed by node to predecessor in the
(8)                     path and to the distance from the source respectively.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(8)                     If `source` is not in `G`.
(4)                 NetworkXUnbounded
(8)                     If the (di)graph contains a negative (di)cycle, the
(8)                     algorithm raises an exception to indicate the presence of the
(8)                     negative (di)cycle.  Note: any negative weight edge in an
(8)                     undirected graph is a negative cycle.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5, create_using=nx.DiGraph())
(4)                 >>> pred, dist = nx.bellman_ford_predecessor_and_distance(G, 0)
(4)                 >>> sorted(pred.items())
(4)                 [(0, []), (1, [0]), (2, [1]), (3, [2]), (4, [3])]
(4)                 >>> sorted(dist.items())
(4)                 [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
(4)                 >>> pred, dist = nx.bellman_ford_predecessor_and_distance(G, 0, 1)
(4)                 >>> sorted(pred.items())
(4)                 [(0, []), (1, [0]), (2, [1]), (3, [2]), (4, [3])]
(4)                 >>> sorted(dist.items())
(4)                 [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
(4)                 >>> G = nx.cycle_graph(5, create_using=nx.DiGraph())
(4)                 >>> G[1][2]["weight"] = -7
(4)                 >>> nx.bellman_ford_predecessor_and_distance(G, 0)
(4)                 Traceback (most recent call last):
(8)                     ...
(4)                 networkx.exception.NetworkXUnbounded: Negative cycle detected.
(4)                 See Also
(4)                 --------
(4)                 find_negative_cycle
(4)                 Notes
(4)                 -----
```

```
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    The dictionaries returned only have keys for nodes reachable from
    the source.
    In the case where the (di)graph is not connected, if a component
    not containing the source contains a negative (di)cycle, it
    will not be detected.
    In NetworkX v2.1 and prior, the source node had predecessor `[None]`.
    In NetworkX v2.2 this changed to the source node having predecessor `[]`
    """
    if source not in G:
        raise nx.NodeNotFound(f"Node {source} is not found in the graph")
    weight = _weight_function(G, weight)
    if G.is_multigraph():
        if any(
            weight(u, v, {k: d}) < 0
            for u, v, k, d in nx.selfloop_edges(G, keys=True, data=True)
        ):
            raise nx.NetworkXUnbounded("Negative cycle detected.")
    else:
        if any(weight(u, v, d) < 0 for u, v, d in nx.selfloop_edges(G, data=True)):
            raise nx.NetworkXUnbounded("Negative cycle detected.")
    dist = {source: 0}
    pred = {source: []}
    if len(G) == 1:
        return pred, dist
    weight = _weight_function(G, weight)
    dist = _bellman_ford(
        G, [source], weight, pred=pred, dist=dist, target=target, heuristic=heuristic
    )
    return (pred, dist)
def _bellman_ford(
    G,
    source,
    weight,
    pred=None,
    paths=None,
    dist=None,
    target=None,
    heuristic=True,
):
    """Calls relaxation loop for Bellman-Ford algorithm and builds paths
    This is an implementation of the SPFA variant.
    See https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm
    Parameters
    ----------
    G : NetworkX graph
    source: list
        List of source nodes. The shortest path from any of the source
        nodes will be found if multiple sources are provided.
    weight : function
        The weight of an edge is the value returned by the function. The
        function must accept exactly three positional arguments: the two
        endpoints of an edge and the dictionary of edge attributes for
        that edge. The function must return a number.
    pred: dict of lists, optional (default=None)
        dict to store a list of predecessors keyed by that node
        If None, predecessors are not stored
    paths: dict, optional (default=None)
        dict to store the path list from source to each node, keyed by node
        If None, paths are not stored
    dist: dict, optional (default=None)
        dict to store distance from source to the keyed node
        If None, returned dist dict contents default to 0 for every node in the
        source list
    target: node label, optional
```

```
(8)                 Ending node for path. Path lengths to other destinations may (and
(8)                 probably will) be incorrect.
(4)             heuristic : bool
(8)                 Determines whether to use a heuristic to early detect negative
(8)                 cycles at a hopefully negligible cost.
(4)             Returns
(4)             -------
(4)             dist : dict
(8)                 Returns a dict keyed by node to the distance from the source.
(8)                 Dicts for paths and pred are in the mutated input dicts by those
names.
(4)             Raises
(4)             ------
(4)             NodeNotFound
(8)                 If any of `source` is not in `G`.
(4)             NetworkXUnbounded
(8)                 If the (di)graph contains a negative (di)cycle, the
(8)                 algorithm raises an exception to indicate the presence of the
(8)                 negative (di)cycle.  Note: any negative weight edge in an
(8)                 undirected graph is a negative cycle
(4)             """
(4)             if pred is None:
(8)                 pred = {v: [] for v in source}
(4)             if dist is None:
(8)                 dist = {v: 0 for v in source}
(4)             negative_cycle_found = _inner_bellman_ford(
(8)                 G,
(8)                 source,
(8)                 weight,
(8)                 pred,
(8)                 dist,
(8)                 heuristic,
(4)             )
(4)             if negative_cycle_found is not None:
(8)                 raise nx.NetworkXUnbounded("Negative cycle detected.")
(4)             if paths is not None:
(8)                 sources = set(source)
(8)                 dsts = [target] if target is not None else pred
(8)                 for dst in dsts:
(12)                    gen = _build_paths_from_predecessors(sources, dst, pred)
(12)                    paths[dst] = next(gen)
(4)             return dist
(0)         def _inner_bellman_ford(
(4)             G,
(4)             sources,
(4)             weight,
(4)             pred,
(4)             dist=None,
(4)             heuristic=True,
(0)         ):
(4)             """Inner Relaxation loop for Bellman-Ford algorithm.
(4)             This is an implementation of the SPFA variant.
(4)             See https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source: list
(8)                 List of source nodes. The shortest path from any of the source
(8)                 nodes will be found if multiple sources are provided.
(4)             weight : function
(8)                 The weight of an edge is the value returned by the function. The
(8)                 function must accept exactly three positional arguments: the two
(8)                 endpoints of an edge and the dictionary of edge attributes for
(8)                 that edge. The function must return a number.
(4)             pred: dict of lists
(8)                 dict to store a list of predecessors keyed by that node
(4)             dist: dict, optional (default=None)
(8)                 dict to store distance from source to the keyed node
(8)                 If None, returned dist dict contents default to 0 for every node in
```

```
the
1129: (8)                   source list
1130: (4)               heuristic : bool
1131: (8)                   Determines whether to use a heuristic to early detect negative
1132: (8)                   cycles at a hopefully negligible cost.
1133: (4)               Returns
1134: (4)               -------
1135: (4)               node or None
1136: (8)                   Return a node `v` where processing discovered a negative cycle.
1137: (8)                   If no negative cycle found, return None.
1138: (4)               Raises
1139: (4)               ------
1140: (4)               NodeNotFound
1141: (8)                   If any of `source` is not in `G`.
1142: (4)               """
1143: (4)               for s in sources:
1144: (8)                   if s not in G:
1145: (12)                      raise nx.NodeNotFound(f"Source {s} not in G")
1146: (4)               if pred is None:
1147: (8)                   pred = {v: [] for v in sources}
1148: (4)               if dist is None:
1149: (8)                   dist = {v: 0 for v in sources}
1150: (4)               nonexistent_edge = (None, None)
1151: (4)               pred_edge = {v: None for v in sources}
1152: (4)               recent_update = {v: nonexistent_edge for v in sources}
1153: (4)               G_succ = G._adj  # For speed-up (and works for both directed and
undirected graphs)
1154: (4)               inf = float("inf")
1155: (4)               n = len(G)
1156: (4)               count = {}
1157: (4)               q = deque(sources)
1158: (4)               in_q = set(sources)
1159: (4)               while q:
1160: (8)                   u = q.popleft()
1161: (8)                   in_q.remove(u)
1162: (8)                   if all(pred_u not in in_q for pred_u in pred[u]):
1163: (12)                      dist_u = dist[u]
1164: (12)                      for v, e in G_succ[u].items():
1165: (16)                          dist_v = dist_u + weight(u, v, e)
1166: (16)                          if dist_v < dist.get(v, inf):
1167: (20)                              if heuristic:
1168: (24)                                  if v in recent_update[u]:
1169: (28)                                      pred[v].append(u)
1170: (28)                                      return v
1171: (24)                                  if v in pred_edge and pred_edge[v] == u:
1172: (28)                                      recent_update[v] = recent_update[u]
1173: (24)                                  else:
1174: (28)                                      recent_update[v] = (u, v)
1175: (20)                              if v not in in_q:
1176: (24)                                  q.append(v)
1177: (24)                                  in_q.add(v)
1178: (24)                                  count_v = count.get(v, 0) + 1
1179: (24)                                  if count_v == n:
1180: (28)                                      return v
1181: (24)                                  count[v] = count_v
1182: (20)                              dist[v] = dist_v
1183: (20)                              pred[v] = [u]
1184: (20)                              pred_edge[v] = u
1185: (16)                          elif dist.get(v) is not None and dist_v == dist.get(v):
1186: (20)                              pred[v].append(u)
1187: (4)               return None
1188: (0)           @nx._dispatchable(edge_attrs="weight")
1189: (0)           def bellman_ford_path(G, source, target, weight="weight"):
1190: (4)               """Returns the shortest path from source to target in a weighted graph G.
1191: (4)               Parameters
1192: (4)               ----------
1193: (4)               G : NetworkX graph
1194: (4)               source : node
1195: (8)                   Starting node
```

```
(4)                 target : node
(8)                     Ending node
(4)                 weight : string or function (default="weight")
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number.
(4)                 Returns
(4)                 -------
(4)                 path : list
(8)                     List of nodes in a shortest path.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(8)                     If `source` is not in `G`.
(4)                 NetworkXNoPath
(8)                     If no path exists between source and target.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> nx.bellman_ford_path(G, 0, 4)
(4)                 [0, 1, 2, 3, 4]
(4)                 Notes
(4)                 -----
(4)                 Edge weight attributes must be numerical.
(4)                 Distances are calculated as sums of weighted edges traversed.
(4)                 See Also
(4)                 --------
(4)                 dijkstra_path, bellman_ford_path_length
(4)                 """
(4)                 length, path = single_source_bellman_ford(G, source, target=target,
weight=weight)
(4)                 return path
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def bellman_ford_path_length(G, source, target, weight="weight"):
(4)                 """Returns the shortest path length from source to target
(4)                 in a weighted graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node label
(8)                     starting node for path
(4)                 target : node label
(8)                     ending node for path
(4)                 weight : string or function (default="weight")
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number.
(4)                 Returns
(4)                 -------
(4)                 length : number
(8)                     Shortest path length.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(8)                     If `source` is not in `G`.
```

```
(4)                 NetworkXNoPath
(8)                     If no path exists between source and target.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> nx.bellman_ford_path_length(G, 0, 4)
(4)                 4
(4)                 Notes
(4)                 -----
(4)                 Edge weight attributes must be numerical.
(4)                 Distances are calculated as sums of weighted edges traversed.
(4)                 See Also
(4)                 --------
(4)                 dijkstra_path_length, bellman_ford_path
(4)                 """
(4)                 if source == target:
(8)                     if source not in G:
(12)                        raise nx.NodeNotFound(f"Node {source} not found in graph")
(8)                     return 0
(4)                 weight = _weight_function(G, weight)
(4)                 length = _bellman_ford(G, [source], weight, target=target)
(4)                 try:
(8)                     return length[target]
(4)                 except KeyError as err:
(8)                     raise nx.NetworkXNoPath(f"node {target} not reachable from {source}")
from err
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def single_source_bellman_ford_path(G, source, weight="weight"):
(4)                 """Compute shortest path between source and all other reachable
(4)                 nodes for a weighted graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node
(8)                     Starting node for path.
(4)                 weight : string or function (default="weight")
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
(8)                     If this is a function, the weight of an edge is the value
(8)                     returned by the function. The function must accept exactly three
(8)                     positional arguments: the two endpoints of an edge and the
(8)                     dictionary of edge attributes for that edge. The function must
(8)                     return a number.
(4)                 Returns
(4)                 -------
(4)                 paths : dictionary
(8)                     Dictionary of shortest path lengths keyed by target.
(4)                 Raises
(4)                 ------
(4)                 NodeNotFound
(8)                     If `source` is not in `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> path = nx.single_source_bellman_ford_path(G, 0)
(4)                 >>> path[4]
(4)                 [0, 1, 2, 3, 4]
(4)                 Notes
(4)                 -----
(4)                 Edge weight attributes must be numerical.
(4)                 Distances are calculated as sums of weighted edges traversed.
(4)                 See Also
(4)                 --------
(4)                 single_source_dijkstra, single_source_bellman_ford
(4)                 """
(4)                 (length, path) = single_source_bellman_ford(G, source, weight=weight)
```

```
    return path
@nx._dispatchable(edge_attrs="weight")
def single_source_bellman_ford_path_length(G, source, weight="weight"):
    """Compute the shortest path length between source and all other
    reachable nodes for a weighted graph.
    Parameters
    ----------
    G : NetworkX graph
    source : node label
        Starting node for path
    weight : string or function (default="weight")
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly three
        positional arguments: the two endpoints of an edge and the
        dictionary of edge attributes for that edge. The function must
        return a number.
    Returns
    -------
    length : dictionary
        Dictionary of shortest path length keyed by target
    Raises
    ------
    NodeNotFound
        If `source` is not in `G`.
    Examples
    --------
    >>> G = nx.path_graph(5)
    >>> length = nx.single_source_bellman_ford_path_length(G, 0)
    >>> length[4]
    4
    >>> for node in [0, 1, 2, 3, 4]:
    ...     print(f"{node}: {length[node]}")
    0: 0
    1: 1
    2: 2
    3: 3
    4: 4
    Notes
    -----
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    See Also
    --------
    single_source_dijkstra, single_source_bellman_ford
    """
    weight = _weight_function(G, weight)
    return _bellman_ford(G, [source], weight)
@nx._dispatchable(edge_attrs="weight")
def single_source_bellman_ford(G, source, target=None, weight="weight"):
    """Compute shortest paths and lengths in a weighted graph G.
    Uses Bellman-Ford algorithm for shortest paths.
    Parameters
    ----------
    G : NetworkX graph
    source : node label
        Starting node for path
    target : node label, optional
        Ending node for path
    weight : string or function
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
```

```
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number.
(4)             Returns
(4)             -------
(4)             distance, path : pair of dictionaries, or numeric and list
(8)                 If target is None, returns a tuple of two dictionaries keyed by node.
(8)                 The first dictionary stores distance from one of the source nodes.
(8)                 The second stores the path from one of the sources to that node.
(8)                 If target is not None, returns a tuple of (distance, path) where
(8)                 distance is the distance from source to target and path is a list
(8)                 representing the path from source to target.
(4)             Raises
(4)             ------
(4)             NodeNotFound
(8)                 If `source` is not in `G`.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> length, path = nx.single_source_bellman_ford(G, 0)
(4)             >>> length[4]
(4)             4
(4)             >>> for node in [0, 1, 2, 3, 4]:
(4)             ...     print(f"{node}: {length[node]}")
(4)             0: 0
(4)             1: 1
(4)             2: 2
(4)             3: 3
(4)             4: 4
(4)             >>> path[4]
(4)             [0, 1, 2, 3, 4]
(4)             >>> length, path = nx.single_source_bellman_ford(G, 0, 1)
(4)             >>> length
(4)             1
(4)             >>> path
(4)             [0, 1]
(4)             Notes
(4)             -----
(4)             Edge weight attributes must be numerical.
(4)             Distances are calculated as sums of weighted edges traversed.
(4)             See Also
(4)             --------
(4)             single_source_dijkstra
(4)             single_source_bellman_ford_path
(4)             single_source_bellman_ford_path_length
(4)             """
(4)             if source == target:
(8)                 if source not in G:
(12)                    raise nx.NodeNotFound(f"Node {source} is not found in the graph")
(8)                 return (0, [source])
(4)             weight = _weight_function(G, weight)
(4)             paths = {source: [source]}  # dictionary of paths
(4)             dist = _bellman_ford(G, [source], weight, paths=paths, target=target)
(4)             if target is None:
(8)                 return (dist, paths)
(4)             try:
(8)                 return (dist[target], paths[target])
(4)             except KeyError as err:
(8)                 msg = f"Node {target} not reachable from {source}"
(8)                 raise nx.NetworkXNoPath(msg) from err
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def all_pairs_bellman_ford_path_length(G, weight="weight"):
(4)             """Compute shortest path lengths between all nodes in a weighted graph.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             weight : string or function (default="weight")
```

```
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number.
(4)             Returns
(4)             -------
(4)             distance : iterator
(8)                 (source, dictionary) iterator with dictionary keyed by target and
(8)                 shortest path length as the key value.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> length = dict(nx.all_pairs_bellman_ford_path_length(G))
(4)             >>> for node in [0, 1, 2, 3, 4]:
(4)             ...     print(f"1 - {node}: {length[1][node]}")
(4)             1 - 0: 1
(4)             1 - 1: 0
(4)             1 - 2: 1
(4)             1 - 3: 2
(4)             1 - 4: 3
(4)             >>> length[3][2]
(4)             1
(4)             >>> length[2][2]
(4)             0
(4)             Notes
(4)             -----
(4)             Edge weight attributes must be numerical.
(4)             Distances are calculated as sums of weighted edges traversed.
(4)             The dictionary returned only has keys for reachable node pairs.
(4)             """
(4)             length = single_source_bellman_ford_path_length
(4)             for n in G:
(8)                 yield (n, dict(length(G, n, weight=weight)))
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def all_pairs_bellman_ford_path(G, weight="weight"):
(4)             """Compute shortest paths between all nodes in a weighted graph.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             weight : string or function (default="weight")
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number.
(4)             Returns
(4)             -------
(4)             paths : iterator
(8)                 (source, dictionary) iterator with dictionary keyed by target and
(8)                 shortest path as the key value.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> path = dict(nx.all_pairs_bellman_ford_path(G))
(4)             >>> path[0][4]
(4)             [0, 1, 2, 3, 4]
(4)             Notes
(4)             -----
```

```
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    See Also
    --------
    floyd_warshall, all_pairs_dijkstra_path
    """
    path = single_source_bellman_ford_path
    for n in G:
        yield (n, path(G, n, weight=weight))
@nx._dispatchable(edge_attrs="weight")
def goldberg_radzik(G, source, weight="weight"):
    """Compute shortest path lengths and predecessors on shortest paths
    in weighted graphs.
    The algorithm has a running time of $O(mn)$ where $n$ is the number of
    nodes and $m$ is the number of edges.  It is slower than Dijkstra but
    can handle negative edge weights.
    Parameters
    ----------
    G : NetworkX graph
        The algorithm works for all types of graphs, including directed
        graphs and multigraphs.
    source: node label
        Starting node for path
    weight : string or function
        If this is a string, then edge weights will be accessed via the
        edge attribute with this key (that is, the weight of the edge
        joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
        such edge attribute exists, the weight of the edge is assumed to
        be one.
        If this is a function, the weight of an edge is the value
        returned by the function. The function must accept exactly three
        positional arguments: the two endpoints of an edge and the
        dictionary of edge attributes for that edge. The function must
        return a number.
    Returns
    -------
    pred, dist : dictionaries
        Returns two dictionaries keyed by node to predecessor in the
        path and to the distance from the source respectively.
    Raises
    ------
    NodeNotFound
        If `source` is not in `G`.
    NetworkXUnbounded
        If the (di)graph contains a negative (di)cycle, the
        algorithm raises an exception to indicate the presence of the
        negative (di)cycle.  Note: any negative weight edge in an
        undirected graph is a negative cycle.
        As of NetworkX v3.2, a zero weight cycle is no longer
        incorrectly reported as a negative weight cycle.
    Examples
    --------
    >>> G = nx.path_graph(5, create_using=nx.DiGraph())
    >>> pred, dist = nx.goldberg_radzik(G, 0)
    >>> sorted(pred.items())
    [(0, None), (1, 0), (2, 1), (3, 2), (4, 3)]
    >>> sorted(dist.items())
    [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
    >>> G = nx.cycle_graph(5, create_using=nx.DiGraph())
    >>> G[1][2]["weight"] = -7
    >>> nx.goldberg_radzik(G, 0)
    Traceback (most recent call last):
        ...
    networkx.exception.NetworkXUnbounded: Negative cycle detected.
    Notes
    -----
    Edge weight attributes must be numerical.
    Distances are calculated as sums of weighted edges traversed.
    The dictionaries returned only have keys for nodes reachable from
```

```
1608: (4)                the source.
1609: (4)                In the case where the (di)graph is not connected, if a component
1610: (4)                not containing the source contains a negative (di)cycle, it
1611: (4)                will not be detected.
1612: (4)                """
1613: (4)                if source not in G:
1614: (8)                    raise nx.NodeNotFound(f"Node {source} is not found in the graph")
1615: (4)                weight = _weight_function(G, weight)
1616: (4)                if G.is_multigraph():
1617: (8)                    if any(
1618: (12)                       weight(u, v, {k: d}) < 0
1619: (12)                       for u, v, k, d in nx.selfloop_edges(G, keys=True, data=True)
1620: (8)                    ):
1621: (12)                       raise nx.NetworkXUnbounded("Negative cycle detected.")
1622: (4)                else:
1623: (8)                    if any(weight(u, v, d) < 0 for u, v, d in nx.selfloop_edges(G,
data=True)):
1624: (12)                       raise nx.NetworkXUnbounded("Negative cycle detected.")
1625: (4)                if len(G) == 1:
1626: (8)                    return {source: None}, {source: 0}
1627: (4)                G_succ = G._adj  # For speed-up (and works for both directed and
undirected graphs)
1628: (4)                inf = float("inf")
1629: (4)                d = {u: inf for u in G}
1630: (4)                d[source] = 0
1631: (4)                pred = {source: None}
1632: (4)                def topo_sort(relabeled):
1633: (8)                    """Topologically sort nodes relabeled in the previous round and detect
1634: (8)                    negative cycles.
1635: (8)                    """
1636: (8)                    to_scan = []
1637: (8)                    neg_count = {}
1638: (8)                    for u in relabeled:
1639: (12)                       if u in neg_count:
1640: (16)                           continue
1641: (12)                       d_u = d[u]
1642: (12)                       if all(d_u + weight(u, v, e) >= d[v] for v, e in
G_succ[u].items()):
1643: (16)                           continue
1644: (12)                       stack = [(u, iter(G_succ[u].items()))]
1645: (12)                       in_stack = {u}
1646: (12)                       neg_count[u] = 0
1647: (12)                       while stack:
1648: (16)                           u, it = stack[-1]
1649: (16)                           try:
1650: (20)                               v, e = next(it)
1651: (16)                           except StopIteration:
1652: (20)                               to_scan.append(u)
1653: (20)                               stack.pop()
1654: (20)                               in_stack.remove(u)
1655: (20)                               continue
1656: (16)                           t = d[u] + weight(u, v, e)
1657: (16)                           d_v = d[v]
1658: (16)                           if t < d_v:
1659: (20)                               is_neg = t < d_v
1660: (20)                               d[v] = t
1661: (20)                               pred[v] = u
1662: (20)                               if v not in neg_count:
1663: (24)                                   neg_count[v] = neg_count[u] + int(is_neg)
1664: (24)                                   stack.append((v, iter(G_succ[v].items())))
1665: (24)                                   in_stack.add(v)
1666: (20)                               elif v in in_stack and neg_count[u] + int(is_neg) >
neg_count[v]:
1667: (24)                                   raise nx.NetworkXUnbounded("Negative cycle detected.")
1668: (8)                    to_scan.reverse()
1669: (8)                    return to_scan
1670: (4)                def relax(to_scan):
1671: (8)                    """Relax out-edges of relabeled nodes."""
1672: (8)                    relabeled = set()
```

```
(8)                 for u in to_scan:
(12)                    d_u = d[u]
(12)                    for v, e in G_succ[u].items():
(16)                        w_e = weight(u, v, e)
(16)                        if d_u + w_e < d[v]:
(20)                            d[v] = d_u + w_e
(20)                            pred[v] = u
(20)                            relabeled.add(v)
(8)                 return relabeled
(4)             relabeled = {source}
(4)             while relabeled:
(8)                 to_scan = topo_sort(relabeled)
(8)                 relabeled = relax(to_scan)
(4)             d = {u: d[u] for u in pred}
(4)             return pred, d
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def negative_edge_cycle(G, weight="weight", heuristic=True):
(4)             """Returns True if there exists a negative edge cycle anywhere in G.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             weight : string or function
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number.
(4)             heuristic : bool
(8)                 Determines whether to use a heuristic to early detect negative
(8)                 cycles at a negligible cost. In case of graphs with a negative cycle,
(8)                 the performance of detection increases by at least an order of
magnitude.
(4)             Returns
(4)             -------
(4)             negative_cycle : bool
(8)                 True if a negative edge cycle exists, otherwise False.
(4)             Examples
(4)             --------
(4)             >>> G = nx.cycle_graph(5, create_using=nx.DiGraph())
(4)             >>> print(nx.negative_edge_cycle(G))
(4)             False
(4)             >>> G[1][2]["weight"] = -7
(4)             >>> print(nx.negative_edge_cycle(G))
(4)             True
(4)             Notes
(4)             -----
(4)             Edge weight attributes must be numerical.
(4)             Distances are calculated as sums of weighted edges traversed.
(4)             This algorithm uses bellman_ford_predecessor_and_distance() but finds
(4)             negative cycles on any component by first adding a new node connected to
(4)             every node, and starting bellman_ford_predecessor_and_distance on that
(4)             node.  It then removes that extra node.
(4)             """
(4)             if G.size() == 0:
(8)                 return False
(4)             newnode = -1
(4)             while newnode in G:
(8)                 newnode -= 1
(4)             G.add_edges_from([(newnode, n) for n in G])
(4)             try:
(8)                 bellman_ford_predecessor_and_distance(
(12)                    G, newnode, weight=weight, heuristic=heuristic
(8)                 )
(4)             except nx.NetworkXUnbounded:
```

```
(8)                 return True
(4)             finally:
(8)                 G.remove_node(newnode)
(4)             return False
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def find_negative_cycle(G, source, weight="weight"):
(4)             """Returns a cycle with negative total weight if it exists.
(4)             Bellman-Ford is used to find shortest_paths. That algorithm
(4)             stops if there exists a negative cycle. This algorithm
(4)             picks up from there and returns the found negative cycle.
(4)             The cycle consists of a list of nodes in the cycle order. The last
(4)             node equals the first to make it a cycle.
(4)             You can look up the edge weights in the original graph. In the case
(4)             of multigraphs the relevant edge is the minimal weight edge between
(4)             the nodes in the 2-tuple.
(4)             If the graph has no negative cycle, a NetworkXError is raised.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source: node label
(8)                 The search for the negative cycle will start from this node.
(4)             weight : string or function
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number.
(4)             Examples
(4)             --------
(4)             >>> G = nx.DiGraph()
(4)             >>> G.add_weighted_edges_from([(0, 1, 2), (1, 2, 2), (2, 0, 1), (1, 4, 2),
(4, 0, -5)])
(4)             >>> nx.find_negative_cycle(G, 0)
(4)             [4, 0, 1, 4]
(4)             Returns
(4)             -------
(4)             cycle : list
(8)                 A list of nodes in the order of the cycle found. The last node
(8)                 equals the first to indicate a cycle.
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If no negative cycle is found.
(4)             """
(4)             weight = _weight_function(G, weight)
(4)             pred = {source: []}
(4)             v = _inner_bellman_ford(G, [source], weight, pred=pred)
(4)             if v is None:
(8)                 raise nx.NetworkXError("No negative cycles detected.")
(4)             neg_cycle = []
(4)             stack = [(v, list(pred[v]))]
(4)             seen = {v}
(4)             while stack:
(8)                 node, preds = stack[-1]
(8)                 if v in preds:
(12)                    neg_cycle.extend([node, v])
(12)                    neg_cycle = list(reversed(neg_cycle))
(12)                    return neg_cycle
(8)                 if preds:
(12)                    nbr = preds.pop()
(12)                    if nbr not in seen:
(16)                        stack.append((nbr, list(pred[nbr])))
(16)                        neg_cycle.append(node)
(16)                        seen.add(nbr)
```

```
(8)                 else:
(12)                    stack.pop()
(12)                    if neg_cycle:
(16)                        neg_cycle.pop()
(12)                    else:
(16)                        if v in G[v] and weight(G, v, v) < 0:
(20)                            return [v, v]
(16)                        raise nx.NetworkXError("Negative cycle is detected but not found")
(4)             msg = "negative cycle detected but not identified"
(4)             raise nx.NetworkXUnbounded(msg)
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def bidirectional_dijkstra(G, source, target, weight="weight"):
(4)             r"""Dijkstra's algorithm for shortest paths using bidirectional search.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source : node
(8)                 Starting node.
(4)             target : node
(8)                 Ending node.
(4)             weight : string or function
(8)                 If this is a string, then edge weights will be accessed via the
(8)                 edge attribute with this key (that is, the weight of the edge
(8)                 joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                 such edge attribute exists, the weight of the edge is assumed to
(8)                 be one.
(8)                 If this is a function, the weight of an edge is the value
(8)                 returned by the function. The function must accept exactly three
(8)                 positional arguments: the two endpoints of an edge and the
(8)                 dictionary of edge attributes for that edge. The function must
(8)                 return a number or None to indicate a hidden edge.
(4)             Returns
(4)             -------
(4)             length, path : number and list
(8)                 length is the distance from source to target.
(8)                 path is a list of nodes on a path from source to target.
(4)             Raises
(4)             ------
(4)             NodeNotFound
(8)                 If either `source` or `target` is not in `G`.
(4)             NetworkXNoPath
(8)                 If no path exists between source and target.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> length, path = nx.bidirectional_dijkstra(G, 0, 4)
(4)             >>> print(length)
(4)             4
(4)             >>> print(path)
(4)             [0, 1, 2, 3, 4]
(4)             Notes
(4)             -----
(4)             Edge weight attributes must be numerical.
(4)             Distances are calculated as sums of weighted edges traversed.
(4)             The weight function can be used to hide edges by returning None.
(4)             So ``weight = lambda u, v, d: 1 if d['color']=="red" else None``
(4)             will find the shortest red path.
(4)             In practice  bidirectional Dijkstra is much more than twice as fast as
(4)             ordinary Dijkstra.
(4)             Ordinary Dijkstra expands nodes in a sphere-like manner from the
(4)             source. The radius of this sphere will eventually be the length
(4)             of the shortest path. Bidirectional Dijkstra will expand nodes
(4)             from both the source and the target, making two spheres of half
(4)             this radius. Volume of the first sphere is `\pi*r*r` while the
(4)             others are `2*\pi*r/2*r/2`, making up half the volume.
(4)             This algorithm is not guaranteed to work if edge weights
(4)             are negative or are floating point numbers
(4)             (overflows and roundoff errors can cause problems).
```

```
(4)                 See Also
(4)                 --------
(4)                 shortest_path
(4)                 shortest_path_length
(4)                 """
(4)                 if source not in G or target not in G:
(8)                     msg = f"Either source {source} or target {target} is not in G"
(8)                     raise nx.NodeNotFound(msg)
(4)                 if source == target:
(8)                     return (0, [source])
(4)                 weight = _weight_function(G, weight)
(4)                 push = heappush
(4)                 pop = heappop
(4)                 dists = [{}, {}]  # dictionary of final distances
(4)                 paths = [{source: [source]}, {target: [target]}]  # dictionary of paths
(4)                 fringe = [[], []]  # heap of (distance, node) for choosing node to expand
(4)                 seen = [{source: 0}, {target: 0}]  # dict of distances to seen nodes
(4)                 c = count()
(4)                 push(fringe[0], (0, next(c), source))
(4)                 push(fringe[1], (0, next(c), target))
(4)                 if G.is_directed():
(8)                     neighs = [G._succ, G._pred]
(4)                 else:
(8)                     neighs = [G._adj, G._adj]
(4)                 finalpath = []
(4)                 dir = 1
(4)                 while fringe[0] and fringe[1]:
(8)                     dir = 1 - dir
(8)                     (dist, _, v) = pop(fringe[dir])
(8)                     if v in dists[dir]:
(12)                        continue
(8)                     dists[dir][v] = dist  # equal to seen[dir][v]
(8)                     if v in dists[1 - dir]:
(12)                        return (finaldist, finalpath)
(8)                     for w, d in neighs[dir][v].items():
(12)                        cost = weight(v, w, d) if dir == 0 else weight(w, v, d)
(12)                        if cost is None:
(16)                            continue
(12)                        vwLength = dists[dir][v] + cost
(12)                        if w in dists[dir]:
(16)                            if vwLength < dists[dir][w]:
(20)                                raise ValueError("Contradictory paths found: negative
weights?")
(12)                        elif w not in seen[dir] or vwLength < seen[dir][w]:
(16)                            seen[dir][w] = vwLength
(16)                            push(fringe[dir], (vwLength, next(c), w))
(16)                            paths[dir][w] = paths[dir][v] + [w]
(16)                            if w in seen[0] and w in seen[1]:
(20)                                totaldist = seen[0][w] + seen[1][w]
(20)                                if finalpath == [] or finaldist > totaldist:
(24)                                    finaldist = totaldist
(24)                                    revpath = paths[1][w][:]
(24)                                    revpath.reverse()
(24)                                    finalpath = paths[0][w] + revpath[1:]
(4)                 raise nx.NetworkXNoPath(f"No path between {source} and {target}.")
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def johnson(G, weight="weight"):
(4)                 r"""Uses Johnson's Algorithm to compute shortest paths.
(4)                 Johnson's Algorithm finds a shortest path between each pair of
(4)                 nodes in a weighted graph even if negative weights are present.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 weight : string or function
(8)                     If this is a string, then edge weights will be accessed via the
(8)                     edge attribute with this key (that is, the weight of the edge
(8)                     joining `u` to `v` will be ``G.edges[u, v][weight]``). If no
(8)                     such edge attribute exists, the weight of the edge is assumed to
(8)                     be one.
```

```
1945: (8)                   If this is a function, the weight of an edge is the value
1946: (8)                   returned by the function. The function must accept exactly three
1947: (8)                   positional arguments: the two endpoints of an edge and the
1948: (8)                   dictionary of edge attributes for that edge. The function must
1949: (8)                   return a number.
1950: (4)               Returns
1951: (4)               -------
1952: (4)               distance : dictionary
1953: (8)                   Dictionary, keyed by source and target, of shortest paths.
1954: (4)               Examples
1955: (4)               --------
1956: (4)               >>> graph = nx.DiGraph()
1957: (4)               >>> graph.add_weighted_edges_from(
1958: (4)               ...     [("0", "3", 3), ("0", "1", -5), ("0", "2", 2), ("1", "2", 4),
("2", "3", 1)]
1959: (4)               ... )
1960: (4)               >>> paths = nx.johnson(graph, weight="weight")
1961: (4)               >>> paths["0"]["2"]
1962: (4)               ['0', '1', '2']
1963: (4)               Notes
1964: (4)               -----
1965: (4)               Johnson's algorithm is suitable even for graphs with negative weights. It
1966: (4)               works by using the Bellman-Ford algorithm to compute a transformation of
1967: (4)               the input graph that removes all negative weights, allowing Dijkstra's
1968: (4)               algorithm to be used on the transformed graph.
1969: (4)               The time complexity of this algorithm is $O(n^2 \log n + n m)$,
1970: (4)               where $n$ is the number of nodes and $m$ the number of edges in the
1971: (4)               graph. For dense graphs, this may be faster than the Floyd-Warshall
1972: (4)               algorithm.
1973: (4)               See Also
1974: (4)               --------
1975: (4)               floyd_warshall_predecessor_and_distance
1976: (4)               floyd_warshall_numpy
1977: (4)               all_pairs_shortest_path
1978: (4)               all_pairs_shortest_path_length
1979: (4)               all_pairs_dijkstra_path
1980: (4)               bellman_ford_predecessor_and_distance
1981: (4)               all_pairs_bellman_ford_path
1982: (4)               all_pairs_bellman_ford_path_length
1983: (4)               """
1984: (4)               dist = {v: 0 for v in G}
1985: (4)               pred = {v: [] for v in G}
1986: (4)               weight = _weight_function(G, weight)
1987: (4)               dist_bellman = _bellman_ford(G, list(G), weight, pred=pred, dist=dist)
1988: (4)               def new_weight(u, v, d):
1989: (8)                   return weight(u, v, d) + dist_bellman[u] - dist_bellman[v]
1990: (4)               def dist_path(v):
1991: (8)                   paths = {v: [v]}
1992: (8)                   _dijkstra(G, v, new_weight, paths=paths)
1993: (8)                   return paths
1994: (4)               return {v: dist_path(v) for v in G}

----------------------------------------

File 298 - __init__.py:

1: (0)

----------------------------------------

File 299 - test_astar.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import pairwise
4: (0)              class TestAStar:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      edges = [
```

```
 8: (12)                        ("s", "u", 10),
 9: (12)                        ("s", "x", 5),
 10: (12)                       ("u", "v", 1),
 11: (12)                       ("u", "x", 2),
 12: (12)                       ("v", "y", 1),
 13: (12)                       ("x", "u", 3),
 14: (12)                       ("x", "v", 5),
 15: (12)                       ("x", "y", 2),
 16: (12)                       ("y", "s", 7),
 17: (12)                       ("y", "v", 6),
 18: (8)                    ]
 19: (8)                    cls.XG = nx.DiGraph()
 20: (8)                    cls.XG.add_weighted_edges_from(edges)
 21: (4)                def test_multiple_optimal_paths(self):
 22: (8)                    """Tests that A* algorithm finds any of multiple optimal paths"""
 23: (8)                    heuristic_values = {"a": 1.35, "b": 1.18, "c": 0.67, "d": 0}
 24: (8)                    def h(u, v):
 25: (12)                       return heuristic_values[u]
 26: (8)                    graph = nx.Graph()
 27: (8)                    points = ["a", "b", "c", "d"]
 28: (8)                    edges = [("a", "b", 0.18), ("a", "c", 0.68), ("b", "c", 0.50), ("c",
"d", 0.67)]
 29: (8)                    graph.add_nodes_from(points)
 30: (8)                    graph.add_weighted_edges_from(edges)
 31: (8)                    path1 = ["a", "c", "d"]
 32: (8)                    path2 = ["a", "b", "c", "d"]
 33: (8)                    assert nx.astar_path(graph, "a", "d", h) in (path1, path2)
 34: (4)                def test_astar_directed(self):
 35: (8)                    assert nx.astar_path(self.XG, "s", "v") == ["s", "x", "u", "v"]
 36: (8)                    assert nx.astar_path_length(self.XG, "s", "v") == 9
 37: (4)                def test_astar_directed_weight_function(self):
 38: (8)                    w1 = lambda u, v, d: d["weight"]
 39: (8)                    assert nx.astar_path(self.XG, "x", "u", weight=w1) == ["x", "u"]
 40: (8)                    assert nx.astar_path_length(self.XG, "x", "u", weight=w1) == 3
 41: (8)                    assert nx.astar_path(self.XG, "s", "v", weight=w1) == ["s", "x", "u",
"v"]
 42: (8)                    assert nx.astar_path_length(self.XG, "s", "v", weight=w1) == 9
 43: (8)                    w2 = lambda u, v, d: None if (u, v) == ("x", "u") else d["weight"]
 44: (8)                    assert nx.astar_path(self.XG, "x", "u", weight=w2) == ["x", "y", "s",
"u"]
 45: (8)                    assert nx.astar_path_length(self.XG, "x", "u", weight=w2) == 19
 46: (8)                    assert nx.astar_path(self.XG, "s", "v", weight=w2) == ["s", "x", "v"]
 47: (8)                    assert nx.astar_path_length(self.XG, "s", "v", weight=w2) == 10
 48: (8)                    w3 = lambda u, v, d: d["weight"] + 10
 49: (8)                    assert nx.astar_path(self.XG, "x", "u", weight=w3) == ["x", "u"]
 50: (8)                    assert nx.astar_path_length(self.XG, "x", "u", weight=w3) == 13
 51: (8)                    assert nx.astar_path(self.XG, "s", "v", weight=w3) == ["s", "x", "v"]
 52: (8)                    assert nx.astar_path_length(self.XG, "s", "v", weight=w3) == 30
 53: (4)                def test_astar_multigraph(self):
 54: (8)                    G = nx.MultiDiGraph(self.XG)
 55: (8)                    G.add_weighted_edges_from((u, v, 1000) for (u, v) in list(G.edges()))
 56: (8)                    assert nx.astar_path(G, "s", "v") == ["s", "x", "u", "v"]
 57: (8)                    assert nx.astar_path_length(G, "s", "v") == 9
 58: (4)                def test_astar_undirected(self):
 59: (8)                    GG = self.XG.to_undirected()
 60: (8)                    GG["u"]["x"]["weight"] = 2
 61: (8)                    GG["y"]["v"]["weight"] = 2
 62: (8)                    assert nx.astar_path(GG, "s", "v") == ["s", "x", "u", "v"]
 63: (8)                    assert nx.astar_path_length(GG, "s", "v") == 8
 64: (4)                def test_astar_directed2(self):
 65: (8)                    XG2 = nx.DiGraph()
 66: (8)                    edges = [
 67: (12)                       (1, 4, 1),
 68: (12)                       (4, 5, 1),
 69: (12)                       (5, 6, 1),
 70: (12)                       (6, 3, 1),
 71: (12)                       (1, 3, 50),
 72: (12)                       (1, 2, 100),
 73: (12)                       (2, 3, 100),
```

```
74: (8)                     ]
75: (8)                     XG2.add_weighted_edges_from(edges)
76: (8)                     assert nx.astar_path(XG2, 1, 3) == [1, 4, 5, 6, 3]
77: (4)                 def test_astar_undirected2(self):
78: (8)                     XG3 = nx.Graph()
79: (8)                     edges = [(0, 1, 2), (1, 2, 12), (2, 3, 1), (3, 4, 5), (4, 5, 1), (5,
0, 10)]
80: (8)                     XG3.add_weighted_edges_from(edges)
81: (8)                     assert nx.astar_path(XG3, 0, 3) == [0, 1, 2, 3]
82: (8)                     assert nx.astar_path_length(XG3, 0, 3) == 15
83: (4)                 def test_astar_undirected3(self):
84: (8)                     XG4 = nx.Graph()
85: (8)                     edges = [
86: (12)                        (0, 1, 2),
87: (12)                        (1, 2, 2),
88: (12)                        (2, 3, 1),
89: (12)                        (3, 4, 1),
90: (12)                        (4, 5, 1),
91: (12)                        (5, 6, 1),
92: (12)                        (6, 7, 1),
93: (12)                        (7, 0, 1),
94: (8)                     ]
95: (8)                     XG4.add_weighted_edges_from(edges)
96: (8)                     assert nx.astar_path(XG4, 0, 2) == [0, 1, 2]
97: (8)                     assert nx.astar_path_length(XG4, 0, 2) == 4
98: (4)                 """ Tests that A* finds correct path when multiple paths exist
99: (8)                     and the best one is not expanded first (GH issue #3464)
100: (4)                """
101: (4)                def test_astar_directed3(self):
102: (8)                    heuristic_values = {"n5": 36, "n2": 4, "n1": 0, "n0": 0}
103: (8)                    def h(u, v):
104: (12)                       return heuristic_values[u]
105: (8)                    edges = [("n5", "n1", 11), ("n5", "n2", 9), ("n2", "n1", 1), ("n1",
"n0", 32)]
106: (8)                    graph = nx.DiGraph()
107: (8)                    graph.add_weighted_edges_from(edges)
108: (8)                    answer = ["n5", "n2", "n1", "n0"]
109: (8)                    assert nx.astar_path(graph, "n5", "n0", h) == answer
110: (4)                """ Tests that parent is not wrongly overridden when a node
111: (8)                    is re-explored multiple times.
112: (4)                """
113: (4)                def test_astar_directed4(self):
114: (8)                    edges = [
115: (12)                       ("a", "b", 1),
116: (12)                       ("a", "c", 1),
117: (12)                       ("b", "d", 2),
118: (12)                       ("c", "d", 1),
119: (12)                       ("d", "e", 1),
120: (8)                    ]
121: (8)                    graph = nx.DiGraph()
122: (8)                    graph.add_weighted_edges_from(edges)
123: (8)                    assert nx.astar_path(graph, "a", "e") == ["a", "c", "d", "e"]
124: (4)                def test_astar_w1(self):
125: (8)                    G = nx.DiGraph()
126: (8)                    G.add_edges_from(
127: (12)                       [
128: (16)                           ("s", "u"),
129: (16)                           ("s", "x"),
130: (16)                           ("u", "v"),
131: (16)                           ("u", "x"),
132: (16)                           ("v", "y"),
133: (16)                           ("x", "u"),
134: (16)                           ("x", "w"),
135: (16)                           ("w", "v"),
136: (16)                           ("x", "y"),
137: (16)                           ("y", "s"),
138: (16)                           ("y", "v"),
139: (12)                       ]
140: (8)                    )
```

```
141: (8)                    assert nx.astar_path(G, "s", "v") == ["s", "u", "v"]
142: (8)                    assert nx.astar_path_length(G, "s", "v") == 2
143: (4)                def test_astar_nopath(self):
144: (8)                    with pytest.raises(nx.NodeNotFound):
145: (12)                       nx.astar_path(self.XG, "s", "moon")
146: (4)                def test_astar_cutoff(self):
147: (8)                    with pytest.raises(nx.NetworkXNoPath):
148: (12)                       nx.astar_path(self.XG, "s", "v", cutoff=8.0)
149: (8)                    with pytest.raises(nx.NetworkXNoPath):
150: (12)                       nx.astar_path_length(self.XG, "s", "v", cutoff=8.0)
151: (4)                def test_astar_admissible_heuristic_with_cutoff(self):
152: (8)                    heuristic_values = {"s": 36, "y": 4, "x": 0, "u": 0, "v": 0}
153: (8)                    def h(u, v):
154: (12)                       return heuristic_values[u]
155: (8)                    assert nx.astar_path_length(self.XG, "s", "v") == 9
156: (8)                    assert nx.astar_path_length(self.XG, "s", "v", heuristic=h) == 9
157: (8)                    assert nx.astar_path_length(self.XG, "s", "v", heuristic=h, cutoff=12)
== 9
158: (8)                    assert nx.astar_path_length(self.XG, "s", "v", heuristic=h, cutoff=9)
== 9
159: (8)                    with pytest.raises(nx.NetworkXNoPath):
160: (12)                       nx.astar_path_length(self.XG, "s", "v", heuristic=h, cutoff=8)
161: (4)                def test_astar_inadmissible_heuristic_with_cutoff(self):
162: (8)                    heuristic_values = {"s": 36, "y": 14, "x": 10, "u": 10, "v": 0}
163: (8)                    def h(u, v):
164: (12)                       return heuristic_values[u]
165: (8)                    assert nx.astar_path_length(self.XG, "s", "v", heuristic=h) == 10
166: (8)                    assert nx.astar_path_length(self.XG, "s", "v", heuristic=h, cutoff=15)
== 10
167: (8)                    with pytest.raises(nx.NetworkXNoPath):
168: (12)                       nx.astar_path_length(self.XG, "s", "v", heuristic=h, cutoff=9)
169: (8)                    with pytest.raises(nx.NetworkXNoPath):
170: (12)                       nx.astar_path_length(self.XG, "s", "v", heuristic=h, cutoff=12)
171: (4)                def test_astar_cutoff2(self):
172: (8)                    assert nx.astar_path(self.XG, "s", "v", cutoff=10.0) == ["s", "x",
"u", "v"]
173: (8)                    assert nx.astar_path_length(self.XG, "s", "v") == 9
174: (4)                def test_cycle(self):
175: (8)                    C = nx.cycle_graph(7)
176: (8)                    assert nx.astar_path(C, 0, 3) == [0, 1, 2, 3]
177: (8)                    assert nx.dijkstra_path(C, 0, 4) == [0, 6, 5, 4]
178: (4)                def test_unorderable_nodes(self):
179: (8)                    """Tests that A* accommodates nodes that are not orderable.
180: (8)                    For more information, see issue #554.
181: (8)                    """
182: (8)                    nodes = [object() for n in range(4)]
183: (8)                    G = nx.Graph()
184: (8)                    G.add_edges_from(pairwise(nodes, cyclic=True))
185: (8)                    path = nx.astar_path(G, nodes[0], nodes[2])
186: (8)                    assert len(path) == 3
187: (4)                def test_astar_NetworkXNoPath(self):
188: (8)                    """Tests that exception is raised when there exists no
189: (8)                    path between source and target"""
190: (8)                    G = nx.gnp_random_graph(10, 0.2, seed=10)
191: (8)                    with pytest.raises(nx.NetworkXNoPath):
192: (12)                       nx.astar_path(G, 4, 9)
193: (4)                def test_astar_NodeNotFound(self):
194: (8)                    """Tests that exception is raised when either
195: (8)                    source or target is not in graph"""
196: (8)                    G = nx.gnp_random_graph(10, 0.2, seed=10)
197: (8)                    with pytest.raises(nx.NodeNotFound):
198: (12)                       nx.astar_path_length(G, 11, 9)

----------------------------------------

File 300 - test_dense.py:

1: (0)              import pytest
2: (0)              import networkx as nx
```

```
(0)  class TestFloyd:
(4)      @classmethod
(4)      def setup_class(cls):
(8)          pass
(4)      def test_floyd_warshall_predecessor_and_distance(self):
(8)          XG = nx.DiGraph()
(8)          XG.add_weighted_edges_from(
(12)             [
(16)                 ("s", "u", 10),
(16)                 ("s", "x", 5),
(16)                 ("u", "v", 1),
(16)                 ("u", "x", 2),
(16)                 ("v", "y", 1),
(16)                 ("x", "u", 3),
(16)                 ("x", "v", 5),
(16)                 ("x", "y", 2),
(16)                 ("y", "s", 7),
(16)                 ("y", "v", 6),
(12)             ]
(8)          )
(8)          path, dist = nx.floyd_warshall_predecessor_and_distance(XG)
(8)          assert dist["s"]["v"] == 9
(8)          assert path["s"]["v"] == "u"
(8)          assert dist == {
(12)             "y": {"y": 0, "x": 12, "s": 7, "u": 15, "v": 6},
(12)             "x": {"y": 2, "x": 0, "s": 9, "u": 3, "v": 4},
(12)             "s": {"y": 7, "x": 5, "s": 0, "u": 8, "v": 9},
(12)             "u": {"y": 2, "x": 2, "s": 9, "u": 0, "v": 1},
(12)             "v": {"y": 1, "x": 13, "s": 8, "u": 16, "v": 0},
(8)          }
(8)          GG = XG.to_undirected()
(8)          GG["u"]["x"]["weight"] = 2
(8)          path, dist = nx.floyd_warshall_predecessor_and_distance(GG)
(8)          assert dist["s"]["v"] == 8
(8)          G = nx.DiGraph()  # no weights
(8)          G.add_edges_from(
(12)             [
(16)                 ("s", "u"),
(16)                 ("s", "x"),
(16)                 ("u", "v"),
(16)                 ("u", "x"),
(16)                 ("v", "y"),
(16)                 ("x", "u"),
(16)                 ("x", "v"),
(16)                 ("x", "y"),
(16)                 ("y", "s"),
(16)                 ("y", "v"),
(12)             ]
(8)          )
(8)          path, dist = nx.floyd_warshall_predecessor_and_distance(G)
(8)          assert dist["s"]["v"] == 2
(8)          dist = nx.floyd_warshall(G)
(8)          assert dist["s"]["v"] == 2
(8)          XG = nx.DiGraph()
(8)          XG.add_weighted_edges_from(
(12)             [("v", "x", 5.0), ("y", "x", 5.0), ("v", "y", 6.0), ("x", "u",
2.0)]
(8)          )
(8)          path, dist = nx.floyd_warshall_predecessor_and_distance(XG)
(8)          inf = float("inf")
(8)          assert dist == {
(12)             "v": {"v": 0, "x": 5.0, "y": 6.0, "u": 7.0},
(12)             "x": {"x": 0, "u": 2.0, "v": inf, "y": inf},
(12)             "y": {"y": 0, "x": 5.0, "v": inf, "u": 7.0},
(12)             "u": {"u": 0, "v": inf, "x": inf, "y": inf},
(8)          }
(8)          assert path == {
(12)             "v": {"x": "v", "y": "v", "u": "x"},
(12)             "x": {"u": "x"},
```

```
 71: (12)                        "y": {"x": "y", "u": "x"},
 72: (8)                     }
 73: (4)                 def test_reconstruct_path(self):
 74: (8)                     with pytest.raises(KeyError):
 75: (12)                        XG = nx.DiGraph()
 76: (12)                        XG.add_weighted_edges_from(
 77: (16)                            [
 78: (20)                                ("s", "u", 10),
 79: (20)                                ("s", "x", 5),
 80: (20)                                ("u", "v", 1),
 81: (20)                                ("u", "x", 2),
 82: (20)                                ("v", "y", 1),
 83: (20)                                ("x", "u", 3),
 84: (20)                                ("x", "v", 5),
 85: (20)                                ("x", "y", 2),
 86: (20)                                ("y", "s", 7),
 87: (20)                                ("y", "v", 6),
 88: (16)                            ]
 89: (12)                        )
 90: (12)                        predecessors, _ = nx.floyd_warshall_predecessor_and_distance(XG)
 91: (12)                        path = nx.reconstruct_path("s", "v", predecessors)
 92: (12)                        assert path == ["s", "x", "u", "v"]
 93: (12)                        path = nx.reconstruct_path("s", "s", predecessors)
 94: (12)                        assert path == []
 95: (12)                        nx.reconstruct_path("1", "2", predecessors)
 96: (4)                 def test_cycle(self):
 97: (8)                     path, dist =
nx.floyd_warshall_predecessor_and_distance(nx.cycle_graph(7))
 98: (8)                     assert dist[0][3] == 3
 99: (8)                     assert path[0][3] == 2
 100: (8)                    assert dist[0][4] == 3
 101: (4)                def test_weighted(self):
 102: (8)                    XG3 = nx.Graph()
 103: (8)                    XG3.add_weighted_edges_from(
 104: (12)                       [[0, 1, 2], [1, 2, 12], [2, 3, 1], [3, 4, 5], [4, 5, 1], [5, 0,
10]]
 105: (8)                    )
 106: (8)                    path, dist = nx.floyd_warshall_predecessor_and_distance(XG3)
 107: (8)                    assert dist[0][3] == 15
 108: (8)                    assert path[0][3] == 2
 109: (4)                def test_weighted2(self):
 110: (8)                    XG4 = nx.Graph()
 111: (8)                    XG4.add_weighted_edges_from(
 112: (12)                       [
 113: (16)                           [0, 1, 2],
 114: (16)                           [1, 2, 2],
 115: (16)                           [2, 3, 1],
 116: (16)                           [3, 4, 1],
 117: (16)                           [4, 5, 1],
 118: (16)                           [5, 6, 1],
 119: (16)                           [6, 7, 1],
 120: (16)                           [7, 0, 1],
 121: (12)                       ]
 122: (8)                    )
 123: (8)                    path, dist = nx.floyd_warshall_predecessor_and_distance(XG4)
 124: (8)                    assert dist[0][2] == 4
 125: (8)                    assert path[0][2] == 1
 126: (4)                def test_weight_parameter(self):
 127: (8)                    XG4 = nx.Graph()
 128: (8)                    XG4.add_edges_from(
 129: (12)                       [
 130: (16)                           (0, 1, {"heavy": 2}),
 131: (16)                           (1, 2, {"heavy": 2}),
 132: (16)                           (2, 3, {"heavy": 1}),
 133: (16)                           (3, 4, {"heavy": 1}),
 134: (16)                           (4, 5, {"heavy": 1}),
 135: (16)                           (5, 6, {"heavy": 1}),
 136: (16)                           (6, 7, {"heavy": 1}),
 137: (16)                           (7, 0, {"heavy": 1}),
```

```
138: (12)                        ]
139: (8)                    )
140: (8)                    path, dist = nx.floyd_warshall_predecessor_and_distance(XG4,
weight="heavy")
141: (8)                    assert dist[0][2] == 4
142: (8)                    assert path[0][2] == 1
143: (4)                def test_zero_distance(self):
144: (8)                    XG = nx.DiGraph()
145: (8)                    XG.add_weighted_edges_from(
146: (12)                        [
147: (16)                            ("s", "u", 10),
148: (16)                            ("s", "x", 5),
149: (16)                            ("u", "v", 1),
150: (16)                            ("u", "x", 2),
151: (16)                            ("v", "y", 1),
152: (16)                            ("x", "u", 3),
153: (16)                            ("x", "v", 5),
154: (16)                            ("x", "y", 2),
155: (16)                            ("y", "s", 7),
156: (16)                            ("y", "v", 6),
157: (12)                        ]
158: (8)                    )
159: (8)                    path, dist = nx.floyd_warshall_predecessor_and_distance(XG)
160: (8)                    for u in XG:
161: (12)                        assert dist[u][u] == 0
162: (8)                    GG = XG.to_undirected()
163: (8)                    GG["u"]["x"]["weight"] = 2
164: (8)                    path, dist = nx.floyd_warshall_predecessor_and_distance(GG)
165: (8)                    for u in GG:
166: (12)                        dist[u][u] = 0
167: (4)                def test_zero_weight(self):
168: (8)                    G = nx.DiGraph()
169: (8)                    edges = [(1, 2, -2), (2, 3, -4), (1, 5, 1), (5, 4, 0), (4, 3, -5), (2,
5, -7)]
170: (8)                    G.add_weighted_edges_from(edges)
171: (8)                    dist = nx.floyd_warshall(G)
172: (8)                    assert dist[1][3] == -14
173: (8)                    G = nx.MultiDiGraph()
174: (8)                    edges.append((2, 5, -7))
175: (8)                    G.add_weighted_edges_from(edges)
176: (8)                    dist = nx.floyd_warshall(G)
177: (8)                    assert dist[1][3] == -14

----------------------------------------

File 301 - test_dense_numpy.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              import networkx as nx
4: (0)              def test_cycle_numpy():
5: (4)                  dist = nx.floyd_warshall_numpy(nx.cycle_graph(7))
6: (4)                  assert dist[0, 3] == 3
7: (4)                  assert dist[0, 4] == 3
8: (0)              def test_weighted_numpy_three_edges():
9: (4)                  XG3 = nx.Graph()
10: (4)                 XG3.add_weighted_edges_from(
11: (8)                     [[0, 1, 2], [1, 2, 12], [2, 3, 1], [3, 4, 5], [4, 5, 1], [5, 0, 10]]
12: (4)                 )
13: (4)                 dist = nx.floyd_warshall_numpy(XG3)
14: (4)                 assert dist[0, 3] == 15
15: (0)             def test_weighted_numpy_two_edges():
16: (4)                 XG4 = nx.Graph()
17: (4)                 XG4.add_weighted_edges_from(
18: (8)                     [
19: (12)                        [0, 1, 2],
20: (12)                        [1, 2, 2],
21: (12)                        [2, 3, 1],
22: (12)                        [3, 4, 1],
```

```
23: (12)                        [4, 5, 1],
24: (12)                        [5, 6, 1],
25: (12)                        [6, 7, 1],
26: (12)                        [7, 0, 1],
27: (8)                     ]
28: (4)                 )
29: (4)                 dist = nx.floyd_warshall_numpy(XG4)
30: (4)                 assert dist[0, 2] == 4
31: (0)             def test_weight_parameter_numpy():
32: (4)                 XG4 = nx.Graph()
33: (4)                 XG4.add_edges_from(
34: (8)                     [
35: (12)                        (0, 1, {"heavy": 2}),
36: (12)                        (1, 2, {"heavy": 2}),
37: (12)                        (2, 3, {"heavy": 1}),
38: (12)                        (3, 4, {"heavy": 1}),
39: (12)                        (4, 5, {"heavy": 1}),
40: (12)                        (5, 6, {"heavy": 1}),
41: (12)                        (6, 7, {"heavy": 1}),
42: (12)                        (7, 0, {"heavy": 1}),
43: (8)                     ]
44: (4)                 )
45: (4)                 dist = nx.floyd_warshall_numpy(XG4, weight="heavy")
46: (4)                 assert dist[0, 2] == 4
47: (0)             def test_directed_cycle_numpy():
48: (4)                 G = nx.DiGraph()
49: (4)                 nx.add_cycle(G, [0, 1, 2, 3])
50: (4)                 pred, dist = nx.floyd_warshall_predecessor_and_distance(G)
51: (4)                 D = nx.utils.dict_to_numpy_array(dist)
52: (4)                 np.testing.assert_equal(nx.floyd_warshall_numpy(G), D)
53: (0)             def test_zero_weight():
54: (4)                 G = nx.DiGraph()
55: (4)                 edges = [(1, 2, -2), (2, 3, -4), (1, 5, 1), (5, 4, 0), (4, 3, -5), (2, 5,
-7)]
56: (4)                 G.add_weighted_edges_from(edges)
57: (4)                 dist = nx.floyd_warshall_numpy(G)
58: (4)                 assert int(np.min(dist)) == -14
59: (4)                 G = nx.MultiDiGraph()
60: (4)                 edges.append((2, 5, -7))
61: (4)                 G.add_weighted_edges_from(edges)
62: (4)                 dist = nx.floyd_warshall_numpy(G)
63: (4)                 assert int(np.min(dist)) == -14
64: (0)             def test_nodelist():
65: (4)                 G = nx.path_graph(7)
66: (4)                 dist = nx.floyd_warshall_numpy(G, nodelist=[3, 5, 4, 6, 2, 1, 0])
67: (4)                 assert dist[0, 3] == 3
68: (4)                 assert dist[0, 1] == 2
69: (4)                 assert dist[6, 2] == 4
70: (4)                 pytest.raises(nx.NetworkXError, nx.floyd_warshall_numpy, G, [1, 3])
71: (4)                 pytest.raises(nx.NetworkXError, nx.floyd_warshall_numpy, G,
list(range(9)))

----------------------------------------

File 302 - test_generic.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def validate_grid_path(r, c, s, t, p):
4: (4)                  assert isinstance(p, list)
5: (4)                  assert p[0] == s
6: (4)                  assert p[-1] == t
7: (4)                  s = ((s - 1) // c, (s - 1) % c)
8: (4)                  t = ((t - 1) // c, (t - 1) % c)
9: (4)                  assert len(p) == abs(t[0] - s[0]) + abs(t[1] - s[1]) + 1
10: (4)                 p = [((u - 1) // c, (u - 1) % c) for u in p]
11: (4)                 for u in p:
12: (8)                     assert 0 <= u[0] < r
13: (8)                     assert 0 <= u[1] < c
```

```
14: (4)                 for u, v in zip(p[:-1], p[1:]):
15: (8)                     assert (abs(v[0] - u[0]), abs(v[1] - u[1])) in [(0, 1), (1, 0)]
16: (0)             class TestGenericPath:
17: (4)                 @classmethod
18: (4)                 def setup_class(cls):
19: (8)                     from networkx import convert_node_labels_to_integers as cnlti
20: (8)                     cls.grid = cnlti(nx.grid_2d_graph(4, 4), first_label=1,
ordering="sorted")
21: (8)                     cls.cycle = nx.cycle_graph(7)
22: (8)                     cls.directed_cycle = nx.cycle_graph(7, create_using=nx.DiGraph())
23: (8)                     cls.neg_weights = nx.DiGraph()
24: (8)                     cls.neg_weights.add_edge(0, 1, weight=1)
25: (8)                     cls.neg_weights.add_edge(0, 2, weight=3)
26: (8)                     cls.neg_weights.add_edge(1, 3, weight=1)
27: (8)                     cls.neg_weights.add_edge(2, 3, weight=-2)
28: (4)                 def test_shortest_path(self):
29: (8)                     assert nx.shortest_path(self.cycle, 0, 3) == [0, 1, 2, 3]
30: (8)                     assert nx.shortest_path(self.cycle, 0, 4) == [0, 6, 5, 4]
31: (8)                     validate_grid_path(4, 4, 1, 12, nx.shortest_path(self.grid, 1, 12))
32: (8)                     assert nx.shortest_path(self.directed_cycle, 0, 3) == [0, 1, 2, 3]
33: (8)                     assert nx.shortest_path(self.cycle, 0, 3, weight="weight") == [0, 1,
2, 3]
34: (8)                     assert nx.shortest_path(self.cycle, 0, 4, weight="weight") == [0, 6,
5, 4]
35: (8)                     validate_grid_path(
36: (12)                        4, 4, 1, 12, nx.shortest_path(self.grid, 1, 12, weight="weight")
37: (8)                     )
38: (8)                     assert nx.shortest_path(self.directed_cycle, 0, 3, weight="weight") ==
[
39: (12)                        0,
40: (12)                        1,
41: (12)                        2,
42: (12)                        3,
43: (8)                     ]
44: (8)                     assert nx.shortest_path(
45: (12)                        self.directed_cycle, 0, 3, weight="weight", method="dijkstra"
46: (8)                     ) == [0, 1, 2, 3]
47: (8)                     assert nx.shortest_path(
48: (12)                        self.directed_cycle, 0, 3, weight="weight", method="bellman-ford"
49: (8)                     ) == [0, 1, 2, 3]
50: (8)                     assert nx.shortest_path(
51: (12)                        self.neg_weights, 0, 3, weight="weight", method="bellman-ford"
52: (8)                     ) == [0, 2, 3]
53: (8)                     pytest.raises(ValueError, nx.shortest_path, self.cycle, method="SPAM")
54: (8)                     pytest.raises(nx.NodeNotFound, nx.shortest_path, self.cycle, 8)
55: (4)                 def test_shortest_path_target(self):
56: (8)                     answer = {0: [0, 1], 1: [1], 2: [2, 1]}
57: (8)                     sp = nx.shortest_path(nx.path_graph(3), target=1)
58: (8)                     assert sp == answer
59: (8)                     sp = nx.shortest_path(nx.path_graph(3), target=1, weight="weight")
60: (8)                     assert sp == answer
61: (8)                     sp = nx.shortest_path(
62: (12)                        nx.path_graph(3), target=1, weight="weight", method="dijkstra"
63: (8)                     )
64: (8)                     assert sp == answer
65: (8)                     sp = nx.shortest_path(
66: (12)                        nx.path_graph(3), target=1, weight="weight", method="bellman-ford"
67: (8)                     )
68: (8)                     assert sp == answer
69: (4)                 def test_shortest_path_length(self):
70: (8)                     assert nx.shortest_path_length(self.cycle, 0, 3) == 3
71: (8)                     assert nx.shortest_path_length(self.grid, 1, 12) == 5
72: (8)                     assert nx.shortest_path_length(self.directed_cycle, 0, 4) == 4
73: (8)                     assert nx.shortest_path_length(self.cycle, 0, 3, weight="weight") == 3
74: (8)                     assert nx.shortest_path_length(self.grid, 1, 12, weight="weight") == 5
75: (8)                     assert nx.shortest_path_length(self.directed_cycle, 0, 4,
weight="weight") == 4
76: (8)                     assert (
77: (12)                        nx.shortest_path_length(
```

```
 78: (16)                            self.cycle, 0, 3, weight="weight", method="dijkstra"
 79: (12)                        )
 80: (12)                        == 3
 81: (8)                     )
 82: (8)                     assert (
 83: (12)                        nx.shortest_path_length(
 84: (16)                            self.cycle, 0, 3, weight="weight", method="bellman-ford"
 85: (12)                        )
 86: (12)                        == 3
 87: (8)                     )
 88: (8)                     pytest.raises(ValueError, nx.shortest_path_length, self.cycle,
method="SPAM")
 89: (8)                     pytest.raises(nx.NodeNotFound, nx.shortest_path_length, self.cycle, 8)
 90: (4)                 def test_shortest_path_length_target(self):
 91: (8)                     answer = {0: 1, 1: 0, 2: 1}
 92: (8)                     sp = dict(nx.shortest_path_length(nx.path_graph(3), target=1))
 93: (8)                     assert sp == answer
 94: (8)                     sp = nx.shortest_path_length(nx.path_graph(3), target=1,
weight="weight")
 95: (8)                     assert sp == answer
 96: (8)                     sp = nx.shortest_path_length(
 97: (12)                        nx.path_graph(3), target=1, weight="weight", method="dijkstra"
 98: (8)                     )
 99: (8)                     assert sp == answer
 100: (8)                    sp = nx.shortest_path_length(
 101: (12)                       nx.path_graph(3), target=1, weight="weight", method="bellman-ford"
 102: (8)                    )
 103: (8)                    assert sp == answer
 104: (4)                def test_single_source_shortest_path(self):
 105: (8)                    p = nx.shortest_path(self.cycle, 0)
 106: (8)                    assert p[3] == [0, 1, 2, 3]
 107: (8)                    assert p == nx.single_source_shortest_path(self.cycle, 0)
 108: (8)                    p = nx.shortest_path(self.grid, 1)
 109: (8)                    validate_grid_path(4, 4, 1, 12, p[12])
 110: (8)                    p = nx.shortest_path(self.cycle, 0, weight="weight")
 111: (8)                    assert p[3] == [0, 1, 2, 3]
 112: (8)                    assert p == nx.single_source_dijkstra_path(self.cycle, 0)
 113: (8)                    p = nx.shortest_path(self.grid, 1, weight="weight")
 114: (8)                    validate_grid_path(4, 4, 1, 12, p[12])
 115: (8)                    p = nx.shortest_path(self.cycle, 0, method="dijkstra",
weight="weight")
 116: (8)                    assert p[3] == [0, 1, 2, 3]
 117: (8)                    assert p == nx.single_source_shortest_path(self.cycle, 0)
 118: (8)                    p = nx.shortest_path(self.cycle, 0, method="bellman-ford",
weight="weight")
 119: (8)                    assert p[3] == [0, 1, 2, 3]
 120: (8)                    assert p == nx.single_source_shortest_path(self.cycle, 0)
 121: (4)                def test_single_source_shortest_path_length(self):
 122: (8)                    ans = dict(nx.shortest_path_length(self.cycle, 0))
 123: (8)                    assert ans == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
 124: (8)                    assert ans == dict(nx.single_source_shortest_path_length(self.cycle,
0))
 125: (8)                    ans = dict(nx.shortest_path_length(self.grid, 1))
 126: (8)                    assert ans[16] == 6
 127: (8)                    ans = dict(nx.shortest_path_length(self.cycle, 0, weight="weight"))
 128: (8)                    assert ans == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
 129: (8)                    assert ans == dict(nx.single_source_dijkstra_path_length(self.cycle,
0))
 130: (8)                    ans = dict(nx.shortest_path_length(self.grid, 1, weight="weight"))
 131: (8)                    assert ans[16] == 6
 132: (8)                    ans = dict(
 133: (12)                       nx.shortest_path_length(self.cycle, 0, weight="weight",
method="dijkstra")
 134: (8)                    )
 135: (8)                    assert ans == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
 136: (8)                    assert ans == dict(nx.single_source_dijkstra_path_length(self.cycle,
0))
 137: (8)                    ans = dict(
 138: (12)                       nx.shortest_path_length(
```

```
139: (16)                            self.cycle, 0, weight="weight", method="bellman-ford"
140: (12)                        )
141: (8)                     )
142: (8)                     assert ans == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
143: (8)                     assert ans ==
dict(nx.single_source_bellman_ford_path_length(self.cycle, 0))
144: (4)                 def test_single_source_all_shortest_paths(self):
145: (8)                     cycle_ans = {0: [[0]], 1: [[0, 1]], 2: [[0, 1, 2], [0, 3, 2]], 3: [[0,
3]]}
146: (8)                     ans = dict(nx.single_source_all_shortest_paths(nx.cycle_graph(4), 0))
147: (8)                     assert sorted(ans[2]) == cycle_ans[2]
148: (8)                     ans = dict(nx.single_source_all_shortest_paths(self.grid, 1))
149: (8)                     grid_ans = [
150: (12)                        [1, 2, 3, 7, 11],
151: (12)                        [1, 2, 6, 7, 11],
152: (12)                        [1, 2, 6, 10, 11],
153: (12)                        [1, 5, 6, 7, 11],
154: (12)                        [1, 5, 6, 10, 11],
155: (12)                        [1, 5, 9, 10, 11],
156: (8)                     ]
157: (8)                     assert sorted(ans[11]) == grid_ans
158: (8)                     ans = dict(
159: (12)                        nx.single_source_all_shortest_paths(nx.cycle_graph(4), 0,
weight="weight")
160: (8)                     )
161: (8)                     assert sorted(ans[2]) == cycle_ans[2]
162: (8)                     ans = dict(
163: (12)                        nx.single_source_all_shortest_paths(
164: (16)                            nx.cycle_graph(4), 0, method="bellman-ford", weight="weight"
165: (12)                        )
166: (8)                     )
167: (8)                     assert sorted(ans[2]) == cycle_ans[2]
168: (8)                     ans = dict(nx.single_source_all_shortest_paths(self.grid, 1,
weight="weight"))
169: (8)                     assert sorted(ans[11]) == grid_ans
170: (8)                     ans = dict(
171: (12)                        nx.single_source_all_shortest_paths(
172: (16)                            self.grid, 1, method="bellman-ford", weight="weight"
173: (12)                        )
174: (8)                     )
175: (8)                     assert sorted(ans[11]) == grid_ans
176: (8)                     G = nx.cycle_graph(4)
177: (8)                     G.add_node(4)
178: (8)                     ans = dict(nx.single_source_all_shortest_paths(G, 0))
179: (8)                     assert sorted(ans[2]) == [[0, 1, 2], [0, 3, 2]]
180: (8)                     ans = dict(nx.single_source_all_shortest_paths(G, 4))
181: (8)                     assert sorted(ans[4]) == [[4]]
182: (4)                 def test_all_pairs_shortest_path(self):
183: (8)                     p = nx.shortest_path(self.cycle)
184: (8)                     assert p[0][3] == [0, 1, 2, 3]
185: (8)                     assert p == dict(nx.all_pairs_shortest_path(self.cycle))
186: (8)                     p = dict(nx.shortest_path(self.grid))
187: (8)                     validate_grid_path(4, 4, 1, 12, p[1][12])
188: (8)                     p = dict(nx.shortest_path(self.cycle, weight="weight"))
189: (8)                     assert p[0][3] == [0, 1, 2, 3]
190: (8)                     assert p == dict(nx.all_pairs_dijkstra_path(self.cycle))
191: (8)                     p = dict(nx.shortest_path(self.grid, weight="weight"))
192: (8)                     validate_grid_path(4, 4, 1, 12, p[1][12])
193: (8)                     p = dict(nx.shortest_path(self.cycle, weight="weight",
method="dijkstra"))
194: (8)                     assert p[0][3] == [0, 1, 2, 3]
195: (8)                     assert p == dict(nx.all_pairs_dijkstra_path(self.cycle))
196: (8)                     p = dict(nx.shortest_path(self.cycle, weight="weight",
method="bellman-ford"))
197: (8)                     assert p[0][3] == [0, 1, 2, 3]
198: (8)                     assert p == dict(nx.all_pairs_bellman_ford_path(self.cycle))
199: (4)                 def test_all_pairs_shortest_path_length(self):
200: (8)                     ans = dict(nx.shortest_path_length(self.cycle))
201: (8)                     assert ans[0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
```

```
(8)                     assert ans == dict(nx.all_pairs_shortest_path_length(self.cycle))
(8)                     ans = dict(nx.shortest_path_length(self.grid))
(8)                     assert ans[1][16] == 6
(8)                     ans = dict(nx.shortest_path_length(self.cycle, weight="weight"))
(8)                     assert ans[0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
(8)                     assert ans == dict(nx.all_pairs_dijkstra_path_length(self.cycle))
(8)                     ans = dict(nx.shortest_path_length(self.grid, weight="weight"))
(8)                     assert ans[1][16] == 6
(8)                     ans = dict(
(12)                        nx.shortest_path_length(self.cycle, weight="weight",
method="dijkstra")
(8)                     )
(8)                     assert ans[0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
(8)                     assert ans == dict(nx.all_pairs_dijkstra_path_length(self.cycle))
(8)                     ans = dict(
(12)                        nx.shortest_path_length(self.cycle, weight="weight",
method="bellman-ford")
(8)                     )
(8)                     assert ans[0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
(8)                     assert ans == dict(nx.all_pairs_bellman_ford_path_length(self.cycle))
(4)                 def test_all_pairs_all_shortest_paths(self):
(8)                     ans = dict(nx.all_pairs_all_shortest_paths(nx.cycle_graph(4)))
(8)                     assert sorted(ans[1][3]) == [[1, 0, 3], [1, 2, 3]]
(8)                     ans = dict(nx.all_pairs_all_shortest_paths(nx.cycle_graph(4)),
weight="weight")
(8)                     assert sorted(ans[1][3]) == [[1, 0, 3], [1, 2, 3]]
(8)                     ans = dict(
(12)                        nx.all_pairs_all_shortest_paths(nx.cycle_graph(4)),
(12)                        method="bellman-ford",
(12)                        weight="weight",
(8)                     )
(8)                     assert sorted(ans[1][3]) == [[1, 0, 3], [1, 2, 3]]
(8)                     G = nx.cycle_graph(4)
(8)                     G.add_node(4)
(8)                     ans = dict(nx.all_pairs_all_shortest_paths(G))
(8)                     assert sorted(ans[4][4]) == [[4]]
(4)                 def test_has_path(self):
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, range(3))
(8)                     nx.add_path(G, range(3, 5))
(8)                     assert nx.has_path(G, 0, 2)
(8)                     assert not nx.has_path(G, 0, 4)
(4)                 def test_has_path_singleton(self):
(8)                     G = nx.empty_graph(1)
(8)                     assert nx.has_path(G, 0, 0)
(4)                 def test_all_shortest_paths(self):
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [0, 1, 2, 3])
(8)                     nx.add_path(G, [0, 10, 20, 3])
(8)                     assert [[0, 1, 2, 3], [0, 10, 20, 3]] ==
sorted(nx.all_shortest_paths(G, 0, 3))
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [0, 1, 2, 3])
(8)                     nx.add_path(G, [0, 10, 20, 3])
(8)                     assert [[0, 1, 2, 3], [0, 10, 20, 3]] == sorted(
(12)                        nx.all_shortest_paths(G, 0, 3, weight="weight")
(8)                     )
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [0, 1, 2, 3])
(8)                     nx.add_path(G, [0, 10, 20, 3])
(8)                     assert [[0, 1, 2, 3], [0, 10, 20, 3]] == sorted(
(12)                        nx.all_shortest_paths(G, 0, 3, weight="weight", method="dijkstra")
(8)                     )
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [0, 1, 2, 3])
(8)                     nx.add_path(G, [0, 10, 20, 3])
(8)                     assert [[0, 1, 2, 3], [0, 10, 20, 3]] == sorted(
(12)                        nx.all_shortest_paths(G, 0, 3, weight="weight", method="bellman-
ford")
```

```
266: (8)                    )
267: (4)                def test_all_shortest_paths_raise(self):
268: (8)                    with pytest.raises(nx.NetworkXNoPath):
269: (12)                       G = nx.path_graph(4)
270: (12)                       G.add_node(4)
271: (12)                       list(nx.all_shortest_paths(G, 0, 4))
272: (4)                def test_bad_method(self):
273: (8)                    with pytest.raises(ValueError):
274: (12)                       G = nx.path_graph(2)
275: (12)                       list(nx.all_shortest_paths(G, 0, 1, weight="weight",
method="SPAM"))
276: (4)                def test_single_source_all_shortest_paths_bad_method(self):
277: (8)                    with pytest.raises(ValueError):
278: (12)                       G = nx.path_graph(2)
279: (12)                       dict(
280: (16)                           nx.single_source_all_shortest_paths(
281: (20)                               G, 0, weight="weight", method="SPAM"
282: (16)                           )
283: (12)                       )
284: (4)                def test_all_shortest_paths_zero_weight_edge(self):
285: (8)                    g = nx.Graph()
286: (8)                    nx.add_path(g, [0, 1, 3])
287: (8)                    nx.add_path(g, [0, 1, 2, 3])
288: (8)                    g.edges[1, 2]["weight"] = 0
289: (8)                    paths30d = list(
290: (12)                       nx.all_shortest_paths(g, 3, 0, weight="weight", method="dijkstra")
291: (8)                    )
292: (8)                    paths03d = list(
293: (12)                       nx.all_shortest_paths(g, 0, 3, weight="weight", method="dijkstra")
294: (8)                    )
295: (8)                    paths30b = list(
296: (12)                       nx.all_shortest_paths(g, 3, 0, weight="weight", method="bellman-
ford")
297: (8)                    )
298: (8)                    paths03b = list(
299: (12)                       nx.all_shortest_paths(g, 0, 3, weight="weight", method="bellman-
ford")
300: (8)                    )
301: (8)                    assert sorted(paths03d) == sorted(p[::-1] for p in paths30d)
302: (8)                    assert sorted(paths03d) == sorted(p[::-1] for p in paths30b)
303: (8)                    assert sorted(paths03b) == sorted(p[::-1] for p in paths30b)
304: (0)            class TestAverageShortestPathLength:
305: (4)                def test_cycle_graph(self):
306: (8)                    ans = nx.average_shortest_path_length(nx.cycle_graph(7))
307: (8)                    assert ans == pytest.approx(2, abs=1e-7)
308: (4)                def test_path_graph(self):
309: (8)                    ans = nx.average_shortest_path_length(nx.path_graph(5))
310: (8)                    assert ans == pytest.approx(2, abs=1e-7)
311: (4)                def test_weighted(self):
312: (8)                    G = nx.Graph()
313: (8)                    nx.add_cycle(G, range(7), weight=2)
314: (8)                    ans = nx.average_shortest_path_length(G, weight="weight")
315: (8)                    assert ans == pytest.approx(4, abs=1e-7)
316: (8)                    G = nx.Graph()
317: (8)                    nx.add_path(G, range(5), weight=2)
318: (8)                    ans = nx.average_shortest_path_length(G, weight="weight")
319: (8)                    assert ans == pytest.approx(4, abs=1e-7)
320: (4)                def test_specified_methods(self):
321: (8)                    G = nx.Graph()
322: (8)                    nx.add_cycle(G, range(7), weight=2)
323: (8)                    ans = nx.average_shortest_path_length(G, weight="weight",
method="dijkstra")
324: (8)                    assert ans == pytest.approx(4, abs=1e-7)
325: (8)                    ans = nx.average_shortest_path_length(G, weight="weight",
method="bellman-ford")
326: (8)                    assert ans == pytest.approx(4, abs=1e-7)
327: (8)                    ans = nx.average_shortest_path_length(
328: (12)                       G, weight="weight", method="floyd-warshall"
329: (8)                    )
```

```
330: (8)                    assert ans == pytest.approx(4, abs=1e-7)
331: (8)                    G = nx.Graph()
332: (8)                    nx.add_path(G, range(5), weight=2)
333: (8)                    ans = nx.average_shortest_path_length(G, weight="weight",
method="dijkstra")
334: (8)                    assert ans == pytest.approx(4, abs=1e-7)
335: (8)                    ans = nx.average_shortest_path_length(G, weight="weight",
method="bellman-ford")
336: (8)                    assert ans == pytest.approx(4, abs=1e-7)
337: (8)                    ans = nx.average_shortest_path_length(
338: (12)                       G, weight="weight", method="floyd-warshall"
339: (8)                    )
340: (8)                    assert ans == pytest.approx(4, abs=1e-7)
341: (4)                def test_directed_not_strongly_connected(self):
342: (8)                    G = nx.DiGraph([(0, 1)])
343: (8)                    with pytest.raises(nx.NetworkXError, match="Graph is not strongly
connected"):
344: (12)                       nx.average_shortest_path_length(G)
345: (4)                def test_undirected_not_connected(self):
346: (8)                    g = nx.Graph()
347: (8)                    g.add_nodes_from(range(3))
348: (8)                    g.add_edge(0, 1)
349: (8)                    pytest.raises(nx.NetworkXError, nx.average_shortest_path_length, g)
350: (4)                def test_trivial_graph(self):
351: (8)                    """Tests that the trivial graph has average path length zero,
352: (8)                    since there is exactly one path of length zero in the trivial
353: (8)                    graph.
354: (8)                    For more information, see issue #1960.
355: (8)                    """
356: (8)                    G = nx.trivial_graph()
357: (8)                    assert nx.average_shortest_path_length(G) == 0
358: (4)                def test_null_graph(self):
359: (8)                    with pytest.raises(nx.NetworkXPointlessConcept):
360: (12)                       nx.average_shortest_path_length(nx.null_graph())
361: (4)                def test_bad_method(self):
362: (8)                    with pytest.raises(ValueError):
363: (12)                       G = nx.path_graph(2)
364: (12)                       nx.average_shortest_path_length(G, weight="weight", method="SPAM")
365: (0)            class TestAverageShortestPathLengthNumpy:
366: (4)                @classmethod
367: (4)                def setup_class(cls):
368: (8)                    global np
369: (8)                    import pytest
370: (8)                    np = pytest.importorskip("numpy")
371: (4)                def test_specified_methods_numpy(self):
372: (8)                    G = nx.Graph()
373: (8)                    nx.add_cycle(G, range(7), weight=2)
374: (8)                    ans = nx.average_shortest_path_length(
375: (12)                       G, weight="weight", method="floyd-warshall-numpy"
376: (8)                    )
377: (8)                    np.testing.assert_almost_equal(ans, 4)
378: (8)                    G = nx.Graph()
379: (8)                    nx.add_path(G, range(5), weight=2)
380: (8)                    ans = nx.average_shortest_path_length(
381: (12)                       G, weight="weight", method="floyd-warshall-numpy"
382: (8)                    )
383: (8)                    np.testing.assert_almost_equal(ans, 4)

----------------------------------------

File 303 - test_unweighted.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def validate_grid_path(r, c, s, t, p):
4: (4)                  assert isinstance(p, list)
5: (4)                  assert p[0] == s
6: (4)                  assert p[-1] == t
7: (4)                  s = ((s - 1) // c, (s - 1) % c)
```

```
8: (4)                  t = ((t - 1) // c, (t - 1) % c)
9: (4)                  assert len(p) == abs(t[0] - s[0]) + abs(t[1] - s[1]) + 1
10: (4)                 p = [((u - 1) // c, (u - 1) % c) for u in p]
11: (4)                 for u in p:
12: (8)                     assert 0 <= u[0] < r
13: (8)                     assert 0 <= u[1] < c
14: (4)                 for u, v in zip(p[:-1], p[1:]):
15: (8)                     assert (abs(v[0] - u[0]), abs(v[1] - u[1])) in [(0, 1), (1, 0)]
16: (0)             class TestUnweightedPath:
17: (4)                 @classmethod
18: (4)                 def setup_class(cls):
19: (8)                     from networkx import convert_node_labels_to_integers as cnlti
20: (8)                     cls.grid = cnlti(nx.grid_2d_graph(4, 4), first_label=1,
ordering="sorted")
21: (8)                     cls.cycle = nx.cycle_graph(7)
22: (8)                     cls.directed_cycle = nx.cycle_graph(7, create_using=nx.DiGraph())
23: (4)                 def test_bidirectional_shortest_path(self):
24: (8)                     assert nx.bidirectional_shortest_path(self.cycle, 0, 3) == [0, 1, 2,
3]
25: (8)                     assert nx.bidirectional_shortest_path(self.cycle, 0, 4) == [0, 6, 5,
4]
26: (8)                     validate_grid_path(
27: (12)                        4, 4, 1, 12, nx.bidirectional_shortest_path(self.grid, 1, 12)
28: (8)                     )
29: (8)                     assert nx.bidirectional_shortest_path(self.directed_cycle, 0, 3) ==
[0, 1, 2, 3]
30: (8)                     assert nx.bidirectional_shortest_path(self.cycle, 3, 3) == [3]
31: (4)                 @pytest.mark.parametrize(
32: (8)                     ("src", "tgt"),
33: (8)                     (
34: (12)                        (8, 3),  # source not in graph
35: (12)                        (3, 8),  # target not in graph
36: (12)                        (8, 10),  # neither source nor target in graph
37: (12)                        (8, 8),  # src == tgt, neither in graph - tests order of input
checks
38: (8)                     ),
39: (4)                 )
40: (4)                 def test_bidirectional_shortest_path_src_tgt_not_in_graph(self, src, tgt):
41: (8)                     with pytest.raises(
42: (12)                        nx.NodeNotFound,
43: (12)                        match=f"Either source {src} or target {tgt} is not in G",
44: (8)                     ):
45: (12)                        nx.bidirectional_shortest_path(self.cycle, src, tgt)
46: (4)                 def test_shortest_path_length(self):
47: (8)                     assert nx.shortest_path_length(self.cycle, 0, 3) == 3
48: (8)                     assert nx.shortest_path_length(self.grid, 1, 12) == 5
49: (8)                     assert nx.shortest_path_length(self.directed_cycle, 0, 4) == 4
50: (8)                     assert nx.shortest_path_length(self.cycle, 0, 3, weight=True) == 3
51: (8)                     assert nx.shortest_path_length(self.grid, 1, 12, weight=True) == 5
52: (8)                     assert nx.shortest_path_length(self.directed_cycle, 0, 4, weight=True)
== 4
53: (4)                 def test_single_source_shortest_path(self):
54: (8)                     p = nx.single_source_shortest_path(self.directed_cycle, 3)
55: (8)                     assert p[0] == [3, 4, 5, 6, 0]
56: (8)                     p = nx.single_source_shortest_path(self.cycle, 0)
57: (8)                     assert p[3] == [0, 1, 2, 3]
58: (8)                     p = nx.single_source_shortest_path(self.cycle, 0, cutoff=0)
59: (8)                     assert p == {0: [0]}
60: (4)                 def test_single_source_shortest_path_length(self):
61: (8)                     pl = nx.single_source_shortest_path_length
62: (8)                     lengths = {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
63: (8)                     assert dict(pl(self.cycle, 0)) == lengths
64: (8)                     lengths = {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6}
65: (8)                     assert dict(pl(self.directed_cycle, 0)) == lengths
66: (4)                 def test_single_target_shortest_path(self):
67: (8)                     p = nx.single_target_shortest_path(self.directed_cycle, 0)
68: (8)                     assert p[3] == [3, 4, 5, 6, 0]
69: (8)                     p = nx.single_target_shortest_path(self.cycle, 0)
70: (8)                     assert p[3] == [3, 2, 1, 0]
```

```
71: (8)                     p = nx.single_target_shortest_path(self.cycle, 0, cutoff=0)
72: (8)                     assert p == {0: [0]}
73: (8)                     target = 8
74: (8)                     with pytest.raises(nx.NodeNotFound, match=f"Target {target} not in
G"):
75: (12)                        nx.single_target_shortest_path(self.cycle, target)
76: (4)                 def test_single_target_shortest_path_length(self):
77: (8)                     pl = nx.single_target_shortest_path_length
78: (8)                     lengths = {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
79: (8)                     assert dict(pl(self.cycle, 0)) == lengths
80: (8)                     lengths = {0: 0, 1: 6, 2: 5, 3: 4, 4: 3, 5: 2, 6: 1}
81: (8)                     assert dict(pl(self.directed_cycle, 0)) == lengths
82: (8)                     target = 8
83: (8)                     with pytest.raises(nx.NodeNotFound, match=f"Target {target} is not in
G"):
84: (12)                        nx.single_target_shortest_path_length(self.cycle, target)
85: (4)                 def test_all_pairs_shortest_path(self):
86: (8)                     p = dict(nx.all_pairs_shortest_path(self.cycle))
87: (8)                     assert p[0][3] == [0, 1, 2, 3]
88: (8)                     p = dict(nx.all_pairs_shortest_path(self.grid))
89: (8)                     validate_grid_path(4, 4, 1, 12, p[1][12])
90: (4)                 def test_all_pairs_shortest_path_length(self):
91: (8)                     l = dict(nx.all_pairs_shortest_path_length(self.cycle))
92: (8)                     assert l[0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
93: (8)                     l = dict(nx.all_pairs_shortest_path_length(self.grid))
94: (8)                     assert l[1][16] == 6
95: (4)                 def test_predecessor_path(self):
96: (8)                     G = nx.path_graph(4)
97: (8)                     assert nx.predecessor(G, 0) == {0: [], 1: [0], 2: [1], 3: [2]}
98: (8)                     assert nx.predecessor(G, 0, 3) == [2]
99: (4)                 def test_predecessor_cycle(self):
100: (8)                    G = nx.cycle_graph(4)
101: (8)                    pred = nx.predecessor(G, 0)
102: (8)                    assert pred[0] == []
103: (8)                    assert pred[1] == [0]
104: (8)                    assert pred[2] in [[1, 3], [3, 1]]
105: (8)                    assert pred[3] == [0]
106: (4)                def test_predecessor_cutoff(self):
107: (8)                    G = nx.path_graph(4)
108: (8)                    p = nx.predecessor(G, 0, 3)
109: (8)                    assert 4 not in p
110: (4)                def test_predecessor_target(self):
111: (8)                    G = nx.path_graph(4)
112: (8)                    p = nx.predecessor(G, 0, 3)
113: (8)                    assert p == [2]
114: (8)                    p = nx.predecessor(G, 0, 3, cutoff=2)
115: (8)                    assert p == []
116: (8)                    p, s = nx.predecessor(G, 0, 3, return_seen=True)
117: (8)                    assert p == [2]
118: (8)                    assert s == 3
119: (8)                    p, s = nx.predecessor(G, 0, 3, cutoff=2, return_seen=True)
120: (8)                    assert p == []
121: (8)                    assert s == -1
122: (4)                def test_predecessor_missing_source(self):
123: (8)                    source = 8
124: (8)                    with pytest.raises(nx.NodeNotFound, match=f"Source {source} not in
G"):
125: (12)                       nx.predecessor(self.cycle, source)

----------------------------------------

File 304 - __init__.py:

1: (0)

----------------------------------------

File 305 - test_weighted.py:
```

```
(0)  import pytest
(0)  import networkx as nx
(0)  from networkx.utils import pairwise
(0)  def validate_path(G, s, t, soln_len, path, weight="weight"):
(4)      assert path[0] == s
(4)      assert path[-1] == t
(4)      if callable(weight):
(8)          weight_f = weight
(4)      else:
(8)          if G.is_multigraph():
(12)             def weight_f(u, v, d):
(16)                 return min(e.get(weight, 1) for e in d.values())
(8)          else:
(12)             def weight_f(u, v, d):
(16)                 return d.get(weight, 1)
(4)      computed = sum(weight_f(u, v, G[u][v]) for u, v in pairwise(path))
(4)      assert soln_len == computed
(0)  def validate_length_path(G, s, t, soln_len, length, path, weight="weight"):
(4)      assert soln_len == length
(4)      validate_path(G, s, t, length, path, weight=weight)
(0)  class WeightedTestBase:
(4)      """Base class for test classes that test functions for computing
(4)      shortest paths in weighted graphs.
(4)      """
(4)      def setup_method(self):
(8)          """Creates some graphs for use in the unit tests."""
(8)          cnlti = nx.convert_node_labels_to_integers
(8)          self.grid = cnlti(nx.grid_2d_graph(4, 4), first_label=1,
ordering="sorted")
(8)          self.cycle = nx.cycle_graph(7)
(8)          self.directed_cycle = nx.cycle_graph(7, create_using=nx.DiGraph())
(8)          self.XG = nx.DiGraph()
(8)          self.XG.add_weighted_edges_from(
(12)             [
(16)                 ("s", "u", 10),
(16)                 ("s", "x", 5),
(16)                 ("u", "v", 1),
(16)                 ("u", "x", 2),
(16)                 ("v", "y", 1),
(16)                 ("x", "u", 3),
(16)                 ("x", "v", 5),
(16)                 ("x", "y", 2),
(16)                 ("y", "s", 7),
(16)                 ("y", "v", 6),
(12)             ]
(8)          )
(8)          self.MXG = nx.MultiDiGraph(self.XG)
(8)          self.MXG.add_edge("s", "u", weight=15)
(8)          self.XG2 = nx.DiGraph()
(8)          self.XG2.add_weighted_edges_from(
(12)             [
(16)                 [1, 4, 1],
(16)                 [4, 5, 1],
(16)                 [5, 6, 1],
(16)                 [6, 3, 1],
(16)                 [1, 3, 50],
(16)                 [1, 2, 100],
(16)                 [2, 3, 100],
(12)             ]
(8)          )
(8)          self.XG3 = nx.Graph()
(8)          self.XG3.add_weighted_edges_from(
(12)             [[0, 1, 2], [1, 2, 12], [2, 3, 1], [3, 4, 5], [4, 5, 1], [5, 0,
10]]
(8)          )
(8)          self.XG4 = nx.Graph()
(8)          self.XG4.add_weighted_edges_from(
(12)             [
(16)                 [0, 1, 2],
```

```
68: (16)                            [1, 2, 2],
69: (16)                            [2, 3, 1],
70: (16)                            [3, 4, 1],
71: (16)                            [4, 5, 1],
72: (16)                            [5, 6, 1],
73: (16)                            [6, 7, 1],
74: (16)                            [7, 0, 1],
75: (12)                        ]
76: (8)                     )
77: (8)                     self.MXG4 = nx.MultiGraph(self.XG4)
78: (8)                     self.MXG4.add_edge(0, 1, weight=3)
79: (8)                     self.G = nx.DiGraph()  # no weights
80: (8)                     self.G.add_edges_from(
81: (12)                        [
82: (16)                            ("s", "u"),
83: (16)                            ("s", "x"),
84: (16)                            ("u", "v"),
85: (16)                            ("u", "x"),
86: (16)                            ("v", "y"),
87: (16)                            ("x", "u"),
88: (16)                            ("x", "v"),
89: (16)                            ("x", "y"),
90: (16)                            ("y", "s"),
91: (16)                            ("y", "v"),
92: (12)                        ]
93: (8)                     )
94: (0)             class TestWeightedPath(WeightedTestBase):
95: (4)                 def test_dijkstra(self):
96: (8)                     (D, P) = nx.single_source_dijkstra(self.XG, "s")
97: (8)                     validate_path(self.XG, "s", "v", 9, P["v"])
98: (8)                     assert D["v"] == 9
99: (8)                     validate_path(
100: (12)                       self.XG, "s", "v", 9, nx.single_source_dijkstra_path(self.XG, "s")
["v"]
101: (8)                    )
102: (8)                    assert dict(nx.single_source_dijkstra_path_length(self.XG, "s"))["v"]
== 9
103: (8)                    validate_path(
104: (12)                       self.XG, "s", "v", 9, nx.single_source_dijkstra(self.XG, "s")[1]
["v"]
105: (8)                    )
106: (8)                    validate_path(
107: (12)                       self.MXG, "s", "v", 9, nx.single_source_dijkstra_path(self.MXG,
"s")["v"]
108: (8)                    )
109: (8)                    GG = self.XG.to_undirected()
110: (8)                    GG["u"]["x"]["weight"] = 2
111: (8)                    (D, P) = nx.single_source_dijkstra(GG, "s")
112: (8)                    validate_path(GG, "s", "v", 8, P["v"])
113: (8)                    assert D["v"] == 8  # uses lower weight of 2 on u<->x edge
114: (8)                    validate_path(GG, "s", "v", 8, nx.dijkstra_path(GG, "s", "v"))
115: (8)                    assert nx.dijkstra_path_length(GG, "s", "v") == 8
116: (8)                    validate_path(self.XG2, 1, 3, 4, nx.dijkstra_path(self.XG2, 1, 3))
117: (8)                    validate_path(self.XG3, 0, 3, 15, nx.dijkstra_path(self.XG3, 0, 3))
118: (8)                    assert nx.dijkstra_path_length(self.XG3, 0, 3) == 15
119: (8)                    validate_path(self.XG4, 0, 2, 4, nx.dijkstra_path(self.XG4, 0, 2))
120: (8)                    assert nx.dijkstra_path_length(self.XG4, 0, 2) == 4
121: (8)                    validate_path(self.MXG4, 0, 2, 4, nx.dijkstra_path(self.MXG4, 0, 2))
122: (8)                    validate_path(
123: (12)                       self.G, "s", "v", 2, nx.single_source_dijkstra(self.G, "s", "v")
[1]
124: (8)                    )
125: (8)                    validate_path(
126: (12)                       self.G, "s", "v", 2, nx.single_source_dijkstra(self.G, "s")[1]
["v"]
127: (8)                    )
128: (8)                    validate_path(self.G, "s", "v", 2, nx.dijkstra_path(self.G, "s", "v"))
129: (8)                    assert nx.dijkstra_path_length(self.G, "s", "v") == 2
130: (8)                    pytest.raises(nx.NetworkXNoPath, nx.dijkstra_path, self.G, "s",
```

```
"moon")
131: (8)                    pytest.raises(nx.NetworkXNoPath, nx.dijkstra_path_length, self.G, "s",
"moon")
132: (8)                    validate_path(self.cycle, 0, 3, 3, nx.dijkstra_path(self.cycle, 0, 3))
133: (8)                    validate_path(self.cycle, 0, 4, 3, nx.dijkstra_path(self.cycle, 0, 4))
134: (8)                    assert nx.single_source_dijkstra(self.cycle, 0, 0) == (0, [0])
135: (4)                def test_bidirectional_dijkstra(self):
136: (8)                    validate_length_path(
137: (12)                       self.XG, "s", "v", 9, *nx.bidirectional_dijkstra(self.XG, "s",
"v")
138: (8)                    )
139: (8)                    validate_length_path(
140: (12)                       self.G, "s", "v", 2, *nx.bidirectional_dijkstra(self.G, "s", "v")
141: (8)                    )
142: (8)                    validate_length_path(
143: (12)                       self.cycle, 0, 3, 3, *nx.bidirectional_dijkstra(self.cycle, 0, 3)
144: (8)                    )
145: (8)                    validate_length_path(
146: (12)                       self.cycle, 0, 4, 3, *nx.bidirectional_dijkstra(self.cycle, 0, 4)
147: (8)                    )
148: (8)                    validate_length_path(
149: (12)                       self.XG3, 0, 3, 15, *nx.bidirectional_dijkstra(self.XG3, 0, 3)
150: (8)                    )
151: (8)                    validate_length_path(
152: (12)                       self.XG4, 0, 2, 4, *nx.bidirectional_dijkstra(self.XG4, 0, 2)
153: (8)                    )
154: (8)                    P = nx.single_source_dijkstra_path(self.XG, "s")["v"]
155: (8)                    validate_path(
156: (12)                       self.XG,
157: (12)                       "s",
158: (12)                       "v",
159: (12)                       sum(self.XG[u][v]["weight"] for u, v in zip(P[:-1], P[1:])),
160: (12)                       nx.dijkstra_path(self.XG, "s", "v"),
161: (8)                    )
162: (8)                    G = nx.path_graph(2)
163: (8)                    pytest.raises(nx.NodeNotFound, nx.bidirectional_dijkstra, G, 3, 0)
164: (4)                def test_weight_functions(self):
165: (8)                    def heuristic(*z):
166: (12)                       return sum(val**2 for val in z)
167: (8)                    def getpath(pred, v, s):
168: (12)                       return [v] if v == s else getpath(pred, pred[v], s) + [v]
169: (8)                    def goldberg_radzik(g, s, t, weight="weight"):
170: (12)                       pred, dist = nx.goldberg_radzik(g, s, weight=weight)
171: (12)                       dist = dist[t]
172: (12)                       return dist, getpath(pred, t, s)
173: (8)                    def astar(g, s, t, weight="weight"):
174: (12)                       path = nx.astar_path(g, s, t, heuristic, weight=weight)
175: (12)                       dist = nx.astar_path_length(g, s, t, heuristic, weight=weight)
176: (12)                       return dist, path
177: (8)                    def vlp(G, s, t, l, F, w):
178: (12)                       res = F(G, s, t, weight=w)
179: (12)                       validate_length_path(G, s, t, l, *res, weight=w)
180: (8)                    G = self.cycle
181: (8)                    s = 6
182: (8)                    t = 4
183: (8)                    path = [6] + list(range(t + 1))
184: (8)                    def weight(u, v, _):
185: (12)                       return 1 + v**2
186: (8)                    length = sum(weight(u, v, None) for u, v in pairwise(path))
187: (8)                    vlp(G, s, t, length, nx.bidirectional_dijkstra, weight)
188: (8)                    vlp(G, s, t, length, nx.single_source_dijkstra, weight)
189: (8)                    vlp(G, s, t, length, nx.single_source_bellman_ford, weight)
190: (8)                    vlp(G, s, t, length, goldberg_radzik, weight)
191: (8)                    vlp(G, s, t, length, astar, weight)
192: (8)                    def weight(u, v, _):
193: (12)                       return 2 ** (u * v)
194: (8)                    length = sum(weight(u, v, None) for u, v in pairwise(path))
195: (8)                    vlp(G, s, t, length, nx.bidirectional_dijkstra, weight)
196: (8)                    vlp(G, s, t, length, nx.single_source_dijkstra, weight)
```

```
        vlp(G, s, t, length, nx.single_source_bellman_ford, weight)
        vlp(G, s, t, length, goldberg_radzik, weight)
        vlp(G, s, t, length, astar, weight)
    def test_bidirectional_dijkstra_no_path(self):
        with pytest.raises(nx.NetworkXNoPath):
            G = nx.Graph()
            nx.add_path(G, [1, 2, 3])
            nx.add_path(G, [4, 5, 6])
            path = nx.bidirectional_dijkstra(G, 1, 6)
    @pytest.mark.parametrize(
        "fn",
        (
            nx.dijkstra_path,
            nx.dijkstra_path_length,
            nx.single_source_dijkstra_path,
            nx.single_source_dijkstra_path_length,
            nx.single_source_dijkstra,
            nx.dijkstra_predecessor_and_distance,
        ),
    )
    def test_absent_source(self, fn):
        G = nx.path_graph(2)
        with pytest.raises(nx.NodeNotFound):
            fn(G, 3, 0)
        with pytest.raises(nx.NodeNotFound):
            fn(G, 3, 3)
    def test_dijkstra_predecessor1(self):
        G = nx.path_graph(4)
        assert nx.dijkstra_predecessor_and_distance(G, 0) == (
            {0: [], 1: [0], 2: [1], 3: [2]},
            {0: 0, 1: 1, 2: 2, 3: 3},
        )
    def test_dijkstra_predecessor2(self):
        G = nx.Graph([(0, 1), (1, 2), (2, 3), (3, 0)])
        pred, dist = nx.dijkstra_predecessor_and_distance(G, (0))
        assert pred[0] == []
        assert pred[1] == [0]
        assert pred[2] in [[1, 3], [3, 1]]
        assert pred[3] == [0]
        assert dist == {0: 0, 1: 1, 2: 2, 3: 1}
    def test_dijkstra_predecessor3(self):
        XG = nx.DiGraph()
        XG.add_weighted_edges_from(
            [
                ("s", "u", 10),
                ("s", "x", 5),
                ("u", "v", 1),
                ("u", "x", 2),
                ("v", "y", 1),
                ("x", "u", 3),
                ("x", "v", 5),
                ("x", "y", 2),
                ("y", "s", 7),
                ("y", "v", 6),
            ]
        )
        (P, D) = nx.dijkstra_predecessor_and_distance(XG, "s")
        assert P["v"] == ["u"]
        assert D["v"] == 9
        (P, D) = nx.dijkstra_predecessor_and_distance(XG, "s", cutoff=8)
        assert "v" not in D
    def test_single_source_dijkstra_path_length(self):
        pl = nx.single_source_dijkstra_path_length
        assert dict(pl(self.MXG4, 0))[2] == 4
        spl = pl(self.MXG4, 0, cutoff=2)
        assert 2 not in spl
    def test_bidirectional_dijkstra_multigraph(self):
        G = nx.MultiGraph()
        G.add_edge("a", "b", weight=10)
```

```
(8)                 G.add_edge("a", "b", weight=100)
(8)                 dp = nx.bidirectional_dijkstra(G, "a", "b")
(8)                 assert dp == (10, ["a", "b"])
(4)             def test_dijkstra_pred_distance_multigraph(self):
(8)                 G = nx.MultiGraph()
(8)                 G.add_edge("a", "b", key="short", foo=5, weight=100)
(8)                 G.add_edge("a", "b", key="long", bar=1, weight=110)
(8)                 p, d = nx.dijkstra_predecessor_and_distance(G, "a")
(8)                 assert p == {"a": [], "b": ["a"]}
(8)                 assert d == {"a": 0, "b": 100}
(4)             def test_negative_edge_cycle(self):
(8)                 G = nx.cycle_graph(5, create_using=nx.DiGraph())
(8)                 assert not nx.negative_edge_cycle(G)
(8)                 G.add_edge(8, 9, weight=-7)
(8)                 G.add_edge(9, 8, weight=3)
(8)                 graph_size = len(G)
(8)                 assert nx.negative_edge_cycle(G)
(8)                 assert graph_size == len(G)
(8)                 pytest.raises(ValueError, nx.single_source_dijkstra_path_length, G, 8)
(8)                 pytest.raises(ValueError, nx.single_source_dijkstra, G, 8)
(8)                 pytest.raises(ValueError, nx.dijkstra_predecessor_and_distance, G, 8)
(8)                 G.add_edge(9, 10)
(8)                 pytest.raises(ValueError, nx.bidirectional_dijkstra, G, 8, 10)
(8)                 G = nx.MultiDiGraph()
(8)                 G.add_edge(2, 2, weight=-1)
(8)                 assert nx.negative_edge_cycle(G)
(4)             def test_negative_edge_cycle_empty(self):
(8)                 G = nx.DiGraph()
(8)                 assert not nx.negative_edge_cycle(G)
(4)             def test_negative_edge_cycle_custom_weight_key(self):
(8)                 d = nx.DiGraph()
(8)                 d.add_edge("a", "b", w=-2)
(8)                 d.add_edge("b", "a", w=-1)
(8)                 assert nx.negative_edge_cycle(d, weight="w")
(4)             def test_weight_function(self):
(8)                 """Tests that a callable weight is interpreted as a weight
(8)                 function instead of an edge attribute.
(8)                 """
(8)                 G = nx.complete_graph(3)
(8)                 G.adj[0][2]["weight"] = 10
(8)                 G.adj[0][1]["weight"] = 1
(8)                 G.adj[1][2]["weight"] = 1
(8)                 def weight(u, v, d):
(12)                    return 1 / d["weight"]
(8)                 distance, path = nx.single_source_dijkstra(G, 0, 2)
(8)                 assert distance == 2
(8)                 assert path == [0, 1, 2]
(8)                 distance, path = nx.single_source_dijkstra(G, 0, 2, weight=weight)
(8)                 assert distance == 1 / 10
(8)                 assert path == [0, 2]
(4)             def test_all_pairs_dijkstra_path(self):
(8)                 cycle = nx.cycle_graph(7)
(8)                 p = dict(nx.all_pairs_dijkstra_path(cycle))
(8)                 assert p[0][3] == [0, 1, 2, 3]
(8)                 cycle[1][2]["weight"] = 10
(8)                 p = dict(nx.all_pairs_dijkstra_path(cycle))
(8)                 assert p[0][3] == [0, 6, 5, 4, 3]
(4)             def test_all_pairs_dijkstra_path_length(self):
(8)                 cycle = nx.cycle_graph(7)
(8)                 pl = dict(nx.all_pairs_dijkstra_path_length(cycle))
(8)                 assert pl[0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
(8)                 cycle[1][2]["weight"] = 10
(8)                 pl = dict(nx.all_pairs_dijkstra_path_length(cycle))
(8)                 assert pl[0] == {0: 0, 1: 1, 2: 5, 3: 4, 4: 3, 5: 2, 6: 1}
(4)             def test_all_pairs_dijkstra(self):
(8)                 cycle = nx.cycle_graph(7)
(8)                 out = dict(nx.all_pairs_dijkstra(cycle))
(8)                 assert out[0][0] == {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 2, 6: 1}
(8)                 assert out[0][1][3] == [0, 1, 2, 3]
```

```
335: (8)                    cycle[1][2]["weight"] = 10
336: (8)                    out = dict(nx.all_pairs_dijkstra(cycle))
337: (8)                    assert out[0][0] == {0: 0, 1: 1, 2: 5, 3: 4, 4: 3, 5: 2, 6: 1}
338: (8)                    assert out[0][1][3] == [0, 6, 5, 4, 3]
339: (0)            class TestDijkstraPathLength:
340: (4)                """Unit tests for the :func:`networkx.dijkstra_path_length`
341: (4)                function.
342: (4)                """
343: (4)                def test_weight_function(self):
344: (8)                    """Tests for computing the length of the shortest path using
345: (8)                    Dijkstra's algorithm with a user-defined weight function.
346: (8)                    """
347: (8)                    G = nx.complete_graph(3)
348: (8)                    G.adj[0][2]["weight"] = 10
349: (8)                    G.adj[0][1]["weight"] = 1
350: (8)                    G.adj[1][2]["weight"] = 1
351: (8)                    def weight(u, v, d):
352: (12)                       return 1 / d["weight"]
353: (8)                    length = nx.dijkstra_path_length(G, 0, 2, weight=weight)
354: (8)                    assert length == 1 / 10
355: (0)            class TestMultiSourceDijkstra:
356: (4)                """Unit tests for the multi-source dialect of Dijkstra's shortest
357: (4)                path algorithms.
358: (4)                """
359: (4)                def test_no_sources(self):
360: (8)                    with pytest.raises(ValueError):
361: (12)                       nx.multi_source_dijkstra(nx.Graph(), {})
362: (4)                def test_path_no_sources(self):
363: (8)                    with pytest.raises(ValueError):
364: (12)                       nx.multi_source_dijkstra_path(nx.Graph(), {})
365: (4)                def test_path_length_no_sources(self):
366: (8)                    with pytest.raises(ValueError):
367: (12)                       nx.multi_source_dijkstra_path_length(nx.Graph(), {})
368: (4)                @pytest.mark.parametrize(
369: (8)                    "fn",
370: (8)                    (
371: (12)                       nx.multi_source_dijkstra_path,
372: (12)                       nx.multi_source_dijkstra_path_length,
373: (12)                       nx.multi_source_dijkstra,
374: (8)                    ),
375: (4)                )
376: (4)                def test_absent_source(self, fn):
377: (8)                    G = nx.path_graph(2)
378: (8)                    with pytest.raises(nx.NodeNotFound):
379: (12)                       fn(G, [3], 0)
380: (8)                    with pytest.raises(nx.NodeNotFound):
381: (12)                       fn(G, [3], 3)
382: (4)                def test_two_sources(self):
383: (8)                    edges = [(0, 1, 1), (1, 2, 1), (2, 3, 10), (3, 4, 1)]
384: (8)                    G = nx.Graph()
385: (8)                    G.add_weighted_edges_from(edges)
386: (8)                    sources = {0, 4}
387: (8)                    distances, paths = nx.multi_source_dijkstra(G, sources)
388: (8)                    expected_distances = {0: 0, 1: 1, 2: 2, 3: 1, 4: 0}
389: (8)                    expected_paths = {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [4, 3], 4: [4]}
390: (8)                    assert distances == expected_distances
391: (8)                    assert paths == expected_paths
392: (4)                def test_simple_paths(self):
393: (8)                    G = nx.path_graph(4)
394: (8)                    lengths = nx.multi_source_dijkstra_path_length(G, [0])
395: (8)                    assert lengths == {n: n for n in G}
396: (8)                    paths = nx.multi_source_dijkstra_path(G, [0])
397: (8)                    assert paths == {n: list(range(n + 1)) for n in G}
398: (0)            class TestBellmanFordAndGoldbergRadzik(WeightedTestBase):
399: (4)                def test_single_node_graph(self):
400: (8)                    G = nx.DiGraph()
401: (8)                    G.add_node(0)
402: (8)                    assert nx.single_source_bellman_ford_path(G, 0) == {0: [0]}
403: (8)                    assert nx.single_source_bellman_ford_path_length(G, 0) == {0: 0}
```

```
(8)                     assert nx.single_source_bellman_ford(G, 0) == ({0: 0}, {0: [0]})
(8)                     assert nx.bellman_ford_predecessor_and_distance(G, 0) == ({0: []}, {0:
0})
(8)                     assert nx.goldberg_radzik(G, 0) == ({0: None}, {0: 0})
(4)                 def test_absent_source_bellman_ford(self):
(8)                     G = nx.path_graph(2)
(8)                     for fn in (
(12)                        nx.bellman_ford_predecessor_and_distance,
(12)                        nx.bellman_ford_path,
(12)                        nx.bellman_ford_path_length,
(12)                        nx.single_source_bellman_ford_path,
(12)                        nx.single_source_bellman_ford_path_length,
(12)                        nx.single_source_bellman_ford,
(8)                     ):
(12)                        pytest.raises(nx.NodeNotFound, fn, G, 3, 0)
(12)                        pytest.raises(nx.NodeNotFound, fn, G, 3, 3)
(4)                 def test_absent_source_goldberg_radzik(self):
(8)                     with pytest.raises(nx.NodeNotFound):
(12)                        G = nx.path_graph(2)
(12)                        nx.goldberg_radzik(G, 3, 0)
(4)                 def test_negative_cycle_heuristic(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge(0, 1, weight=-1)
(8)                     G.add_edge(1, 2, weight=-1)
(8)                     G.add_edge(2, 3, weight=-1)
(8)                     G.add_edge(3, 0, weight=3)
(8)                     assert not nx.negative_edge_cycle(G, heuristic=True)
(8)                     G.add_edge(2, 0, weight=1.999)
(8)                     assert nx.negative_edge_cycle(G, heuristic=True)
(8)                     G.edges[2, 0]["weight"] = 2
(8)                     assert not nx.negative_edge_cycle(G, heuristic=True)
(4)                 def test_negative_cycle_consistency(self):
(8)                     import random
(8)                     unif = random.uniform
(8)                     for random_seed in range(2):  # range(20):
(12)                        random.seed(random_seed)
(12)                        for density in [0.1, 0.9]:  # .3, .7, .9]:
(16)                            for N in [1, 10, 20]:  # range(1, 60 - int(30 * density)):
(20)                                for max_cost in [1, 90]:  # [1, 10, 40, 90]:
(24)                                    G = nx.binomial_graph(N, density, seed=4,
directed=True)
(24)                                    edges = ((u, v, unif(-1, max_cost)) for u, v in
G.edges)
(24)                                    G.add_weighted_edges_from(edges)
(24)                                    no_heuristic = nx.negative_edge_cycle(G,
heuristic=False)
(24)                                    with_heuristic = nx.negative_edge_cycle(G,
heuristic=True)
(24)                                    assert no_heuristic == with_heuristic
(4)                 def test_negative_cycle(self):
(8)                     G = nx.cycle_graph(5, create_using=nx.DiGraph())
(8)                     G.add_edge(1, 2, weight=-7)
(8)                     for i in range(5):
(12)                        pytest.raises(
(16)                            nx.NetworkXUnbounded, nx.single_source_bellman_ford_path, G, i
(12)                        )
(12)                        pytest.raises(
(16)                            nx.NetworkXUnbounded,
nx.single_source_bellman_ford_path_length, G, i
(12)                        )
(12)                        pytest.raises(nx.NetworkXUnbounded, nx.single_source_bellman_ford,
G, i)
(12)                        pytest.raises(
(16)                            nx.NetworkXUnbounded,
nx.bellman_ford_predecessor_and_distance, G, i
(12)                        )
(12)                        pytest.raises(nx.NetworkXUnbounded, nx.goldberg_radzik, G, i)
(8)                     G = nx.cycle_graph(5)  # undirected Graph
(8)                     G.add_edge(1, 2, weight=-3)
```

```
465: (8)                    for i in range(5):
466: (12)                       pytest.raises(
467: (16)                           nx.NetworkXUnbounded, nx.single_source_bellman_ford_path, G, i
468: (12)                       )
469: (12)                       pytest.raises(
470: (16)                           nx.NetworkXUnbounded,
nx.single_source_bellman_ford_path_length, G, i
471: (12)                       )
472: (12)                       pytest.raises(nx.NetworkXUnbounded, nx.single_source_bellman_ford,
G, i)
473: (12)                       pytest.raises(
474: (16)                           nx.NetworkXUnbounded,
nx.bellman_ford_predecessor_and_distance, G, i
475: (12)                       )
476: (12)                       pytest.raises(nx.NetworkXUnbounded, nx.goldberg_radzik, G, i)
477: (8)                    G = nx.DiGraph([(1, 1, {"weight": -1})])
478: (8)                    pytest.raises(nx.NetworkXUnbounded,
nx.single_source_bellman_ford_path, G, 1)
479: (8)                    pytest.raises(
480: (12)                       nx.NetworkXUnbounded, nx.single_source_bellman_ford_path_length,
G, 1
481: (8)                    )
482: (8)                    pytest.raises(nx.NetworkXUnbounded, nx.single_source_bellman_ford, G,
1)
483: (8)                    pytest.raises(
484: (12)                       nx.NetworkXUnbounded, nx.bellman_ford_predecessor_and_distance, G,
1
485: (8)                    )
486: (8)                    pytest.raises(nx.NetworkXUnbounded, nx.goldberg_radzik, G, 1)
487: (8)                    G = nx.MultiDiGraph([(1, 1, {"weight": -1})])
488: (8)                    pytest.raises(nx.NetworkXUnbounded,
nx.single_source_bellman_ford_path, G, 1)
489: (8)                    pytest.raises(
490: (12)                       nx.NetworkXUnbounded, nx.single_source_bellman_ford_path_length,
G, 1
491: (8)                    )
492: (8)                    pytest.raises(nx.NetworkXUnbounded, nx.single_source_bellman_ford, G,
1)
493: (8)                    pytest.raises(
494: (12)                       nx.NetworkXUnbounded, nx.bellman_ford_predecessor_and_distance, G,
1
495: (8)                    )
496: (8)                    pytest.raises(nx.NetworkXUnbounded, nx.goldberg_radzik, G, 1)
497: (4)                def test_zero_cycle(self):
498: (8)                    G = nx.cycle_graph(5, create_using=nx.DiGraph())
499: (8)                    G.add_edge(2, 3, weight=-4)
500: (8)                    nx.goldberg_radzik(G, 1)
501: (8)                    nx.bellman_ford_predecessor_and_distance(G, 1)
502: (8)                    G.add_edge(2, 3, weight=-4.0001)
503: (8)                    pytest.raises(
504: (12)                       nx.NetworkXUnbounded, nx.bellman_ford_predecessor_and_distance, G,
1
505: (8)                    )
506: (8)                    pytest.raises(nx.NetworkXUnbounded, nx.goldberg_radzik, G, 1)
507: (4)                def test_find_negative_cycle_longer_cycle(self):
508: (8)                    G = nx.cycle_graph(5, create_using=nx.DiGraph())
509: (8)                    nx.add_cycle(G, [3, 5, 6, 7, 8, 9])
510: (8)                    G.add_edge(1, 2, weight=-30)
511: (8)                    assert nx.find_negative_cycle(G, 1) == [0, 1, 2, 3, 4, 0]
512: (8)                    assert nx.find_negative_cycle(G, 7) == [2, 3, 4, 0, 1, 2]
513: (4)                def test_find_negative_cycle_no_cycle(self):
514: (8)                    G = nx.path_graph(5, create_using=nx.DiGraph())
515: (8)                    pytest.raises(nx.NetworkXError, nx.find_negative_cycle, G, 3)
516: (4)                def test_find_negative_cycle_single_edge(self):
517: (8)                    G = nx.Graph()
518: (8)                    G.add_edge(0, 1, weight=-1)
519: (8)                    assert nx.find_negative_cycle(G, 1) == [1, 0, 1]
520: (4)                def test_negative_weight(self):
521: (8)                    G = nx.cycle_graph(5, create_using=nx.DiGraph())
```

```
        G.add_edge(1, 2, weight=-3)
        assert nx.single_source_bellman_ford_path(G, 0) == {
            0: [0],
            1: [0, 1],
            2: [0, 1, 2],
            3: [0, 1, 2, 3],
            4: [0, 1, 2, 3, 4],
        }
        assert nx.single_source_bellman_ford_path_length(G, 0) == {
            0: 0,
            1: 1,
            2: -2,
            3: -1,
            4: 0,
        }
        assert nx.single_source_bellman_ford(G, 0) == (
            {0: 0, 1: 1, 2: -2, 3: -1, 4: 0},
            {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 1, 2, 3, 4]},
        )
        assert nx.bellman_ford_predecessor_and_distance(G, 0) == (
            {0: [], 1: [0], 2: [1], 3: [2], 4: [3]},
            {0: 0, 1: 1, 2: -2, 3: -1, 4: 0},
        )
        assert nx.goldberg_radzik(G, 0) == (
            {0: None, 1: 0, 2: 1, 3: 2, 4: 3},
            {0: 0, 1: 1, 2: -2, 3: -1, 4: 0},
        )
    def test_not_connected(self):
        G = nx.complete_graph(6)
        G.add_edge(10, 11)
        G.add_edge(10, 12)
        assert nx.single_source_bellman_ford_path(G, 0) == {
            0: [0],
            1: [0, 1],
            2: [0, 2],
            3: [0, 3],
            4: [0, 4],
            5: [0, 5],
        }
        assert nx.single_source_bellman_ford_path_length(G, 0) == {
            0: 0,
            1: 1,
            2: 1,
            3: 1,
            4: 1,
            5: 1,
        }
        assert nx.single_source_bellman_ford(G, 0) == (
            {0: 0, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1},
            {0: [0], 1: [0, 1], 2: [0, 2], 3: [0, 3], 4: [0, 4], 5: [0, 5]},
        )
        assert nx.bellman_ford_predecessor_and_distance(G, 0) == (
            {0: [], 1: [0], 2: [0], 3: [0], 4: [0], 5: [0]},
            {0: 0, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1},
        )
        assert nx.goldberg_radzik(G, 0) == (
            {0: None, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0},
            {0: 0, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1},
        )
        G = nx.complete_graph(6)
        G.add_edges_from(
            [
                ("A", "B", {"load": 3}),
                ("B", "C", {"load": -10}),
                ("C", "A", {"load": 2}),
            ]
        )
        assert nx.single_source_bellman_ford_path(G, 0, weight="load") == {
```

```
(12)                        0: [0],
(12)                        1: [0, 1],
(12)                        2: [0, 2],
(12)                        3: [0, 3],
(12)                        4: [0, 4],
(12)                        5: [0, 5],
(8)                     }
(8)                     assert nx.single_source_bellman_ford_path_length(G, 0, weight="load")
== {
(12)                        0: 0,
(12)                        1: 1,
(12)                        2: 1,
(12)                        3: 1,
(12)                        4: 1,
(12)                        5: 1,
(8)                     }
(8)                     assert nx.single_source_bellman_ford(G, 0, weight="load") == (
(12)                        {0: 0, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1},
(12)                        {0: [0], 1: [0, 1], 2: [0, 2], 3: [0, 3], 4: [0, 4], 5: [0, 5]},
(8)                     )
(8)                     assert nx.bellman_ford_predecessor_and_distance(G, 0, weight="load")
== (
(12)                        {0: [], 1: [0], 2: [0], 3: [0], 4: [0], 5: [0]},
(12)                        {0: 0, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1},
(8)                     )
(8)                     assert nx.goldberg_radzik(G, 0, weight="load") == (
(12)                        {0: None, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0},
(12)                        {0: 0, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1},
(8)                     )
(4)                 def test_multigraph(self):
(8)                     assert nx.bellman_ford_path(self.MXG, "s", "v") == ["s", "x", "u",
"v"]
(8)                     assert nx.bellman_ford_path_length(self.MXG, "s", "v") == 9
(8)                     assert nx.single_source_bellman_ford_path(self.MXG, "s")["v"] == [
(12)                        "s",
(12)                        "x",
(12)                        "u",
(12)                        "v",
(8)                     ]
(8)                     assert nx.single_source_bellman_ford_path_length(self.MXG, "s")["v"]
== 9
(8)                     D, P = nx.single_source_bellman_ford(self.MXG, "s", target="v")
(8)                     assert D == 9
(8)                     assert P == ["s", "x", "u", "v"]
(8)                     P, D = nx.bellman_ford_predecessor_and_distance(self.MXG, "s")
(8)                     assert P["v"] == ["u"]
(8)                     assert D["v"] == 9
(8)                     P, D = nx.goldberg_radzik(self.MXG, "s")
(8)                     assert P["v"] == "u"
(8)                     assert D["v"] == 9
(8)                     assert nx.bellman_ford_path(self.MXG4, 0, 2) == [0, 1, 2]
(8)                     assert nx.bellman_ford_path_length(self.MXG4, 0, 2) == 4
(8)                     assert nx.single_source_bellman_ford_path(self.MXG4, 0)[2] == [0, 1,
2]
(8)                     assert nx.single_source_bellman_ford_path_length(self.MXG4, 0)[2] == 4
(8)                     D, P = nx.single_source_bellman_ford(self.MXG4, 0, target=2)
(8)                     assert D == 4
(8)                     assert P == [0, 1, 2]
(8)                     P, D = nx.bellman_ford_predecessor_and_distance(self.MXG4, 0)
(8)                     assert P[2] == [1]
(8)                     assert D[2] == 4
(8)                     P, D = nx.goldberg_radzik(self.MXG4, 0)
(8)                     assert P[2] == 1
(8)                     assert D[2] == 4
(4)                 def test_others(self):
(8)                     assert nx.bellman_ford_path(self.XG, "s", "v") == ["s", "x", "u", "v"]
(8)                     assert nx.bellman_ford_path_length(self.XG, "s", "v") == 9
(8)                     assert nx.single_source_bellman_ford_path(self.XG, "s")["v"] == [
(12)                        "s",
```

```
(12)                        "x",
(12)                        "u",
(12)                        "v",
(8)                     ]
(8)                     assert nx.single_source_bellman_ford_path_length(self.XG, "s")["v"] ==
9
(8)                     D, P = nx.single_source_bellman_ford(self.XG, "s", target="v")
(8)                     assert D == 9
(8)                     assert P == ["s", "x", "u", "v"]
(8)                     (P, D) = nx.bellman_ford_predecessor_and_distance(self.XG, "s")
(8)                     assert P["v"] == ["u"]
(8)                     assert D["v"] == 9
(8)                     (P, D) = nx.goldberg_radzik(self.XG, "s")
(8)                     assert P["v"] == "u"
(8)                     assert D["v"] == 9
(4)                 def test_path_graph(self):
(8)                     G = nx.path_graph(4)
(8)                     assert nx.single_source_bellman_ford_path(G, 0) == {
(12)                        0: [0],
(12)                        1: [0, 1],
(12)                        2: [0, 1, 2],
(12)                        3: [0, 1, 2, 3],
(8)                     }
(8)                     assert nx.single_source_bellman_ford_path_length(G, 0) == {
(12)                        0: 0,
(12)                        1: 1,
(12)                        2: 2,
(12)                        3: 3,
(8)                     }
(8)                     assert nx.single_source_bellman_ford(G, 0) == (
(12)                        {0: 0, 1: 1, 2: 2, 3: 3},
(12)                        {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3]},
(8)                     )
(8)                     assert nx.bellman_ford_predecessor_and_distance(G, 0) == (
(12)                        {0: [], 1: [0], 2: [1], 3: [2]},
(12)                        {0: 0, 1: 1, 2: 2, 3: 3},
(8)                     )
(8)                     assert nx.goldberg_radzik(G, 0) == (
(12)                        {0: None, 1: 0, 2: 1, 3: 2},
(12)                        {0: 0, 1: 1, 2: 2, 3: 3},
(8)                     )
(8)                     assert nx.single_source_bellman_ford_path(G, 3) == {
(12)                        0: [3, 2, 1, 0],
(12)                        1: [3, 2, 1],
(12)                        2: [3, 2],
(12)                        3: [3],
(8)                     }
(8)                     assert nx.single_source_bellman_ford_path_length(G, 3) == {
(12)                        0: 3,
(12)                        1: 2,
(12)                        2: 1,
(12)                        3: 0,
(8)                     }
(8)                     assert nx.single_source_bellman_ford(G, 3) == (
(12)                        {0: 3, 1: 2, 2: 1, 3: 0},
(12)                        {0: [3, 2, 1, 0], 1: [3, 2, 1], 2: [3, 2], 3: [3]},
(8)                     )
(8)                     assert nx.bellman_ford_predecessor_and_distance(G, 3) == (
(12)                        {0: [1], 1: [2], 2: [3], 3: []},
(12)                        {0: 3, 1: 2, 2: 1, 3: 0},
(8)                     )
(8)                     assert nx.goldberg_radzik(G, 3) == (
(12)                        {0: 1, 1: 2, 2: 3, 3: None},
(12)                        {0: 3, 1: 2, 2: 1, 3: 0},
(8)                     )
(4)                 def test_4_cycle(self):
(8)                     G = nx.Graph([(0, 1), (1, 2), (2, 3), (3, 0)])
(8)                     dist, path = nx.single_source_bellman_ford(G, 0)
(8)                     assert dist == {0: 0, 1: 1, 2: 2, 3: 1}
```

```
(8)                     assert path[0] == [0]
(8)                     assert path[1] == [0, 1]
(8)                     assert path[2] in [[0, 1, 2], [0, 3, 2]]
(8)                     assert path[3] == [0, 3]
(8)                     pred, dist = nx.bellman_ford_predecessor_and_distance(G, 0)
(8)                     assert pred[0] == []
(8)                     assert pred[1] == [0]
(8)                     assert pred[2] in [[1, 3], [3, 1]]
(8)                     assert pred[3] == [0]
(8)                     assert dist == {0: 0, 1: 1, 2: 2, 3: 1}
(8)                     pred, dist = nx.goldberg_radzik(G, 0)
(8)                     assert pred[0] is None
(8)                     assert pred[1] == 0
(8)                     assert pred[2] in [1, 3]
(8)                     assert pred[3] == 0
(8)                     assert dist == {0: 0, 1: 1, 2: 2, 3: 1}
(4)                 def test_negative_weight_bf_path(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from("abcd")
(8)                     G.add_edge("a", "d", weight=0)
(8)                     G.add_edge("a", "b", weight=1)
(8)                     G.add_edge("b", "c", weight=-3)
(8)                     G.add_edge("c", "d", weight=1)
(8)                     assert nx.bellman_ford_path(G, "a", "d") == ["a", "b", "c", "d"]
(8)                     assert nx.bellman_ford_path_length(G, "a", "d") == -1
(4)                 def test_zero_cycle_smoke(self):
(8)                     D = nx.DiGraph()
(8)                     D.add_weighted_edges_from([(0, 1, 1), (1, 2, 1), (2, 3, 1), (3, 1,
-2)])
(8)                     nx.bellman_ford_path(D, 1, 3)
(8)                     nx.dijkstra_path(D, 1, 3)
(8)                     nx.bidirectional_dijkstra(D, 1, 3)
(0)             class TestJohnsonAlgorithm(WeightedTestBase):
(4)                 def test_single_node_graph(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_node(0)
(8)                     assert nx.johnson(G) == {0: {0: [0]}}
(4)                 def test_negative_cycle(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            ("0", "3", 3),
(16)                            ("0", "1", -5),
(16)                            ("1", "0", -5),
(16)                            ("0", "2", 2),
(16)                            ("1", "2", 4),
(16)                            ("2", "3", 1),
(12)                        ]
(8)                     )
(8)                     pytest.raises(nx.NetworkXUnbounded, nx.johnson, G)
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            ("0", "3", 3),
(16)                            ("0", "1", -5),
(16)                            ("1", "0", -5),
(16)                            ("0", "2", 2),
(16)                            ("1", "2", 4),
(16)                            ("2", "3", 1),
(12)                        ]
(8)                     )
(8)                     pytest.raises(nx.NetworkXUnbounded, nx.johnson, G)
(4)                 def test_negative_weights(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_weighted_edges_from(
(12)                        [("0", "3", 3), ("0", "1", -5), ("0", "2", 2), ("1", "2", 4),
("2", "3", 1)]
(8)                     )
(8)                     paths = nx.johnson(G)
```

```
789: (8)                    assert paths == {
790: (12)                       "1": {"1": ["1"], "3": ["1", "2", "3"], "2": ["1", "2"]},
791: (12)                       "0": {
792: (16)                           "1": ["0", "1"],
793: (16)                           "0": ["0"],
794: (16)                           "3": ["0", "1", "2", "3"],
795: (16)                           "2": ["0", "1", "2"],
796: (12)                       },
797: (12)                       "3": {"3": ["3"]},
798: (12)                       "2": {"3": ["2", "3"], "2": ["2"]},
799: (8)                    }
800: (4)                def test_unweighted_graph(self):
801: (8)                    G = nx.Graph()
802: (8)                    G.add_edges_from([(1, 0), (2, 1)])
803: (8)                    H = G.copy()
804: (8)                    nx.set_edge_attributes(H, values=1, name="weight")
805: (8)                    assert nx.johnson(G) == nx.johnson(H)
806: (4)                def test_partially_weighted_graph_with_negative_edges(self):
807: (8)                    G = nx.DiGraph()
808: (8)                    G.add_edges_from([(0, 1), (1, 2), (2, 0), (1, 0)])
809: (8)                    G[1][0]["weight"] = -2
810: (8)                    G[0][1]["weight"] = 3
811: (8)                    G[1][2]["weight"] = -4
812: (8)                    H = G.copy()
813: (8)                    H[2][0]["weight"] = 1
814: (8)                    I = G.copy()
815: (8)                    I[2][0]["weight"] = 8
816: (8)                    assert nx.johnson(G) == nx.johnson(H)
817: (8)                    assert nx.johnson(G) != nx.johnson(I)
818: (4)                def test_graphs(self):
819: (8)                    validate_path(self.XG, "s", "v", 9, nx.johnson(self.XG)["s"]["v"])
820: (8)                    validate_path(self.MXG, "s", "v", 9, nx.johnson(self.MXG)["s"]["v"])
821: (8)                    validate_path(self.XG2, 1, 3, 4, nx.johnson(self.XG2)[1][3])
822: (8)                    validate_path(self.XG3, 0, 3, 15, nx.johnson(self.XG3)[0][3])
823: (8)                    validate_path(self.XG4, 0, 2, 4, nx.johnson(self.XG4)[0][2])
824: (8)                    validate_path(self.MXG4, 0, 2, 4, nx.johnson(self.MXG4)[0][2])

----------------------------------------

File 306 - test_boundary.py:

1: (0)              """Unit tests for the :mod:`networkx.algorithms.boundary` module."""
2: (0)              from itertools import combinations
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx import convert_node_labels_to_integers as cnlti
6: (0)              from networkx.utils import edges_equal
7: (0)              class TestNodeBoundary:
8: (4)                  """Unit tests for the :func:`~networkx.node_boundary` function."""
9: (4)                  def test_null_graph(self):
10: (8)                     """Tests that the null graph has empty node boundaries."""
11: (8)                     null = nx.null_graph()
12: (8)                     assert nx.node_boundary(null, []) == set()
13: (8)                     assert nx.node_boundary(null, [], []) == set()
14: (8)                     assert nx.node_boundary(null, [1, 2, 3]) == set()
15: (8)                     assert nx.node_boundary(null, [1, 2, 3], [4, 5, 6]) == set()
16: (8)                     assert nx.node_boundary(null, [1, 2, 3], [3, 4, 5]) == set()
17: (4)                 def test_path_graph(self):
18: (8)                     P10 = cnlti(nx.path_graph(10), first_label=1)
19: (8)                     assert nx.node_boundary(P10, []) == set()
20: (8)                     assert nx.node_boundary(P10, [], []) == set()
21: (8)                     assert nx.node_boundary(P10, [1, 2, 3]) == {4}
22: (8)                     assert nx.node_boundary(P10, [4, 5, 6]) == {3, 7}
23: (8)                     assert nx.node_boundary(P10, [3, 4, 5, 6, 7]) == {2, 8}
24: (8)                     assert nx.node_boundary(P10, [8, 9, 10]) == {7}
25: (8)                     assert nx.node_boundary(P10, [4, 5, 6], [9, 10]) == set()
26: (4)                 def test_complete_graph(self):
27: (8)                     K10 = cnlti(nx.complete_graph(10), first_label=1)
28: (8)                     assert nx.node_boundary(K10, []) == set()
```

```
29: (8)                     assert nx.node_boundary(K10, [], []) == set()
30: (8)                     assert nx.node_boundary(K10, [1, 2, 3]) == {4, 5, 6, 7, 8, 9, 10}
31: (8)                     assert nx.node_boundary(K10, [4, 5, 6]) == {1, 2, 3, 7, 8, 9, 10}
32: (8)                     assert nx.node_boundary(K10, [3, 4, 5, 6, 7]) == {1, 2, 8, 9, 10}
33: (8)                     assert nx.node_boundary(K10, [4, 5, 6], []) == set()
34: (8)                     assert nx.node_boundary(K10, K10) == set()
35: (8)                     assert nx.node_boundary(K10, [1, 2, 3], [3, 4, 5]) == {4, 5}
36: (4)                 def test_petersen(self):
37: (8)                     """Check boundaries in the petersen graph
38: (8)                     cheeger(G,k)=min(|bdy(S)|/|S| for |S|=k, 0<k<=|V(G)|/2)
39: (8)                     """
40: (8)                     def cheeger(G, k):
41: (12)                        return min(len(nx.node_boundary(G, nn)) / k for nn in
combinations(G, k))
42: (8)                     P = nx.petersen_graph()
43: (8)                     assert cheeger(P, 1) == pytest.approx(3.00, abs=1e-2)
44: (8)                     assert cheeger(P, 2) == pytest.approx(2.00, abs=1e-2)
45: (8)                     assert cheeger(P, 3) == pytest.approx(1.67, abs=1e-2)
46: (8)                     assert cheeger(P, 4) == pytest.approx(1.00, abs=1e-2)
47: (8)                     assert cheeger(P, 5) == pytest.approx(0.80, abs=1e-2)
48: (4)                 def test_directed(self):
49: (8)                     """Tests the node boundary of a directed graph."""
50: (8)                     G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)])
51: (8)                     S = {0, 1}
52: (8)                     boundary = nx.node_boundary(G, S)
53: (8)                     expected = {2}
54: (8)                     assert boundary == expected
55: (4)                 def test_multigraph(self):
56: (8)                     """Tests the node boundary of a multigraph."""
57: (8)                     G = nx.MultiGraph(list(nx.cycle_graph(5).edges()) * 2)
58: (8)                     S = {0, 1}
59: (8)                     boundary = nx.node_boundary(G, S)
60: (8)                     expected = {2, 4}
61: (8)                     assert boundary == expected
62: (4)                 def test_multidigraph(self):
63: (8)                     """Tests the edge boundary of a multidigraph."""
64: (8)                     edges = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
65: (8)                     G = nx.MultiDiGraph(edges * 2)
66: (8)                     S = {0, 1}
67: (8)                     boundary = nx.node_boundary(G, S)
68: (8)                     expected = {2}
69: (8)                     assert boundary == expected
70: (0)             class TestEdgeBoundary:
71: (4)                 """Unit tests for the :func:`~networkx.edge_boundary` function."""
72: (4)                 def test_null_graph(self):
73: (8)                     null = nx.null_graph()
74: (8)                     assert list(nx.edge_boundary(null, [])) == []
75: (8)                     assert list(nx.edge_boundary(null, [], [])) == []
76: (8)                     assert list(nx.edge_boundary(null, [1, 2, 3])) == []
77: (8)                     assert list(nx.edge_boundary(null, [1, 2, 3], [4, 5, 6])) == []
78: (8)                     assert list(nx.edge_boundary(null, [1, 2, 3], [3, 4, 5])) == []
79: (4)                 def test_path_graph(self):
80: (8)                     P10 = cnlti(nx.path_graph(10), first_label=1)
81: (8)                     assert list(nx.edge_boundary(P10, [])) == []
82: (8)                     assert list(nx.edge_boundary(P10, [], [])) == []
83: (8)                     assert list(nx.edge_boundary(P10, [1, 2, 3])) == [(3, 4)]
84: (8)                     assert sorted(nx.edge_boundary(P10, [4, 5, 6])) == [(4, 3), (6, 7)]
85: (8)                     assert sorted(nx.edge_boundary(P10, [3, 4, 5, 6, 7])) == [(3, 2), (7,
8)]
86: (8)                     assert list(nx.edge_boundary(P10, [8, 9, 10])) == [(8, 7)]
87: (8)                     assert sorted(nx.edge_boundary(P10, [4, 5, 6], [9, 10])) == []
88: (8)                     assert list(nx.edge_boundary(P10, [1, 2, 3], [3, 4, 5])) == [(2, 3),
(3, 4)]
89: (4)                 def test_complete_graph(self):
90: (8)                     K10 = cnlti(nx.complete_graph(10), first_label=1)
91: (8)                     def ilen(iterable):
92: (12)                        return sum(1 for i in iterable)
93: (8)                     assert list(nx.edge_boundary(K10, [])) == []
94: (8)                     assert list(nx.edge_boundary(K10, [], [])) == []
```

```
95: (8)                     assert ilen(nx.edge_boundary(K10, [1, 2, 3])) == 21
96: (8)                     assert ilen(nx.edge_boundary(K10, [4, 5, 6, 7])) == 24
97: (8)                     assert ilen(nx.edge_boundary(K10, [3, 4, 5, 6, 7])) == 25
98: (8)                     assert ilen(nx.edge_boundary(K10, [8, 9, 10])) == 21
99: (8)                     assert edges_equal(
100: (12)                       nx.edge_boundary(K10, [4, 5, 6], [9, 10]),
101: (12)                       [(4, 9), (4, 10), (5, 9), (5, 10), (6, 9), (6, 10)],
102: (8)                    )
103: (8)                    assert edges_equal(
104: (12)                       nx.edge_boundary(K10, [1, 2, 3], [3, 4, 5]),
105: (12)                       [(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5)],
106: (8)                    )
107: (4)                def test_directed(self):
108: (8)                    """Tests the edge boundary of a directed graph."""
109: (8)                    G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)])
110: (8)                    S = {0, 1}
111: (8)                    boundary = list(nx.edge_boundary(G, S))
112: (8)                    expected = [(1, 2)]
113: (8)                    assert boundary == expected
114: (4)                def test_multigraph(self):
115: (8)                    """Tests the edge boundary of a multigraph."""
116: (8)                    G = nx.MultiGraph(list(nx.cycle_graph(5).edges()) * 2)
117: (8)                    S = {0, 1}
118: (8)                    boundary = list(nx.edge_boundary(G, S))
119: (8)                    expected = [(0, 4), (0, 4), (1, 2), (1, 2)]
120: (8)                    assert boundary == expected
121: (4)                def test_multidigraph(self):
122: (8)                    """Tests the edge boundary of a multidigraph."""
123: (8)                    edges = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
124: (8)                    G = nx.MultiDiGraph(edges * 2)
125: (8)                    S = {0, 1}
126: (8)                    boundary = list(nx.edge_boundary(G, S))
127: (8)                    expected = [(1, 2), (1, 2)]
128: (8)                    assert boundary == expected

----------------------------------------

File 307 - test_asteroidal.py:

1: (0)              import networkx as nx
2: (0)              def test_is_at_free():
3: (4)                  is_at_free = nx.asteroidal.is_at_free
4: (4)                  cycle = nx.cycle_graph(6)
5: (4)                  assert not is_at_free(cycle)
6: (4)                  path = nx.path_graph(6)
7: (4)                  assert is_at_free(path)
8: (4)                  small_graph = nx.complete_graph(2)
9: (4)                  assert is_at_free(small_graph)
10: (4)                 petersen = nx.petersen_graph()
11: (4)                 assert not is_at_free(petersen)
12: (4)                 clique = nx.complete_graph(6)
13: (4)                 assert is_at_free(clique)
14: (4)                 line_clique = nx.line_graph(clique)
15: (4)                 assert not is_at_free(line_clique)

----------------------------------------

File 308 - test_bridges.py:

1: (0)              """Unit tests for bridge-finding algorithms."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestBridges:
5: (4)                  """Unit tests for the bridge-finding function."""
6: (4)                  def test_single_bridge(self):
7: (8)                      edges = [
8: (12)                         (1, 2),
9: (12)                         (2, 3),
10: (12)                        (3, 4),
```

```
 11: (12)                        (3, 5),
 12: (12)                        (5, 6),
 13: (12)                        (6, 7),
 14: (12)                        (7, 8),
 15: (12)                        (5, 9),
 16: (12)                        (9, 10),
 17: (12)                        (1, 3),
 18: (12)                        (1, 4),
 19: (12)                        (2, 5),
 20: (12)                        (5, 10),
 21: (12)                        (6, 8),
 22: (8)                     ]
 23: (8)                     G = nx.Graph(edges)
 24: (8)                     source = 1
 25: (8)                     bridges = list(nx.bridges(G, source))
 26: (8)                     assert bridges == [(5, 6)]
 27: (4)                 def test_barbell_graph(self):
 28: (8)                     G = nx.barbell_graph(3, 0)
 29: (8)                     source = 0
 30: (8)                     bridges = list(nx.bridges(G, source))
 31: (8)                     assert bridges == [(2, 3)]
 32: (4)                 def test_multiedge_bridge(self):
 33: (8)                     edges = [
 34: (12)                        (0, 1),
 35: (12)                        (0, 2),
 36: (12)                        (1, 2),
 37: (12)                        (1, 2),
 38: (12)                        (2, 3),
 39: (12)                        (3, 4),
 40: (12)                        (3, 4),
 41: (8)                     ]
 42: (8)                     G = nx.MultiGraph(edges)
 43: (8)                     assert list(nx.bridges(G)) == [(2, 3)]
 44: (0)             class TestHasBridges:
 45: (4)                 """Unit tests for the has bridges function."""
 46: (4)                 def test_single_bridge(self):
 47: (8)                     edges = [
 48: (12)                        (1, 2),
 49: (12)                        (2, 3),
 50: (12)                        (3, 4),
 51: (12)                        (3, 5),
 52: (12)                        (5, 6),  # The only bridge edge
 53: (12)                        (6, 7),
 54: (12)                        (7, 8),
 55: (12)                        (5, 9),
 56: (12)                        (9, 10),
 57: (12)                        (1, 3),
 58: (12)                        (1, 4),
 59: (12)                        (2, 5),
 60: (12)                        (5, 10),
 61: (12)                        (6, 8),
 62: (8)                     ]
 63: (8)                     G = nx.Graph(edges)
 64: (8)                     assert nx.has_bridges(G)  # Default root
 65: (8)                     assert nx.has_bridges(G, root=1)  # arbitrary root in G
 66: (4)                 def test_has_bridges_raises_root_not_in_G(self):
 67: (8)                     G = nx.Graph()
 68: (8)                     G.add_nodes_from([1, 2, 3])
 69: (8)                     with pytest.raises(nx.NodeNotFound):
 70: (12)                        nx.has_bridges(G, root=6)
 71: (4)                 def test_multiedge_bridge(self):
 72: (8)                     edges = [
 73: (12)                        (0, 1),
 74: (12)                        (0, 2),
 75: (12)                        (1, 2),
 76: (12)                        (1, 2),
 77: (12)                        (2, 3),
 78: (12)                        (3, 4),
 79: (12)                        (3, 4),
```

```
80: (8)                     ]
81: (8)                     G = nx.MultiGraph(edges)
82: (8)                     assert nx.has_bridges(G)
83: (8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
84: (8)                     assert not nx.has_bridges(G)
85: (4)                 def test_bridges_multiple_components(self):
86: (8)                     G = nx.Graph()
87: (8)                     nx.add_path(G, [0, 1, 2])  # One connected component
88: (8)                     nx.add_path(G, [4, 5, 6])  # Another connected component
89: (8)                     assert list(nx.bridges(G, root=4)) == [(4, 5), (5, 6)]
90: (0)             class TestLocalBridges:
91: (4)                 """Unit tests for the local_bridge function."""
92: (4)                 @classmethod
93: (4)                 def setup_class(cls):
94: (8)                     cls.BB = nx.barbell_graph(4, 0)
95: (8)                     cls.square = nx.cycle_graph(4)
96: (8)                     cls.tri = nx.cycle_graph(3)
97: (4)                 def test_nospan(self):
98: (8)                     expected = {(3, 4), (4, 3)}
99: (8)                     assert next(nx.local_bridges(self.BB, with_span=False)) in expected
100: (8)                    assert set(nx.local_bridges(self.square, with_span=False)) ==
self.square.edges
101: (8)                    assert list(nx.local_bridges(self.tri, with_span=False)) == []
102: (4)                def test_no_weight(self):
103: (8)                    inf = float("inf")
104: (8)                    expected = {(3, 4, inf), (4, 3, inf)}
105: (8)                    assert next(nx.local_bridges(self.BB)) in expected
106: (8)                    expected = {(u, v, 3) for u, v in self.square.edges}
107: (8)                    assert set(nx.local_bridges(self.square)) == expected
108: (8)                    assert list(nx.local_bridges(self.tri)) == []
109: (4)                def test_weight(self):
110: (8)                    inf = float("inf")
111: (8)                    G = self.square.copy()
112: (8)                    G.edges[1, 2]["weight"] = 2
113: (8)                    expected = {(u, v, 5 - wt) for u, v, wt in G.edges(data="weight",
default=1)}
114: (8)                    assert set(nx.local_bridges(G, weight="weight")) == expected
115: (8)                    expected = {(u, v, 6) for u, v in G.edges}
116: (8)                    lb = nx.local_bridges(G, weight=lambda u, v, d: 2)
117: (8)                    assert set(lb) == expected
```

File 309 - test_broadcasting.py:

```
1: (0)              """Unit tests for the broadcasting module."""
2: (0)              import math
3: (0)              import networkx as nx
4: (0)              def test_example_tree_broadcast():
5: (4)                  """
6: (4)                  Test the BROADCAST algorithm on the example in the paper titled:
"Information Dissemination in Trees"
7: (4)                  """
8: (4)                  edge_list = [
9: (8)                      (0, 1),
10: (8)                     (1, 2),
11: (8)                     (2, 7),
12: (8)                     (3, 4),
13: (8)                     (5, 4),
14: (8)                     (4, 7),
15: (8)                     (6, 7),
16: (8)                     (7, 9),
17: (8)                     (8, 9),
18: (8)                     (9, 13),
19: (8)                     (13, 14),
20: (8)                     (14, 15),
21: (8)                     (14, 16),
22: (8)                     (14, 17),
23: (8)                     (13, 11),
```

```
24: (8)                     (11, 10),
25: (8)                     (11, 12),
26: (8)                     (13, 18),
27: (8)                     (18, 19),
28: (8)                     (18, 20),
29: (4)                 ]
30: (4)                 G = nx.Graph(edge_list)
31: (4)                 b_T, b_C = nx.tree_broadcast_center(G)
32: (4)                 assert b_T == 6
33: (4)                 assert b_C == {13, 9}
34: (4)                 assert nx.tree_broadcast_time(G, 17) == 8
35: (4)                 assert nx.tree_broadcast_time(G, 3) == 9
36: (4)                 assert nx.tree_broadcast_time(G) == 10
37: (0)             def test_path_broadcast():
38: (4)                 for i in range(2, 12):
39: (8)                     G = nx.path_graph(i)
40: (8)                     b_T, b_C = nx.tree_broadcast_center(G)
41: (8)                     assert b_T == math.ceil(i / 2)
42: (8)                     assert b_C == {
43: (12)                        math.ceil(i / 2),
44: (12)                        math.floor(i / 2),
45: (12)                        math.ceil(i / 2 - 1),
46: (12)                        math.floor(i / 2 - 1),
47: (8)                     }
48: (8)                     assert nx.tree_broadcast_time(G) == i - 1
49: (0)             def test_empty_graph_broadcast():
50: (4)                 H = nx.empty_graph(1)
51: (4)                 b_T, b_C = nx.tree_broadcast_center(H)
52: (4)                 assert b_T == 0
53: (4)                 assert b_C == {0}
54: (4)                 assert nx.tree_broadcast_time(H) == 0
55: (0)             def test_star_broadcast():
56: (4)                 for i in range(4, 12):
57: (8)                     G = nx.star_graph(i)
58: (8)                     b_T, b_C = nx.tree_broadcast_center(G)
59: (8)                     assert b_T == i
60: (8)                     assert b_C == set(G.nodes())
61: (8)                     assert nx.tree_broadcast_time(G) == b_T
62: (0)             def test_binomial_tree_broadcast():
63: (4)                 for i in range(2, 8):
64: (8)                     G = nx.binomial_tree(i)
65: (8)                     b_T, b_C = nx.tree_broadcast_center(G)
66: (8)                     assert b_T == i
67: (8)                     assert b_C == {0, 2 ** (i - 1)}
68: (8)                     assert nx.tree_broadcast_time(G) == 2 * i - 1

----------------------------------------

File 310 - test_chains.py:

1: (0)              """Unit tests for the chain decomposition functions."""
2: (0)              from itertools import cycle, islice
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              def cycles(seq):
6: (4)                  """Yields cyclic permutations of the given sequence.
7: (4)                  For example::
8: (8)                      >>> list(cycles("abc"))
9: (8)                      [('a', 'b', 'c'), ('b', 'c', 'a'), ('c', 'a', 'b')]
10: (4)                 """
11: (4)                 n = len(seq)
12: (4)                 cycled_seq = cycle(seq)
13: (4)                 for x in seq:
14: (8)                     yield tuple(islice(cycled_seq, n))
15: (8)                     next(cycled_seq)
16: (0)             def cyclic_equals(seq1, seq2):
17: (4)                 """Decide whether two sequences are equal up to cyclic permutations.
18: (4)                 For example::
19: (8)                     >>> cyclic_equals("xyz", "zxy")
```

```
 20: (8)                    True
 21: (8)                    >>> cyclic_equals("xyz", "zyx")
 22: (8)                    False
 23: (4)                """
 24: (4)                seq2 = tuple(seq2)
 25: (4)                return any(x == tuple(seq2) for x in cycles(seq1))
 26: (0)            class TestChainDecomposition:
 27: (4)                """Unit tests for the chain decomposition function."""
 28: (4)                def assertContainsChain(self, chain, expected):
 29: (8)                    reversed_chain = list(reversed([tuple(reversed(e)) for e in chain]))
 30: (8)                    for candidate in expected:
 31: (12)                       if cyclic_equals(chain, candidate):
 32: (16)                           break
 33: (12)                       if cyclic_equals(reversed_chain, candidate):
 34: (16)                           break
 35: (8)                    else:
 36: (12)                       self.fail("chain not found")
 37: (4)                def test_decomposition(self):
 38: (8)                    edges = [
 39: (12)                       (1, 2),
 40: (12)                       (2, 3),
 41: (12)                       (3, 4),
 42: (12)                       (3, 5),
 43: (12)                       (5, 6),
 44: (12)                       (6, 7),
 45: (12)                       (7, 8),
 46: (12)                       (5, 9),
 47: (12)                       (9, 10),
 48: (12)                       (1, 3),
 49: (12)                       (1, 4),
 50: (12)                       (2, 5),
 51: (12)                       (5, 10),
 52: (12)                       (6, 8),
 53: (8)                    ]
 54: (8)                    G = nx.Graph(edges)
 55: (8)                    expected = [
 56: (12)                       [(1, 3), (3, 2), (2, 1)],
 57: (12)                       [(1, 4), (4, 3)],
 58: (12)                       [(2, 5), (5, 3)],
 59: (12)                       [(5, 10), (10, 9), (9, 5)],
 60: (12)                       [(6, 8), (8, 7), (7, 6)],
 61: (8)                    ]
 62: (8)                    chains = list(nx.chain_decomposition(G, root=1))
 63: (8)                    assert len(chains) == len(expected)
 64: (4)                def test_barbell_graph(self):
 65: (8)                    G = nx.barbell_graph(3, 0)
 66: (8)                    chains = list(nx.chain_decomposition(G, root=0))
 67: (8)                    expected = [[(0, 1), (1, 2), (2, 0)], [(3, 4), (4, 5), (5, 3)]]
 68: (8)                    assert len(chains) == len(expected)
 69: (8)                    for chain in chains:
 70: (12)                       self.assertContainsChain(chain, expected)
 71: (4)                def test_disconnected_graph(self):
 72: (8)                    """Test for a graph with multiple connected components."""
 73: (8)                    G = nx.barbell_graph(3, 0)
 74: (8)                    H = nx.barbell_graph(3, 0)
 75: (8)                    mapping = dict(zip(range(6), "abcdef"))
 76: (8)                    nx.relabel_nodes(H, mapping, copy=False)
 77: (8)                    G = nx.union(G, H)
 78: (8)                    chains = list(nx.chain_decomposition(G))
 79: (8)                    expected = [
 80: (12)                       [(0, 1), (1, 2), (2, 0)],
 81: (12)                       [(3, 4), (4, 5), (5, 3)],
 82: (12)                       [("a", "b"), ("b", "c"), ("c", "a")],
 83: (12)                       [("d", "e"), ("e", "f"), ("f", "d")],
 84: (8)                    ]
 85: (8)                    assert len(chains) == len(expected)
 86: (8)                    for chain in chains:
 87: (12)                       self.assertContainsChain(chain, expected)
 88: (4)                def test_disconnected_graph_root_node(self):
```

```
89: (8)                     """Test for a single component of a disconnected graph."""
90: (8)                     G = nx.barbell_graph(3, 0)
91: (8)                     H = nx.barbell_graph(3, 0)
92: (8)                     mapping = dict(zip(range(6), "abcdef"))
93: (8)                     nx.relabel_nodes(H, mapping, copy=False)
94: (8)                     G = nx.union(G, H)
95: (8)                     chains = list(nx.chain_decomposition(G, root="a"))
96: (8)                     expected = [
97: (12)                        [("a", "b"), ("b", "c"), ("c", "a")],
98: (12)                        [("d", "e"), ("e", "f"), ("f", "d")],
99: (8)                     ]
100: (8)                    assert len(chains) == len(expected)
101: (8)                    for chain in chains:
102: (12)                       self.assertContainsChain(chain, expected)
103: (4)                def test_chain_decomposition_root_not_in_G(self):
104: (8)                    """Test chain decomposition when root is not in graph"""
105: (8)                    G = nx.Graph()
106: (8)                    G.add_nodes_from([1, 2, 3])
107: (8)                    with pytest.raises(nx.NodeNotFound):
108: (12)                       nx.has_bridges(G, root=6)

----------------------------------------

File 311 - test_chordal.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestMCS:
4: (4)                  @classmethod
5: (4)                  def setup_class(cls):
6: (8)                      connected_chordal_G = nx.Graph()
7: (8)                      connected_chordal_G.add_edges_from(
8: (12)                         [
9: (16)                             (1, 2),
10: (16)                            (1, 3),
11: (16)                            (2, 3),
12: (16)                            (2, 4),
13: (16)                            (3, 4),
14: (16)                            (3, 5),
15: (16)                            (3, 6),
16: (16)                            (4, 5),
17: (16)                            (4, 6),
18: (16)                            (5, 6),
19: (12)                        ]
20: (8)                     )
21: (8)                     cls.connected_chordal_G = connected_chordal_G
22: (8)                     chordal_G = nx.Graph()
23: (8)                     chordal_G.add_edges_from(
24: (12)                        [
25: (16)                            (1, 2),
26: (16)                            (1, 3),
27: (16)                            (2, 3),
28: (16)                            (2, 4),
29: (16)                            (3, 4),
30: (16)                            (3, 5),
31: (16)                            (3, 6),
32: (16)                            (4, 5),
33: (16)                            (4, 6),
34: (16)                            (5, 6),
35: (16)                            (7, 8),
36: (12)                        ]
37: (8)                     )
38: (8)                     chordal_G.add_node(9)
39: (8)                     cls.chordal_G = chordal_G
40: (8)                     non_chordal_G = nx.Graph()
41: (8)                     non_chordal_G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 5), (3, 4),
(3, 5)])
42: (8)                     cls.non_chordal_G = non_chordal_G
43: (8)                     self_loop_G = nx.Graph()
```

```
44: (8)                     self_loop_G.add_edges_from([(1, 1)])
45: (8)                     cls.self_loop_G = self_loop_G
46: (4)                 @pytest.mark.parametrize("G", (nx.DiGraph(), nx.MultiGraph(),
nx.MultiDiGraph()))
47: (4)                 def test_is_chordal_not_implemented(self, G):
48: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
49: (12)                        nx.is_chordal(G)
50: (4)                 def test_is_chordal(self):
51: (8)                     assert not nx.is_chordal(self.non_chordal_G)
52: (8)                     assert nx.is_chordal(self.chordal_G)
53: (8)                     assert nx.is_chordal(self.connected_chordal_G)
54: (8)                     assert nx.is_chordal(nx.Graph())
55: (8)                     assert nx.is_chordal(nx.complete_graph(3))
56: (8)                     assert nx.is_chordal(nx.cycle_graph(3))
57: (8)                     assert not nx.is_chordal(nx.cycle_graph(5))
58: (8)                     assert nx.is_chordal(self.self_loop_G)
59: (4)                 def test_induced_nodes(self):
60: (8)                     G = nx.generators.classic.path_graph(10)
61: (8)                     Induced_nodes = nx.find_induced_nodes(G, 1, 9, 2)
62: (8)                     assert Induced_nodes == {1, 2, 3, 4, 5, 6, 7, 8, 9}
63: (8)                     pytest.raises(
64: (12)                        nx.NetworkXTreewidthBoundExceeded, nx.find_induced_nodes, G, 1, 9,
1
65: (8)                     )
66: (8)                     Induced_nodes = nx.find_induced_nodes(self.chordal_G, 1, 6)
67: (8)                     assert Induced_nodes == {1, 2, 4, 6}
68: (8)                     pytest.raises(nx.NetworkXError, nx.find_induced_nodes,
self.non_chordal_G, 1, 5)
69: (4)                 def test_graph_treewidth(self):
70: (8)                     with pytest.raises(nx.NetworkXError, match="Input graph is not
chordal"):
71: (12)                        nx.chordal_graph_treewidth(self.non_chordal_G)
72: (4)                 def test_chordal_find_cliques(self):
73: (8)                     cliques = {
74: (12)                        frozenset([9]),
75: (12)                        frozenset([7, 8]),
76: (12)                        frozenset([1, 2, 3]),
77: (12)                        frozenset([2, 3, 4]),
78: (12)                        frozenset([3, 4, 5, 6]),
79: (8)                     }
80: (8)                     assert set(nx.chordal_graph_cliques(self.chordal_G)) == cliques
81: (8)                     with pytest.raises(nx.NetworkXError, match="Input graph is not
chordal"):
82: (12)                        set(nx.chordal_graph_cliques(self.non_chordal_G))
83: (8)                     with pytest.raises(nx.NetworkXError, match="Input graph is not
chordal"):
84: (12)                        set(nx.chordal_graph_cliques(self.self_loop_G))
85: (4)                 def test_chordal_find_cliques_path(self):
86: (8)                     G = nx.path_graph(10)
87: (8)                     cliqueset = nx.chordal_graph_cliques(G)
88: (8)                     for u, v in G.edges():
89: (12)                        assert frozenset([u, v]) in cliqueset or frozenset([v, u]) in
cliqueset
90: (4)                 def test_chordal_find_cliquesCC(self):
91: (8)                     cliques = {frozenset([1, 2, 3]), frozenset([2, 3, 4]), frozenset([3,
4, 5, 6])}
92: (8)                     cgc = nx.chordal_graph_cliques
93: (8)                     assert set(cgc(self.connected_chordal_G)) == cliques
94: (4)                 def test_complete_to_chordal_graph(self):
95: (8)                     fgrg = nx.fast_gnp_random_graph
96: (8)                     test_graphs = [
97: (12)                        nx.barbell_graph(6, 2),
98: (12)                        nx.cycle_graph(15),
99: (12)                        nx.wheel_graph(20),
100: (12)                       nx.grid_graph([10, 4]),
101: (12)                       nx.ladder_graph(15),
102: (12)                       nx.star_graph(5),
103: (12)                       nx.bull_graph(),
104: (12)                       fgrg(20, 0.3, seed=1),
```

```
105: (8)                    ]
106: (8)                    for G in test_graphs:
107: (12)                       H, a = nx.complete_to_chordal_graph(G)
108: (12)                       assert nx.is_chordal(H)
109: (12)                       assert len(a) == H.number_of_nodes()
110: (12)                       if nx.is_chordal(G):
111: (16)                           assert G.number_of_edges() == H.number_of_edges()
112: (16)                           assert set(a.values()) == {0}
113: (12)                       else:
114: (16)                           assert len(set(a.values())) == H.number_of_nodes()
```

File 312 - test_clique.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import convert_node_labels_to_integers as cnlti
4: (0)              class TestCliques:
5: (4)                  def setup_method(self):
6: (8)                      z = [3, 4, 3, 4, 2, 4, 2, 1, 1, 1, 1]
7: (8)                      self.G = cnlti(nx.generators.havel_hakimi_graph(z), first_label=1)
8: (8)                      self.cl = list(nx.find_cliques(self.G))
9: (8)                      H = nx.complete_graph(6)
10: (8)                     H = nx.relabel_nodes(H, {i: i + 1 for i in range(6)})
11: (8)                     H.remove_edges_from([(2, 6), (2, 5), (2, 4), (1, 3), (5, 3)])
12: (8)                     self.H = H
13: (4)                 def test_find_cliques1(self):
14: (8)                     cl = list(nx.find_cliques(self.G))
15: (8)                     rcl = nx.find_cliques_recursive(self.G)
16: (8)                     expected = [[2, 6, 1, 3], [2, 6, 4], [5, 4, 7], [8, 9], [10, 11]]
17: (8)                     assert sorted(map(sorted, cl)) == sorted(map(sorted, rcl))
18: (8)                     assert sorted(map(sorted, cl)) == sorted(map(sorted, expected))
19: (4)                 def test_selfloops(self):
20: (8)                     self.G.add_edge(1, 1)
21: (8)                     cl = list(nx.find_cliques(self.G))
22: (8)                     rcl = list(nx.find_cliques_recursive(self.G))
23: (8)                     assert set(map(frozenset, cl)) == set(map(frozenset, rcl))
24: (8)                     answer = [{2, 6, 1, 3}, {2, 6, 4}, {5, 4, 7}, {8, 9}, {10, 11}]
25: (8)                     assert len(answer) == len(cl)
26: (8)                     assert all(set(c) in answer for c in cl)
27: (4)                 def test_find_cliques2(self):
28: (8)                     hcl = list(nx.find_cliques(self.H))
29: (8)                     assert sorted(map(sorted, hcl)) == [[1, 2], [1, 4, 5, 6], [2, 3], [3,
4, 6]]
30: (4)                 def test_find_cliques3(self):
31: (8)                     cl = list(nx.find_cliques(self.G, [2]))
32: (8)                     rcl = nx.find_cliques_recursive(self.G, [2])
33: (8)                     expected = [[2, 6, 1, 3], [2, 6, 4]]
34: (8)                     assert sorted(map(sorted, rcl)) == sorted(map(sorted, expected))
35: (8)                     assert sorted(map(sorted, cl)) == sorted(map(sorted, expected))
36: (8)                     cl = list(nx.find_cliques(self.G, [2, 3]))
37: (8)                     rcl = nx.find_cliques_recursive(self.G, [2, 3])
38: (8)                     expected = [[2, 6, 1, 3]]
39: (8)                     assert sorted(map(sorted, rcl)) == sorted(map(sorted, expected))
40: (8)                     assert sorted(map(sorted, cl)) == sorted(map(sorted, expected))
41: (8)                     cl = list(nx.find_cliques(self.G, [2, 6, 4]))
42: (8)                     rcl = nx.find_cliques_recursive(self.G, [2, 6, 4])
43: (8)                     expected = [[2, 6, 4]]
44: (8)                     assert sorted(map(sorted, rcl)) == sorted(map(sorted, expected))
45: (8)                     assert sorted(map(sorted, cl)) == sorted(map(sorted, expected))
46: (8)                     cl = list(nx.find_cliques(self.G, [2, 6, 4]))
47: (8)                     rcl = nx.find_cliques_recursive(self.G, [2, 6, 4])
48: (8)                     expected = [[2, 6, 4]]
49: (8)                     assert sorted(map(sorted, rcl)) == sorted(map(sorted, expected))
50: (8)                     assert sorted(map(sorted, cl)) == sorted(map(sorted, expected))
51: (8)                     with pytest.raises(ValueError):
52: (12)                        list(nx.find_cliques(self.G, [2, 6, 4, 1]))
53: (8)                     with pytest.raises(ValueError):
```

```
(12)                        list(nx.find_cliques_recursive(self.G, [2, 6, 4, 1]))
(4)                 def test_number_of_cliques(self):
(8)                     G = self.G
(8)                     assert nx.number_of_cliques(G, 1) == 1
(8)                     assert list(nx.number_of_cliques(G, [1]).values()) == [1]
(8)                     assert list(nx.number_of_cliques(G, [1, 2]).values()) == [1, 2]
(8)                     assert nx.number_of_cliques(G, [1, 2]) == {1: 1, 2: 2}
(8)                     assert nx.number_of_cliques(G, 2) == 2
(8)                     assert nx.number_of_cliques(G) == {
(12)                        1: 1,
(12)                        2: 2,
(12)                        3: 1,
(12)                        4: 2,
(12)                        5: 1,
(12)                        6: 2,
(12)                        7: 1,
(12)                        8: 1,
(12)                        9: 1,
(12)                        10: 1,
(12)                        11: 1,
(8)                     }
(8)                     assert nx.number_of_cliques(G, nodes=list(G)) == {
(12)                        1: 1,
(12)                        2: 2,
(12)                        3: 1,
(12)                        4: 2,
(12)                        5: 1,
(12)                        6: 2,
(12)                        7: 1,
(12)                        8: 1,
(12)                        9: 1,
(12)                        10: 1,
(12)                        11: 1,
(8)                     }
(8)                     assert nx.number_of_cliques(G, nodes=[2, 3, 4]) == {2: 2, 3: 1, 4: 2}
(8)                     assert nx.number_of_cliques(G, cliques=self.cl) == {
(12)                        1: 1,
(12)                        2: 2,
(12)                        3: 1,
(12)                        4: 2,
(12)                        5: 1,
(12)                        6: 2,
(12)                        7: 1,
(12)                        8: 1,
(12)                        9: 1,
(12)                        10: 1,
(12)                        11: 1,
(8)                     }
(8)                     assert nx.number_of_cliques(G, list(G), cliques=self.cl) == {
(12)                        1: 1,
(12)                        2: 2,
(12)                        3: 1,
(12)                        4: 2,
(12)                        5: 1,
(12)                        6: 2,
(12)                        7: 1,
(12)                        8: 1,
(12)                        9: 1,
(12)                        10: 1,
(12)                        11: 1,
(8)                     }
(4)                 def test_node_clique_number(self):
(8)                     G = self.G
(8)                     assert nx.node_clique_number(G, 1) == 4
(8)                     assert list(nx.node_clique_number(G, [1]).values()) == [4]
(8)                     assert list(nx.node_clique_number(G, [1, 2]).values()) == [4, 4]
(8)                     assert nx.node_clique_number(G, [1, 2]) == {1: 4, 2: 4}
(8)                     assert nx.node_clique_number(G, 1) == 4
(8)                     assert nx.node_clique_number(G) == {
```

```
123: (12)                        1: 4,
124: (12)                        2: 4,
125: (12)                        3: 4,
126: (12)                        4: 3,
127: (12)                        5: 3,
128: (12)                        6: 4,
129: (12)                        7: 3,
130: (12)                        8: 2,
131: (12)                        9: 2,
132: (12)                        10: 2,
133: (12)                        11: 2,
134: (8)                    }
135: (8)                    assert nx.node_clique_number(G, cliques=self.cl) == {
136: (12)                        1: 4,
137: (12)                        2: 4,
138: (12)                        3: 4,
139: (12)                        4: 3,
140: (12)                        5: 3,
141: (12)                        6: 4,
142: (12)                        7: 3,
143: (12)                        8: 2,
144: (12)                        9: 2,
145: (12)                        10: 2,
146: (12)                        11: 2,
147: (8)                    }
148: (8)                    assert nx.node_clique_number(G, [1, 2], cliques=self.cl) == {1: 4, 2:
4}
149: (8)                    assert nx.node_clique_number(G, 1, cliques=self.cl) == 4
150: (4)                def test_make_clique_bipartite(self):
151: (8)                    G = self.G
152: (8)                    B = nx.make_clique_bipartite(G)
153: (8)                    assert sorted(B) == [-5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11]
154: (8)                    H = nx.projected_graph(B, range(1, 12))
155: (8)                    assert H.adj == G.adj
156: (8)                    H1 = nx.projected_graph(B, range(-5, 0))
157: (8)                    H1 = nx.relabel_nodes(H1, {-v: v for v in range(1, 6)})
158: (8)                    assert sorted(H1) == [1, 2, 3, 4, 5]
159: (4)                def test_make_max_clique_graph(self):
160: (8)                    """Tests that the maximal clique graph is the same as the bipartite
161: (8)                    clique graph after being projected onto the nodes representing the
162: (8)                    cliques.
163: (8)                    """
164: (8)                    G = self.G
165: (8)                    B = nx.make_clique_bipartite(G)
166: (8)                    H1 = nx.projected_graph(B, range(-5, 0))
167: (8)                    H1 = nx.relabel_nodes(H1, {-v: v - 1 for v in range(1, 6)})
168: (8)                    H2 = nx.make_max_clique_graph(G)
169: (8)                    assert H1.adj == H2.adj
170: (4)                def test_directed(self):
171: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
172: (12)                       next(nx.find_cliques(nx.DiGraph()))
173: (4)                def test_find_cliques_trivial(self):
174: (8)                    G = nx.Graph()
175: (8)                    assert sorted(nx.find_cliques(G)) == []
176: (8)                    assert sorted(nx.find_cliques_recursive(G)) == []
177: (4)                def test_make_max_clique_graph_create_using(self):
178: (8)                    G = nx.Graph([(1, 2), (3, 1), (4, 1), (5, 6)])
179: (8)                    E = nx.Graph([(0, 1), (0, 2), (1, 2)])
180: (8)                    E.add_node(3)
181: (8)                    assert nx.is_isomorphic(nx.make_max_clique_graph(G,
create_using=nx.Graph), E)
182: (0)            class TestEnumerateAllCliques:
183: (4)                def test_paper_figure_4(self):
184: (8)                    G = nx.Graph()
185: (8)                    edges_fig_4 = [
186: (12)                       ("a", "b"),
187: (12)                       ("a", "c"),
188: (12)                       ("a", "d"),
```

```
189: (12)                        ("a", "e"),
190: (12)                        ("b", "c"),
191: (12)                        ("b", "d"),
192: (12)                        ("b", "e"),
193: (12)                        ("c", "d"),
194: (12)                        ("c", "e"),
195: (12)                        ("d", "e"),
196: (12)                        ("f", "b"),
197: (12)                        ("f", "c"),
198: (12)                        ("f", "g"),
199: (12)                        ("g", "f"),
200: (12)                        ("g", "c"),
201: (12)                        ("g", "d"),
202: (12)                        ("g", "e"),
203: (8)                    ]
204: (8)                    G.add_edges_from(edges_fig_4)
205: (8)                    cliques = list(nx.enumerate_all_cliques(G))
206: (8)                    clique_sizes = list(map(len, cliques))
207: (8)                    assert sorted(clique_sizes) == clique_sizes
208: (8)                    expected_cliques = [
209: (12)                        ["a"],
210: (12)                        ["b"],
211: (12)                        ["c"],
212: (12)                        ["d"],
213: (12)                        ["e"],
214: (12)                        ["f"],
215: (12)                        ["g"],
216: (12)                        ["a", "b"],
217: (12)                        ["a", "b", "d"],
218: (12)                        ["a", "b", "d", "e"],
219: (12)                        ["a", "b", "e"],
220: (12)                        ["a", "c"],
221: (12)                        ["a", "c", "d"],
222: (12)                        ["a", "c", "d", "e"],
223: (12)                        ["a", "c", "e"],
224: (12)                        ["a", "d"],
225: (12)                        ["a", "d", "e"],
226: (12)                        ["a", "e"],
227: (12)                        ["b", "c"],
228: (12)                        ["b", "c", "d"],
229: (12)                        ["b", "c", "d", "e"],
230: (12)                        ["b", "c", "e"],
231: (12)                        ["b", "c", "f"],
232: (12)                        ["b", "d"],
233: (12)                        ["b", "d", "e"],
234: (12)                        ["b", "e"],
235: (12)                        ["b", "f"],
236: (12)                        ["c", "d"],
237: (12)                        ["c", "d", "e"],
238: (12)                        ["c", "d", "e", "g"],
239: (12)                        ["c", "d", "g"],
240: (12)                        ["c", "e"],
241: (12)                        ["c", "e", "g"],
242: (12)                        ["c", "f"],
243: (12)                        ["c", "f", "g"],
244: (12)                        ["c", "g"],
245: (12)                        ["d", "e"],
246: (12)                        ["d", "e", "g"],
247: (12)                        ["d", "g"],
248: (12)                        ["e", "g"],
249: (12)                        ["f", "g"],
250: (12)                        ["a", "b", "c"],
251: (12)                        ["a", "b", "c", "d"],
252: (12)                        ["a", "b", "c", "d", "e"],
253: (12)                        ["a", "b", "c", "e"],
254: (8)                    ]
255: (8)                    assert sorted(map(sorted, cliques)) == sorted(map(sorted,
expected_cliques))
```

```
----------------------------------------

File 313 - test_cluster.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestTriangles:
4: (4)                  def test_empty(self):
5: (8)                      G = nx.Graph()
6: (8)                      assert list(nx.triangles(G).values()) == []
7: (4)                  def test_path(self):
8: (8)                      G = nx.path_graph(10)
9: (8)                      assert list(nx.triangles(G).values()) == [0, 0, 0, 0, 0, 0, 0, 0, 0,
0]
10: (8)                     assert nx.triangles(G) == {
11: (12)                        0: 0,
12: (12)                        1: 0,
13: (12)                        2: 0,
14: (12)                        3: 0,
15: (12)                        4: 0,
16: (12)                        5: 0,
17: (12)                        6: 0,
18: (12)                        7: 0,
19: (12)                        8: 0,
20: (12)                        9: 0,
21: (8)                     }
22: (4)                 def test_cubical(self):
23: (8)                     G = nx.cubical_graph()
24: (8)                     assert list(nx.triangles(G).values()) == [0, 0, 0, 0, 0, 0, 0, 0]
25: (8)                     assert nx.triangles(G, 1) == 0
26: (8)                     assert list(nx.triangles(G, [1, 2]).values()) == [0, 0]
27: (8)                     assert nx.triangles(G, 1) == 0
28: (8)                     assert nx.triangles(G, [1, 2]) == {1: 0, 2: 0}
29: (4)                 def test_k5(self):
30: (8)                     G = nx.complete_graph(5)
31: (8)                     assert list(nx.triangles(G).values()) == [6, 6, 6, 6, 6]
32: (8)                     assert sum(nx.triangles(G).values()) / 3 == 10
33: (8)                     assert nx.triangles(G, 1) == 6
34: (8)                     G.remove_edge(1, 2)
35: (8)                     assert list(nx.triangles(G).values()) == [5, 3, 3, 5, 5]
36: (8)                     assert nx.triangles(G, 1) == 3
37: (8)                     G.add_edge(3, 3)  # ignore self-edges
38: (8)                     assert list(nx.triangles(G).values()) == [5, 3, 3, 5, 5]
39: (8)                     assert nx.triangles(G, 3) == 5
40: (0)             class TestDirectedClustering:
41: (4)                 def test_clustering(self):
42: (8)                     G = nx.DiGraph()
43: (8)                     assert list(nx.clustering(G).values()) == []
44: (8)                     assert nx.clustering(G) == {}
45: (4)                 def test_path(self):
46: (8)                     G = nx.path_graph(10, create_using=nx.DiGraph())
47: (8)                     assert list(nx.clustering(G).values()) == [
48: (12)                        0,
49: (12)                        0,
50: (12)                        0,
51: (12)                        0,
52: (12)                        0,
53: (12)                        0,
54: (12)                        0,
55: (12)                        0,
56: (12)                        0,
57: (12)                        0,
58: (8)                     ]
59: (8)                     assert nx.clustering(G) == {
60: (12)                        0: 0,
61: (12)                        1: 0,
62: (12)                        2: 0,
63: (12)                        3: 0,
64: (12)                        4: 0,
```

```
65: (12)                        5: 0,
66: (12)                        6: 0,
67: (12)                        7: 0,
68: (12)                        8: 0,
69: (12)                        9: 0,
70: (8)                     }
71: (8)                     assert nx.clustering(G, 0) == 0
72: (4)                 def test_k5(self):
73: (8)                     G = nx.complete_graph(5, create_using=nx.DiGraph())
74: (8)                     assert list(nx.clustering(G).values()) == [1, 1, 1, 1, 1]
75: (8)                     assert nx.average_clustering(G) == 1
76: (8)                     G.remove_edge(1, 2)
77: (8)                     assert list(nx.clustering(G).values()) == [
78: (12)                        11 / 12,
79: (12)                        1,
80: (12)                        1,
81: (12)                        11 / 12,
82: (12)                        11 / 12,
83: (8)                     ]
84: (8)                     assert nx.clustering(G, [1, 4]) == {1: 1, 4: 11 / 12}
85: (8)                     G.remove_edge(2, 1)
86: (8)                     assert list(nx.clustering(G).values()) == [
87: (12)                        5 / 6,
88: (12)                        1,
89: (12)                        1,
90: (12)                        5 / 6,
91: (12)                        5 / 6,
92: (8)                     ]
93: (8)                     assert nx.clustering(G, [1, 4]) == {1: 1, 4: 0.8333333333333337}
94: (8)                     assert nx.clustering(G, 4) == 5 / 6
95: (4)                 def test_triangle_and_edge(self):
96: (8)                     G = nx.cycle_graph(3, create_using=nx.DiGraph())
97: (8)                     G.add_edge(0, 4)
98: (8)                     assert nx.clustering(G)[0] == 1 / 6
99: (0)             class TestDirectedWeightedClustering:
100: (4)                @classmethod
101: (4)                def setup_class(cls):
102: (8)                    global np
103: (8)                    np = pytest.importorskip("numpy")
104: (4)                def test_clustering(self):
105: (8)                    G = nx.DiGraph()
106: (8)                    assert list(nx.clustering(G, weight="weight").values()) == []
107: (8)                    assert nx.clustering(G) == {}
108: (4)                def test_path(self):
109: (8)                    G = nx.path_graph(10, create_using=nx.DiGraph())
110: (8)                    assert list(nx.clustering(G, weight="weight").values()) == [
111: (12)                       0,
112: (12)                       0,
113: (12)                       0,
114: (12)                       0,
115: (12)                       0,
116: (12)                       0,
117: (12)                       0,
118: (12)                       0,
119: (12)                       0,
120: (12)                       0,
121: (8)                    ]
122: (8)                    assert nx.clustering(G, weight="weight") == {
123: (12)                       0: 0,
124: (12)                       1: 0,
125: (12)                       2: 0,
126: (12)                       3: 0,
127: (12)                       4: 0,
128: (12)                       5: 0,
129: (12)                       6: 0,
130: (12)                       7: 0,
131: (12)                       8: 0,
132: (12)                       9: 0,
133: (8)                    }
```

```
(4)                 def test_k5(self):
(8)                     G = nx.complete_graph(5, create_using=nx.DiGraph())
(8)                     assert list(nx.clustering(G, weight="weight").values()) == [1, 1, 1,
1, 1]
(8)                     assert nx.average_clustering(G, weight="weight") == 1
(8)                     G.remove_edge(1, 2)
(8)                     assert list(nx.clustering(G, weight="weight").values()) == [
(12)                        11 / 12,
(12)                        1,
(12)                        1,
(12)                        11 / 12,
(12)                        11 / 12,
(8)                     ]
(8)                     assert nx.clustering(G, [1, 4], weight="weight") == {1: 1, 4: 11 / 12}
(8)                     G.remove_edge(2, 1)
(8)                     assert list(nx.clustering(G, weight="weight").values()) == [
(12)                        5 / 6,
(12)                        1,
(12)                        1,
(12)                        5 / 6,
(12)                        5 / 6,
(8)                     ]
(8)                     assert nx.clustering(G, [1, 4], weight="weight") == {
(12)                        1: 1,
(12)                        4: 0.83333333333333337,
(8)                     }
(4)                 def test_triangle_and_edge(self):
(8)                     G = nx.cycle_graph(3, create_using=nx.DiGraph())
(8)                     G.add_edge(0, 4, weight=2)
(8)                     assert nx.clustering(G)[0] == 1 / 6
(8)                     np.testing.assert_allclose(nx.clustering(G, weight="weight")[0], 1 /
12)
(8)                     np.testing.assert_allclose(nx.clustering(G, 0, weight="weight"), 1 /
12)
(0)             class TestWeightedClustering:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     global np
(8)                     np = pytest.importorskip("numpy")
(4)                 def test_clustering(self):
(8)                     G = nx.Graph()
(8)                     assert list(nx.clustering(G, weight="weight").values()) == []
(8)                     assert nx.clustering(G) == {}
(4)                 def test_path(self):
(8)                     G = nx.path_graph(10)
(8)                     assert list(nx.clustering(G, weight="weight").values()) == [
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(12)                        0,
(8)                     ]
(8)                     assert nx.clustering(G, weight="weight") == {
(12)                        0: 0,
(12)                        1: 0,
(12)                        2: 0,
(12)                        3: 0,
(12)                        4: 0,
(12)                        5: 0,
(12)                        6: 0,
(12)                        7: 0,
(12)                        8: 0,
(12)                        9: 0,
(8)                     }
```

```
(4)     def test_cubical(self):
(8)         G = nx.cubical_graph()
(8)         assert list(nx.clustering(G, weight="weight").values()) == [
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(8)         ]
(8)         assert nx.clustering(G, 1) == 0
(8)         assert list(nx.clustering(G, [1, 2], weight="weight").values()) == [0,
0]
(8)         assert nx.clustering(G, 1, weight="weight") == 0
(8)         assert nx.clustering(G, [1, 2], weight="weight") == {1: 0, 2: 0}
(4)     def test_k5(self):
(8)         G = nx.complete_graph(5)
(8)         assert list(nx.clustering(G, weight="weight").values()) == [1, 1, 1,
1, 1]
(8)         assert nx.average_clustering(G, weight="weight") == 1
(8)         G.remove_edge(1, 2)
(8)         assert list(nx.clustering(G, weight="weight").values()) == [
(12)            5 / 6,
(12)            1,
(12)            1,
(12)            5 / 6,
(12)            5 / 6,
(8)         ]
(8)         assert nx.clustering(G, [1, 4], weight="weight") == {
(12)            1: 1,
(12)            4: 0.83333333333333337,
(8)         }
(4)     def test_triangle_and_edge(self):
(8)         G = nx.cycle_graph(3)
(8)         G.add_edge(0, 4, weight=2)
(8)         assert nx.clustering(G)[0] == 1 / 3
(8)         np.testing.assert_allclose(nx.clustering(G, weight="weight")[0], 1 /
6)
(8)         np.testing.assert_allclose(nx.clustering(G, 0, weight="weight"), 1 /
6)
(4)     def test_triangle_and_signed_edge(self):
(8)         G = nx.cycle_graph(3)
(8)         G.add_edge(0, 1, weight=-1)
(8)         G.add_edge(3, 0, weight=0)
(8)         assert nx.clustering(G)[0] == 1 / 3
(8)         assert nx.clustering(G, weight="weight")[0] == -1 / 3
(0)     class TestClustering:
(4)         @classmethod
(4)         def setup_class(cls):
(8)             pytest.importorskip("numpy")
(4)         def test_clustering(self):
(8)             G = nx.Graph()
(8)             assert list(nx.clustering(G).values()) == []
(8)             assert nx.clustering(G) == {}
(4)         def test_path(self):
(8)             G = nx.path_graph(10)
(8)             assert list(nx.clustering(G).values()) == [
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
(12)            0,
```

```
265: (8)                    ]
266: (8)                    assert nx.clustering(G) == {
267: (12)                       0: 0,
268: (12)                       1: 0,
269: (12)                       2: 0,
270: (12)                       3: 0,
271: (12)                       4: 0,
272: (12)                       5: 0,
273: (12)                       6: 0,
274: (12)                       7: 0,
275: (12)                       8: 0,
276: (12)                       9: 0,
277: (8)                    }
278: (4)                def test_cubical(self):
279: (8)                    G = nx.cubical_graph()
280: (8)                    assert list(nx.clustering(G).values()) == [0, 0, 0, 0, 0, 0, 0, 0]
281: (8)                    assert nx.clustering(G, 1) == 0
282: (8)                    assert list(nx.clustering(G, [1, 2]).values()) == [0, 0]
283: (8)                    assert nx.clustering(G, 1) == 0
284: (8)                    assert nx.clustering(G, [1, 2]) == {1: 0, 2: 0}
285: (4)                def test_k5(self):
286: (8)                    G = nx.complete_graph(5)
287: (8)                    assert list(nx.clustering(G).values()) == [1, 1, 1, 1, 1]
288: (8)                    assert nx.average_clustering(G) == 1
289: (8)                    G.remove_edge(1, 2)
290: (8)                    assert list(nx.clustering(G).values()) == [
291: (12)                       5 / 6,
292: (12)                       1,
293: (12)                       1,
294: (12)                       5 / 6,
295: (12)                       5 / 6,
296: (8)                    ]
297: (8)                    assert nx.clustering(G, [1, 4]) == {1: 1, 4: 0.83333333333333337}
298: (4)                def test_k5_signed(self):
299: (8)                    G = nx.complete_graph(5)
300: (8)                    assert list(nx.clustering(G).values()) == [1, 1, 1, 1, 1]
301: (8)                    assert nx.average_clustering(G) == 1
302: (8)                    G.remove_edge(1, 2)
303: (8)                    G.add_edge(0, 1, weight=-1)
304: (8)                    assert list(nx.clustering(G, weight="weight").values()) == [
305: (12)                       1 / 6,
306: (12)                       -1 / 3,
307: (12)                       1,
308: (12)                       3 / 6,
309: (12)                       3 / 6,
310: (8)                    ]
311: (0)            class TestTransitivity:
312: (4)                def test_transitivity(self):
313: (8)                    G = nx.Graph()
314: (8)                    assert nx.transitivity(G) == 0
315: (4)                def test_path(self):
316: (8)                    G = nx.path_graph(10)
317: (8)                    assert nx.transitivity(G) == 0
318: (4)                def test_cubical(self):
319: (8)                    G = nx.cubical_graph()
320: (8)                    assert nx.transitivity(G) == 0
321: (4)                def test_k5(self):
322: (8)                    G = nx.complete_graph(5)
323: (8)                    assert nx.transitivity(G) == 1
324: (8)                    G.remove_edge(1, 2)
325: (8)                    assert nx.transitivity(G) == 0.875
326: (0)            class TestSquareClustering:
327: (4)                def test_clustering(self):
328: (8)                    G = nx.Graph()
329: (8)                    assert list(nx.square_clustering(G).values()) == []
330: (8)                    assert nx.square_clustering(G) == {}
331: (4)                def test_path(self):
332: (8)                    G = nx.path_graph(10)
333: (8)                    assert list(nx.square_clustering(G).values()) == [
```

```
334: (12)                        0,
335: (12)                        0,
336: (12)                        0,
337: (12)                        0,
338: (12)                        0,
339: (12)                        0,
340: (12)                        0,
341: (12)                        0,
342: (12)                        0,
343: (12)                        0,
344: (8)                    ]
345: (8)                    assert nx.square_clustering(G) == {
346: (12)                        0: 0,
347: (12)                        1: 0,
348: (12)                        2: 0,
349: (12)                        3: 0,
350: (12)                        4: 0,
351: (12)                        5: 0,
352: (12)                        6: 0,
353: (12)                        7: 0,
354: (12)                        8: 0,
355: (12)                        9: 0,
356: (8)                    }
357: (4)                def test_cubical(self):
358: (8)                    G = nx.cubical_graph()
359: (8)                    assert list(nx.square_clustering(G).values()) == [
360: (12)                        1 / 3,
361: (12)                        1 / 3,
362: (12)                        1 / 3,
363: (12)                        1 / 3,
364: (12)                        1 / 3,
365: (12)                        1 / 3,
366: (12)                        1 / 3,
367: (12)                        1 / 3,
368: (8)                    ]
369: (8)                    assert list(nx.square_clustering(G, [1, 2]).values()) == [1 / 3, 1 /
3]
370: (8)                    assert nx.square_clustering(G, [1])[1] == 1 / 3
371: (8)                    assert nx.square_clustering(G, 1) == 1 / 3
372: (8)                    assert nx.square_clustering(G, [1, 2]) == {1: 1 / 3, 2: 1 / 3}
373: (4)                def test_k5(self):
374: (8)                    G = nx.complete_graph(5)
375: (8)                    assert list(nx.square_clustering(G).values()) == [1, 1, 1, 1, 1]
376: (4)                def test_bipartite_k5(self):
377: (8)                    G = nx.complete_bipartite_graph(5, 5)
378: (8)                    assert list(nx.square_clustering(G).values()) == [1, 1, 1, 1, 1, 1, 1,
1, 1, 1]
379: (4)                def test_lind_square_clustering(self):
380: (8)                    """Test C4 for figure 1 Lind et al (2005)"""
381: (8)                    G = nx.Graph(
382: (12)                        [
383: (16)                            (1, 2),
384: (16)                            (1, 3),
385: (16)                            (1, 6),
386: (16)                            (1, 7),
387: (16)                            (2, 4),
388: (16)                            (2, 5),
389: (16)                            (3, 4),
390: (16)                            (3, 5),
391: (16)                            (6, 7),
392: (16)                            (7, 8),
393: (16)                            (6, 8),
394: (16)                            (7, 9),
395: (16)                            (7, 10),
396: (16)                            (6, 11),
397: (16)                            (6, 12),
398: (16)                            (2, 13),
399: (16)                            (2, 14),
400: (16)                            (3, 15),
```

```
(16)                            (3, 16),
(12)                        ]
(8)                     )
(8)                     G1 = G.subgraph([1, 2, 3, 4, 5, 13, 14, 15, 16])
(8)                     G2 = G.subgraph([1, 6, 7, 8, 9, 10, 11, 12])
(8)                     assert nx.square_clustering(G, [1])[1] == 3 / 43
(8)                     assert nx.square_clustering(G1, [1])[1] == 2 / 6
(8)                     assert nx.square_clustering(G2, [1])[1] == 1 / 5
(4)                 def test_peng_square_clustering(self):
(8)                     """Test eq2 for figure 1 Peng et al (2008)"""
(8)                     G = nx.Graph([(1, 2), (1, 3), (2, 4), (3, 4), (3, 5), (3, 6)])
(8)                     assert nx.square_clustering(G, [1])[1] == 1 / 3
(4)                 def test_self_loops_square_clustering(self):
(8)                     G = nx.path_graph(5)
(8)                     assert nx.square_clustering(G) == {0: 0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0}
(8)                     G.add_edges_from([(0, 0), (1, 1), (2, 2)])
(8)                     assert nx.square_clustering(G) == {0: 1, 1: 0.5, 2: 0.2, 3: 0.0, 4: 0}
(0)             class TestAverageClustering:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     pytest.importorskip("numpy")
(4)                 def test_empty(self):
(8)                     G = nx.Graph()
(8)                     with pytest.raises(ZeroDivisionError):
(12)                        nx.average_clustering(G)
(4)                 def test_average_clustering(self):
(8)                     G = nx.cycle_graph(3)
(8)                     G.add_edge(2, 3)
(8)                     assert nx.average_clustering(G) == (1 + 1 + 1 / 3) / 4
(8)                     assert nx.average_clustering(G, count_zeros=True) == (1 + 1 + 1 / 3) /
4
(8)                     assert nx.average_clustering(G, count_zeros=False) == (1 + 1 + 1 / 3)
/ 3
(8)                     assert nx.average_clustering(G, [1, 2, 3]) == (1 + 1 / 3) / 3
(8)                     assert nx.average_clustering(G, [1, 2, 3], count_zeros=True) == (1 + 1
/ 3) / 3
(8)                     assert nx.average_clustering(G, [1, 2, 3], count_zeros=False) == (1 +
1 / 3) / 2
(4)                 def test_average_clustering_signed(self):
(8)                     G = nx.cycle_graph(3)
(8)                     G.add_edge(2, 3)
(8)                     G.add_edge(0, 1, weight=-1)
(8)                     assert nx.average_clustering(G, weight="weight") == (-1 - 1 - 1 / 3) /
4
(8)                     assert (
(12)                        nx.average_clustering(G, weight="weight", count_zeros=True)
(12)                        == (-1 - 1 - 1 / 3) / 4
(8)                     )
(8)                     assert (
(12)                        nx.average_clustering(G, weight="weight", count_zeros=False)
(12)                        == (-1 - 1 - 1 / 3) / 3
(8)                     )
(0)             class TestDirectedAverageClustering:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     pytest.importorskip("numpy")
(4)                 def test_empty(self):
(8)                     G = nx.DiGraph()
(8)                     with pytest.raises(ZeroDivisionError):
(12)                        nx.average_clustering(G)
(4)                 def test_average_clustering(self):
(8)                     G = nx.cycle_graph(3, create_using=nx.DiGraph())
(8)                     G.add_edge(2, 3)
(8)                     assert nx.average_clustering(G) == (1 + 1 + 1 / 3) / 8
(8)                     assert nx.average_clustering(G, count_zeros=True) == (1 + 1 + 1 / 3) /
8
(8)                     assert nx.average_clustering(G, count_zeros=False) == (1 + 1 + 1 / 3)
/ 6
(8)                     assert nx.average_clustering(G, [1, 2, 3]) == (1 + 1 / 3) / 6
```

```
463: (8)                    assert nx.average_clustering(G, [1, 2, 3], count_zeros=True) == (1 + 1
/ 3) / 6
464: (8)                    assert nx.average_clustering(G, [1, 2, 3], count_zeros=False) == (1 +
1 / 3) / 4
465: (0)            class TestGeneralizedDegree:
466: (4)                def test_generalized_degree(self):
467: (8)                    G = nx.Graph()
468: (8)                    assert nx.generalized_degree(G) == {}
469: (4)                def test_path(self):
470: (8)                    G = nx.path_graph(5)
471: (8)                    assert nx.generalized_degree(G, 0) == {0: 1}
472: (8)                    assert nx.generalized_degree(G, 1) == {0: 2}
473: (4)                def test_cubical(self):
474: (8)                    G = nx.cubical_graph()
475: (8)                    assert nx.generalized_degree(G, 0) == {0: 3}
476: (4)                def test_k5(self):
477: (8)                    G = nx.complete_graph(5)
478: (8)                    assert nx.generalized_degree(G, 0) == {3: 4}
479: (8)                    G.remove_edge(0, 1)
480: (8)                    assert nx.generalized_degree(G, 0) == {2: 3}
481: (8)                    assert nx.generalized_degree(G, [1, 2]) == {1: {2: 3}, 2: {2: 2, 3:
2}}
482: (8)                    assert nx.generalized_degree(G) == {
483: (12)                       0: {2: 3},
484: (12)                       1: {2: 3},
485: (12)                       2: {2: 2, 3: 2},
486: (12)                       3: {2: 2, 3: 2},
487: (12)                       4: {2: 2, 3: 2},
488: (8)                    }

----------------------------------------

File 314 - test_communicability.py:

1: (0)              from collections import defaultdict
2: (0)              import pytest
3: (0)              pytest.importorskip("numpy")
4: (0)              pytest.importorskip("scipy")
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.communicability_alg import communicability,
communicability_exp
7: (0)              class TestCommunicability:
8: (4)                  def test_communicability(self):
9: (8)                      answer = {
10: (12)                        0: {0: 1.5430806348152435, 1: 1.1752011936438012},
11: (12)                        1: {0: 1.1752011936438012, 1: 1.5430806348152435},
12: (8)                     }
13: (8)                     result = communicability(nx.path_graph(2))
14: (8)                     for k1, val in result.items():
15: (12)                        for k2 in val:
16: (16)                            assert answer[k1][k2] == pytest.approx(result[k1][k2], abs=1e-
7)
17: (4)                 def test_communicability2(self):
18: (8)                     answer_orig = {
19: (12)                        ("1", "1"): 1.6445956054135658,
20: (12)                        ("1", "Albert"): 0.7430186221096251,
21: (12)                        ("1", "Aric"): 0.7430186221096251,
22: (12)                        ("1", "Dan"): 1.6208126320442937,
23: (12)                        ("1", "Franck"): 0.42639707170035257,
24: (12)                        ("Albert", "1"): 0.7430186221096251,
25: (12)                        ("Albert", "Albert"): 2.4368257358712189,
26: (12)                        ("Albert", "Aric"): 1.4368257358712191,
27: (12)                        ("Albert", "Dan"): 2.0472097037446453,
28: (12)                        ("Albert", "Franck"): 1.8340111678944691,
29: (12)                        ("Aric", "1"): 0.7430186221096251,
30: (12)                        ("Aric", "Albert"): 1.4368257358712191,
31: (12)                        ("Aric", "Aric"): 2.4368257358712193,
32: (12)                        ("Aric", "Dan"): 2.0472097037446457,
33: (12)                        ("Aric", "Franck"): 1.8340111678944691,
```

```
34: (12)                        ("Dan", "1"): 1.6208126320442937,
35: (12)                        ("Dan", "Albert"): 2.0472097037446453,
36: (12)                        ("Dan", "Aric"): 2.0472097037446457,
37: (12)                        ("Dan", "Dan"): 3.1306328496328168,
38: (12)                        ("Dan", "Franck"): 1.4860372442192515,
39: (12)                        ("Franck", "1"): 0.42639707170035257,
40: (12)                        ("Franck", "Albert"): 1.8340111678944691,
41: (12)                        ("Franck", "Aric"): 1.8340111678944691,
42: (12)                        ("Franck", "Dan"): 1.4860372442192515,
43: (12)                        ("Franck", "Franck"): 2.3876142275231915,
44: (8)                     }
45: (8)                     answer = defaultdict(dict)
46: (8)                     for (k1, k2), v in answer_orig.items():
47: (12)                        answer[k1][k2] = v
48: (8)                     G1 = nx.Graph(
49: (12)                        [
50: (16)                            ("Franck", "Aric"),
51: (16)                            ("Aric", "Dan"),
52: (16)                            ("Dan", "Albert"),
53: (16)                            ("Albert", "Franck"),
54: (16)                            ("Dan", "1"),
55: (16)                            ("Franck", "Albert"),
56: (12)                        ]
57: (8)                     )
58: (8)                     result = communicability(G1)
59: (8)                     for k1, val in result.items():
60: (12)                        for k2 in val:
61: (16)                            assert answer[k1][k2] == pytest.approx(result[k1][k2], abs=1e-
7)
62: (8)                     result = communicability_exp(G1)
63: (8)                     for k1, val in result.items():
64: (12)                        for k2 in val:
65: (16)                            assert answer[k1][k2] == pytest.approx(result[k1][k2], abs=1e-
7)

----------------------------------------

File 315 - test_core.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import nodes_equal
4: (0)              class TestCore:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      t1 = nx.convert_node_labels_to_integers(nx.tetrahedral_graph(), 1)
8: (8)                      t2 = nx.convert_node_labels_to_integers(t1, 5)
9: (8)                      G = nx.union(t1, t2)
10: (8)                     G.add_edges_from(
11: (12)                        [
12: (16)                            (3, 7),
13: (16)                            (2, 11),
14: (16)                            (11, 5),
15: (16)                            (11, 12),
16: (16)                            (5, 12),
17: (16)                            (12, 19),
18: (16)                            (12, 18),
19: (16)                            (3, 9),
20: (16)                            (7, 9),
21: (16)                            (7, 10),
22: (16)                            (9, 10),
23: (16)                            (9, 20),
24: (16)                            (17, 13),
25: (16)                            (13, 14),
26: (16)                            (14, 15),
27: (16)                            (15, 16),
28: (16)                            (16, 13),
29: (12)                        ]
30: (8)                     )
```

```
 31: (8)                     G.add_node(21)
 32: (8)                     cls.G = G
 33: (8)                     degseq = [0, 1, 2, 2, 2, 2, 3]
 34: (8)                     H = nx.havel_hakimi_graph(degseq)
 35: (8)                     mapping = {6: 0, 0: 1, 4: 3, 5: 6, 3: 4, 1: 2, 2: 5}
 36: (8)                     cls.H = nx.relabel_nodes(H, mapping)
 37: (4)                 def test_trivial(self):
 38: (8)                     """Empty graph"""
 39: (8)                     G = nx.Graph()
 40: (8)                     assert nx.core_number(G) == {}
 41: (4)                 def test_core_number(self):
 42: (8)                     core = nx.core_number(self.G)
 43: (8)                     nodes_by_core = [sorted(n for n in core if core[n] == val) for val in
 range(4)]
 44: (8)                     assert nodes_equal(nodes_by_core[0], [21])
 45: (8)                     assert nodes_equal(nodes_by_core[1], [17, 18, 19, 20])
 46: (8)                     assert nodes_equal(nodes_by_core[2], [9, 10, 11, 12, 13, 14, 15, 16])
 47: (8)                     assert nodes_equal(nodes_by_core[3], [1, 2, 3, 4, 5, 6, 7, 8])
 48: (4)                 def test_core_number2(self):
 49: (8)                     core = nx.core_number(self.H)
 50: (8)                     nodes_by_core = [sorted(n for n in core if core[n] == val) for val in
 range(3)]
 51: (8)                     assert nodes_equal(nodes_by_core[0], [0])
 52: (8)                     assert nodes_equal(nodes_by_core[1], [1, 3])
 53: (8)                     assert nodes_equal(nodes_by_core[2], [2, 4, 5, 6])
 54: (4)                 def test_core_number_multigraph(self):
 55: (8)                     G = nx.complete_graph(3)
 56: (8)                     G = nx.MultiGraph(G)
 57: (8)                     G.add_edge(1, 2)
 58: (8)                     with pytest.raises(
 59: (12)                        nx.NetworkXNotImplemented, match="not implemented for multigraph
 type"
 60: (8)                     ):
 61: (12)                        nx.core_number(G)
 62: (4)                 def test_core_number_self_loop(self):
 63: (8)                     G = nx.cycle_graph(3)
 64: (8)                     G.add_edge(0, 0)
 65: (8)                     with pytest.raises(
 66: (12)                        nx.NetworkXNotImplemented, match="Input graph has self loops"
 67: (8)                     ):
 68: (12)                        nx.core_number(G)
 69: (4)                 def test_directed_core_number(self):
 70: (8)                     """core number had a bug for directed graphs found in issue #1959"""
 71: (8)                     G = nx.DiGraph()
 72: (8)                     edges = [(1, 2), (2, 1), (2, 3), (2, 4), (3, 4), (4, 3)]
 73: (8)                     G.add_edges_from(edges)
 74: (8)                     assert nx.core_number(G) == {1: 2, 2: 2, 3: 2, 4: 2}
 75: (8)                     more_edges = [(1, 5), (3, 5), (4, 5), (3, 6), (4, 6), (5, 6)]
 76: (8)                     G.add_edges_from(more_edges)
 77: (8)                     assert nx.core_number(G) == {1: 3, 2: 3, 3: 3, 4: 3, 5: 3, 6: 3}
 78: (4)                 def test_main_core(self):
 79: (8)                     main_core_subgraph = nx.k_core(self.H)
 80: (8)                     assert sorted(main_core_subgraph.nodes()) == [2, 4, 5, 6]
 81: (4)                 def test_k_core(self):
 82: (8)                     k_core_subgraph = nx.k_core(self.H, k=0)
 83: (8)                     assert sorted(k_core_subgraph.nodes()) == sorted(self.H.nodes())
 84: (8)                     k_core_subgraph = nx.k_core(self.H, k=1)
 85: (8)                     assert sorted(k_core_subgraph.nodes()) == [1, 2, 3, 4, 5, 6]
 86: (8)                     k_core_subgraph = nx.k_core(self.H, k=2)
 87: (8)                     assert sorted(k_core_subgraph.nodes()) == [2, 4, 5, 6]
 88: (4)                 def test_k_core_multigraph(self):
 89: (8)                     core_number = nx.core_number(self.H)
 90: (8)                     H = nx.MultiGraph(self.H)
 91: (8)                     with pytest.deprecated_call():
 92: (12)                        nx.k_core(H, k=0, core_number=core_number)
 93: (4)                 def test_main_crust(self):
 94: (8)                     main_crust_subgraph = nx.k_crust(self.H)
 95: (8)                     assert sorted(main_crust_subgraph.nodes()) == [0, 1, 3]
 96: (4)                 def test_k_crust(self):
```

```
97: (8)                     k_crust_subgraph = nx.k_crust(self.H, k=2)
98: (8)                     assert sorted(k_crust_subgraph.nodes()) == sorted(self.H.nodes())
99: (8)                     k_crust_subgraph = nx.k_crust(self.H, k=1)
100: (8)                    assert sorted(k_crust_subgraph.nodes()) == [0, 1, 3]
101: (8)                    k_crust_subgraph = nx.k_crust(self.H, k=0)
102: (8)                    assert sorted(k_crust_subgraph.nodes()) == [0]
103: (4)                def test_k_crust_multigraph(self):
104: (8)                    core_number = nx.core_number(self.H)
105: (8)                    H = nx.MultiGraph(self.H)
106: (8)                    with pytest.deprecated_call():
107: (12)                       nx.k_crust(H, k=0, core_number=core_number)
108: (4)                def test_main_shell(self):
109: (8)                    main_shell_subgraph = nx.k_shell(self.H)
110: (8)                    assert sorted(main_shell_subgraph.nodes()) == [2, 4, 5, 6]
111: (4)                def test_k_shell(self):
112: (8)                    k_shell_subgraph = nx.k_shell(self.H, k=2)
113: (8)                    assert sorted(k_shell_subgraph.nodes()) == [2, 4, 5, 6]
114: (8)                    k_shell_subgraph = nx.k_shell(self.H, k=1)
115: (8)                    assert sorted(k_shell_subgraph.nodes()) == [1, 3]
116: (8)                    k_shell_subgraph = nx.k_shell(self.H, k=0)
117: (8)                    assert sorted(k_shell_subgraph.nodes()) == [0]
118: (4)                def test_k_shell_multigraph(self):
119: (8)                    core_number = nx.core_number(self.H)
120: (8)                    H = nx.MultiGraph(self.H)
121: (8)                    with pytest.deprecated_call():
122: (12)                       nx.k_shell(H, k=0, core_number=core_number)
123: (4)                def test_k_corona(self):
124: (8)                    k_corona_subgraph = nx.k_corona(self.H, k=2)
125: (8)                    assert sorted(k_corona_subgraph.nodes()) == [2, 4, 5, 6]
126: (8)                    k_corona_subgraph = nx.k_corona(self.H, k=1)
127: (8)                    assert sorted(k_corona_subgraph.nodes()) == [1]
128: (8)                    k_corona_subgraph = nx.k_corona(self.H, k=0)
129: (8)                    assert sorted(k_corona_subgraph.nodes()) == [0]
130: (4)                def test_k_corona_multigraph(self):
131: (8)                    core_number = nx.core_number(self.H)
132: (8)                    H = nx.MultiGraph(self.H)
133: (8)                    with pytest.deprecated_call():
134: (12)                       nx.k_corona(H, k=0, core_number=core_number)
135: (4)                def test_k_truss(self):
136: (8)                    k_truss_subgraph = nx.k_truss(self.G, -1)
137: (8)                    assert sorted(k_truss_subgraph.nodes()) == list(range(1, 21))
138: (8)                    k_truss_subgraph = nx.k_truss(self.G, 0)
139: (8)                    assert sorted(k_truss_subgraph.nodes()) == list(range(1, 21))
140: (8)                    k_truss_subgraph = nx.k_truss(self.G, 1)
141: (8)                    assert sorted(k_truss_subgraph.nodes()) == list(range(1, 21))
142: (8)                    k_truss_subgraph = nx.k_truss(self.G, 2)
143: (8)                    assert sorted(k_truss_subgraph.nodes()) == list(range(1, 21))
144: (8)                    k_truss_subgraph = nx.k_truss(self.G, 3)
145: (8)                    assert sorted(k_truss_subgraph.nodes()) == list(range(1, 13))
146: (8)                    k_truss_subgraph = nx.k_truss(self.G, 4)
147: (8)                    assert sorted(k_truss_subgraph.nodes()) == list(range(1, 9))
148: (8)                    k_truss_subgraph = nx.k_truss(self.G, 5)
149: (8)                    assert sorted(k_truss_subgraph.nodes()) == []
150: (4)                def test_k_truss_digraph(self):
151: (8)                    G = nx.complete_graph(3)
152: (8)                    G = nx.DiGraph(G)
153: (8)                    G.add_edge(2, 1)
154: (8)                    with pytest.raises(
155: (12)                       nx.NetworkXNotImplemented, match="not implemented for directed
type"
156: (8)                    ):
157: (12)                       nx.k_truss(G, k=1)
158: (4)                def test_k_truss_multigraph(self):
159: (8)                    G = nx.complete_graph(3)
160: (8)                    G = nx.MultiGraph(G)
161: (8)                    G.add_edge(1, 2)
162: (8)                    with pytest.raises(
163: (12)                       nx.NetworkXNotImplemented, match="not implemented for multigraph
type"
```

```
164: (8)                    ):
165: (12)                       nx.k_truss(G, k=1)
166: (4)                def test_k_truss_self_loop(self):
167: (8)                    G = nx.cycle_graph(3)
168: (8)                    G.add_edge(0, 0)
169: (8)                    with pytest.raises(
170: (12)                       nx.NetworkXNotImplemented, match="Input graph has self loops"
171: (8)                    ):
172: (12)                       nx.k_truss(G, k=1)
173: (4)                def test_onion_layers(self):
174: (8)                    layers = nx.onion_layers(self.G)
175: (8)                    nodes_by_layer = [
176: (12)                       sorted(n for n in layers if layers[n] == val) for val in range(1,
7)
177: (8)                    ]
178: (8)                    assert nodes_equal(nodes_by_layer[0], [21])
179: (8)                    assert nodes_equal(nodes_by_layer[1], [17, 18, 19, 20])
180: (8)                    assert nodes_equal(nodes_by_layer[2], [10, 12, 13, 14, 15, 16])
181: (8)                    assert nodes_equal(nodes_by_layer[3], [9, 11])
182: (8)                    assert nodes_equal(nodes_by_layer[4], [1, 2, 4, 5, 6, 8])
183: (8)                    assert nodes_equal(nodes_by_layer[5], [3, 7])
184: (4)                def test_onion_digraph(self):
185: (8)                    G = nx.complete_graph(3)
186: (8)                    G = nx.DiGraph(G)
187: (8)                    G.add_edge(2, 1)
188: (8)                    with pytest.raises(
189: (12)                       nx.NetworkXNotImplemented, match="not implemented for directed
type"
190: (8)                    ):
191: (12)                       nx.onion_layers(G)
192: (4)                def test_onion_multigraph(self):
193: (8)                    G = nx.complete_graph(3)
194: (8)                    G = nx.MultiGraph(G)
195: (8)                    G.add_edge(1, 2)
196: (8)                    with pytest.raises(
197: (12)                       nx.NetworkXNotImplemented, match="not implemented for multigraph
type"
198: (8)                    ):
199: (12)                       nx.onion_layers(G)
200: (4)                def test_onion_self_loop(self):
201: (8)                    G = nx.cycle_graph(3)
202: (8)                    G.add_edge(0, 0)
203: (8)                    with pytest.raises(
204: (12)                       nx.NetworkXNotImplemented, match="Input graph contains self loops"
205: (8)                    ):
206: (12)                       nx.onion_layers(G)
```

File 316 - test_covering.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestMinEdgeCover:
4: (4)                  """Tests for :func:`networkx.algorithms.min_edge_cover`"""
5: (4)                  def test_empty_graph(self):
6: (8)                      G = nx.Graph()
7: (8)                      assert nx.min_edge_cover(G) == set()
8: (4)                  def test_graph_with_loop(self):
9: (8)                      G = nx.Graph()
10: (8)                     G.add_edge(0, 0)
11: (8)                     assert nx.min_edge_cover(G) == {(0, 0)}
12: (4)                 def test_graph_with_isolated_v(self):
13: (8)                     G = nx.Graph()
14: (8)                     G.add_node(1)
15: (8)                     with pytest.raises(
16: (12)                        nx.NetworkXException,
17: (12)                        match="Graph has a node with no edge incident on it, so no edge
cover exists.",
```

```
18: (8)                     ):
19: (12)                        nx.min_edge_cover(G)
20: (4)                 def test_graph_single_edge(self):
21: (8)                     G = nx.Graph([(0, 1)])
22: (8)                     assert nx.min_edge_cover(G) in ({(0, 1)}, {(1, 0)})
23: (4)                 def test_graph_two_edge_path(self):
24: (8)                     G = nx.path_graph(3)
25: (8)                     min_cover = nx.min_edge_cover(G)
26: (8)                     assert len(min_cover) == 2
27: (8)                     for u, v in G.edges:
28: (12)                        assert (u, v) in min_cover or (v, u) in min_cover
29: (4)                 def test_bipartite_explicit(self):
30: (8)                     G = nx.Graph()
31: (8)                     G.add_nodes_from([1, 2, 3, 4], bipartite=0)
32: (8)                     G.add_nodes_from(["a", "b", "c"], bipartite=1)
33: (8)                     G.add_edges_from([(1, "a"), (1, "b"), (2, "b"), (2, "c"), (3, "c"),
(4, "a")])
34: (8)                     min_cover = nx.min_edge_cover(
35: (12)                        G, nx.algorithms.bipartite.matching.eppstein_matching
36: (8)                     )
37: (8)                     assert nx.is_edge_cover(G, min_cover)
38: (8)                     assert len(min_cover) == 8
39: (8)                     min_cover2 = nx.min_edge_cover(G)
40: (8)                     assert nx.is_edge_cover(G, min_cover2)
41: (8)                     assert len(min_cover2) == 4
42: (4)                 def test_complete_graph_even(self):
43: (8)                     G = nx.complete_graph(10)
44: (8)                     min_cover = nx.min_edge_cover(G)
45: (8)                     assert nx.is_edge_cover(G, min_cover)
46: (8)                     assert len(min_cover) == 5
47: (4)                 def test_complete_graph_odd(self):
48: (8)                     G = nx.complete_graph(11)
49: (8)                     min_cover = nx.min_edge_cover(G)
50: (8)                     assert nx.is_edge_cover(G, min_cover)
51: (8)                     assert len(min_cover) == 6
52: (0)             class TestIsEdgeCover:
53: (4)                 """Tests for :func:`networkx.algorithms.is_edge_cover`"""
54: (4)                 def test_empty_graph(self):
55: (8)                     G = nx.Graph()
56: (8)                     assert nx.is_edge_cover(G, set())
57: (4)                 def test_graph_with_loop(self):
58: (8)                     G = nx.Graph()
59: (8)                     G.add_edge(1, 1)
60: (8)                     assert nx.is_edge_cover(G, {(1, 1)})
61: (4)                 def test_graph_single_edge(self):
62: (8)                     G = nx.Graph()
63: (8)                     G.add_edge(0, 1)
64: (8)                     assert nx.is_edge_cover(G, {(0, 0), (1, 1)})
65: (8)                     assert nx.is_edge_cover(G, {(0, 1), (1, 0)})
66: (8)                     assert nx.is_edge_cover(G, {(0, 1)})
67: (8)                     assert not nx.is_edge_cover(G, {(0, 0)})
```

File 317 - test_cuts.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.cuts` module."""
2: (0)              import networkx as nx
3: (0)              class TestCutSize:
4: (4)                  """Unit tests for the :func:`~networkx.cut_size` function."""
5: (4)                  def test_symmetric(self):
6: (8)                      """Tests that the cut size is symmetric."""
7: (8)                      G = nx.barbell_graph(3, 0)
8: (8)                      S = {0, 1, 4}
9: (8)                      T = {2, 3, 5}
10: (8)                     assert nx.cut_size(G, S, T) == 4
11: (8)                     assert nx.cut_size(G, T, S) == 4
12: (4)                 def test_single_edge(self):
13: (8)                     """Tests for a cut of a single edge."""
```

```
14: (8)                     G = nx.barbell_graph(3, 0)
15: (8)                     S = {0, 1, 2}
16: (8)                     T = {3, 4, 5}
17: (8)                     assert nx.cut_size(G, S, T) == 1
18: (8)                     assert nx.cut_size(G, T, S) == 1
19: (4)                 def test_directed(self):
20: (8)                     """Tests that each directed edge is counted once in the cut."""
21: (8)                     G = nx.barbell_graph(3, 0).to_directed()
22: (8)                     S = {0, 1, 2}
23: (8)                     T = {3, 4, 5}
24: (8)                     assert nx.cut_size(G, S, T) == 2
25: (8)                     assert nx.cut_size(G, T, S) == 2
26: (4)                 def test_directed_symmetric(self):
27: (8)                     """Tests that a cut in a directed graph is symmetric."""
28: (8)                     G = nx.barbell_graph(3, 0).to_directed()
29: (8)                     S = {0, 1, 4}
30: (8)                     T = {2, 3, 5}
31: (8)                     assert nx.cut_size(G, S, T) == 8
32: (8)                     assert nx.cut_size(G, T, S) == 8
33: (4)                 def test_multigraph(self):
34: (8)                     """Tests that parallel edges are each counted for a cut."""
35: (8)                     G = nx.MultiGraph(["ab", "ab"])
36: (8)                     assert nx.cut_size(G, {"a"}, {"b"}) == 2
37: (0)             class TestVolume:
38: (4)                 """Unit tests for the :func:`~networkx.volume` function."""
39: (4)                 def test_graph(self):
40: (8)                     G = nx.cycle_graph(4)
41: (8)                     assert nx.volume(G, {0, 1}) == 4
42: (4)                 def test_digraph(self):
43: (8)                     G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 0)])
44: (8)                     assert nx.volume(G, {0, 1}) == 2
45: (4)                 def test_multigraph(self):
46: (8)                     edges = list(nx.cycle_graph(4).edges())
47: (8)                     G = nx.MultiGraph(edges * 2)
48: (8)                     assert nx.volume(G, {0, 1}) == 8
49: (4)                 def test_multidigraph(self):
50: (8)                     edges = [(0, 1), (1, 2), (2, 3), (3, 0)]
51: (8)                     G = nx.MultiDiGraph(edges * 2)
52: (8)                     assert nx.volume(G, {0, 1}) == 4
53: (4)                 def test_barbell(self):
54: (8)                     G = nx.barbell_graph(3, 0)
55: (8)                     assert nx.volume(G, {0, 1, 2}) == 7
56: (8)                     assert nx.volume(G, {3, 4, 5}) == 7
57: (0)             class TestNormalizedCutSize:
58: (4)                 """Unit tests for the :func:`~networkx.normalized_cut_size` function."""
59: (4)                 def test_graph(self):
60: (8)                     G = nx.path_graph(4)
61: (8)                     S = {1, 2}
62: (8)                     T = set(G) - S
63: (8)                     size = nx.normalized_cut_size(G, S, T)
64: (8)                     expected = 2 * ((1 / 4) + (1 / 2))
65: (8)                     assert expected == size
66: (8)                     assert expected == nx.normalized_cut_size(G, S)
67: (4)                 def test_directed(self):
68: (8)                     G = nx.DiGraph([(0, 1), (1, 2), (2, 3)])
69: (8)                     S = {1, 2}
70: (8)                     T = set(G) - S
71: (8)                     size = nx.normalized_cut_size(G, S, T)
72: (8)                     expected = 2 * ((1 / 2) + (1 / 1))
73: (8)                     assert expected == size
74: (8)                     assert expected == nx.normalized_cut_size(G, S)
75: (0)             class TestConductance:
76: (4)                 """Unit tests for the :func:`~networkx.conductance` function."""
77: (4)                 def test_graph(self):
78: (8)                     G = nx.barbell_graph(5, 0)
79: (8)                     S = {4}
80: (8)                     T = {5}
81: (8)                     conductance = nx.conductance(G, S, T)
82: (8)                     expected = 1 / 5
```

```
83: (8)                     assert expected == conductance
84: (8)                     G2 = nx.barbell_graph(3, 0)
85: (8)                     S2 = {0, 1, 2}
86: (8)                     assert nx.conductance(G2, S2) == 1 / 7
87: (0)             class TestEdgeExpansion:
88: (4)                 """Unit tests for the :func:`~networkx.edge_expansion` function."""
89: (4)                 def test_graph(self):
90: (8)                     G = nx.barbell_graph(5, 0)
91: (8)                     S = set(range(5))
92: (8)                     T = set(G) - S
93: (8)                     expansion = nx.edge_expansion(G, S, T)
94: (8)                     expected = 1 / 5
95: (8)                     assert expected == expansion
96: (8)                     assert expected == nx.edge_expansion(G, S)
97: (0)             class TestNodeExpansion:
98: (4)                 """Unit tests for the :func:`~networkx.node_expansion` function."""
99: (4)                 def test_graph(self):
100: (8)                    G = nx.path_graph(8)
101: (8)                    S = {3, 4, 5}
102: (8)                    expansion = nx.node_expansion(G, S)
103: (8)                    expected = 5 / 3
104: (8)                    assert expected == expansion
105: (0)            class TestBoundaryExpansion:
106: (4)                """Unit tests for the :func:`~networkx.boundary_expansion` function."""
107: (4)                def test_graph(self):
108: (8)                    G = nx.complete_graph(10)
109: (8)                    S = set(range(4))
110: (8)                    expansion = nx.boundary_expansion(G, S)
111: (8)                    expected = 6 / 4
112: (8)                    assert expected == expansion
113: (0)            class TestMixingExpansion:
114: (4)                """Unit tests for the :func:`~networkx.mixing_expansion` function."""
115: (4)                def test_graph(self):
116: (8)                    G = nx.barbell_graph(5, 0)
117: (8)                    S = set(range(5))
118: (8)                    T = set(G) - S
119: (8)                    expansion = nx.mixing_expansion(G, S, T)
120: (8)                    expected = 1 / (2 * (5 * 4 + 1))
121: (8)                    assert expected == expansion
```

File 318 - test_cycles.py:

```
1: (0)              from itertools import chain, islice, tee
2: (0)              from math import inf
3: (0)              from random import shuffle
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.traversal.edgedfs import FORWARD, REVERSE
7: (0)              def check_independent(basis):
8: (4)                  if len(basis) == 0:
9: (8)                      return
10: (4)                 np = pytest.importorskip("numpy")
11: (4)                 sp = pytest.importorskip("scipy")  # Required by incidence_matrix
12: (4)                 H = nx.Graph()
13: (4)                 for b in basis:
14: (8)                     nx.add_cycle(H, b)
15: (4)                 inc = nx.incidence_matrix(H, oriented=True)
16: (4)                 rank = np.linalg.matrix_rank(inc.toarray(), tol=None, hermitian=False)
17: (4)                 assert inc.shape[1] - rank == len(basis)
18: (0)             class TestCycles:
19: (4)                 @classmethod
20: (4)                 def setup_class(cls):
21: (8)                     G = nx.Graph()
22: (8)                     nx.add_cycle(G, [0, 1, 2, 3])
23: (8)                     nx.add_cycle(G, [0, 3, 4, 5])
24: (8)                     nx.add_cycle(G, [0, 1, 6, 7, 8])
25: (8)                     G.add_edge(8, 9)
```

```
 26: (8)                    cls.G = G
 27: (4)                def is_cyclic_permutation(self, a, b):
 28: (8)                    n = len(a)
 29: (8)                    if len(b) != n:
 30: (12)                       return False
 31: (8)                    l = a + a
 32: (8)                    return any(l[i : i + n] == b for i in range(n))
 33: (4)                def test_cycle_basis(self):
 34: (8)                    G = self.G
 35: (8)                    cy = nx.cycle_basis(G, 0)
 36: (8)                    sort_cy = sorted(sorted(c) for c in cy)
 37: (8)                    assert sort_cy == [[0, 1, 2, 3], [0, 1, 6, 7, 8], [0, 3, 4, 5]]
 38: (8)                    cy = nx.cycle_basis(G, 1)
 39: (8)                    sort_cy = sorted(sorted(c) for c in cy)
 40: (8)                    assert sort_cy == [[0, 1, 2, 3], [0, 1, 6, 7, 8], [0, 3, 4, 5]]
 41: (8)                    cy = nx.cycle_basis(G, 9)
 42: (8)                    sort_cy = sorted(sorted(c) for c in cy)
 43: (8)                    assert sort_cy == [[0, 1, 2, 3], [0, 1, 6, 7, 8], [0, 3, 4, 5]]
 44: (8)                    nx.add_cycle(G, "ABC")
 45: (8)                    cy = nx.cycle_basis(G, 9)
 46: (8)                    sort_cy = sorted(sorted(c) for c in cy[:-1]) + [sorted(cy[-1])]
 47: (8)                    assert sort_cy == [[0, 1, 2, 3], [0, 1, 6, 7, 8], [0, 3, 4, 5], ["A",
"B", "C"]]
 48: (4)                def test_cycle_basis2(self):
 49: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
 50: (12)                       G = nx.DiGraph()
 51: (12)                       cy = nx.cycle_basis(G, 0)
 52: (4)                def test_cycle_basis3(self):
 53: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
 54: (12)                       G = nx.MultiGraph()
 55: (12)                       cy = nx.cycle_basis(G, 0)
 56: (4)                def test_cycle_basis_ordered(self):
 57: (8)                    G = nx.cycle_graph(5)
 58: (8)                    G.update(nx.cycle_graph(range(3, 8)))
 59: (8)                    cbG = nx.cycle_basis(G)
 60: (8)                    perm = {1: 0, 0: 1}  # switch 0 and 1
 61: (8)                    H = nx.relabel_nodes(G, perm)
 62: (8)                    cbH = [[perm.get(n, n) for n in cyc] for cyc in nx.cycle_basis(H)]
 63: (8)                    assert cbG == cbH
 64: (4)                def test_cycle_basis_self_loop(self):
 65: (8)                    """Tests the function for graphs with self loops"""
 66: (8)                    G = nx.Graph()
 67: (8)                    nx.add_cycle(G, [0, 1, 2, 3])
 68: (8)                    nx.add_cycle(G, [0, 0, 6, 2])
 69: (8)                    cy = nx.cycle_basis(G)
 70: (8)                    sort_cy = sorted(sorted(c) for c in cy)
 71: (8)                    assert sort_cy == [[0], [0, 1, 2], [0, 2, 3], [0, 2, 6]]
 72: (4)                def test_simple_cycles(self):
 73: (8)                    edges = [(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]
 74: (8)                    G = nx.DiGraph(edges)
 75: (8)                    cc = sorted(nx.simple_cycles(G))
 76: (8)                    ca = [[0], [0, 1, 2], [0, 2], [1, 2], [2]]
 77: (8)                    assert len(cc) == len(ca)
 78: (8)                    for c in cc:
 79: (12)                       assert any(self.is_cyclic_permutation(c, rc) for rc in ca)
 80: (4)                def test_simple_cycles_singleton(self):
 81: (8)                    G = nx.Graph([(0, 0)])  # self-loop
 82: (8)                    assert list(nx.simple_cycles(G)) == [[0]]
 83: (4)                def test_unsortable(self):
 84: (8)                    G = nx.DiGraph()
 85: (8)                    nx.add_cycle(G, ["a", 1])
 86: (8)                    c = list(nx.simple_cycles(G))
 87: (8)                    assert len(c) == 1
 88: (4)                def test_simple_cycles_small(self):
 89: (8)                    G = nx.DiGraph()
 90: (8)                    nx.add_cycle(G, [1, 2, 3])
 91: (8)                    c = sorted(nx.simple_cycles(G))
 92: (8)                    assert len(c) == 1
 93: (8)                    assert self.is_cyclic_permutation(c[0], [1, 2, 3])
```

```
 94: (8)                    nx.add_cycle(G, [10, 20, 30])
 95: (8)                    cc = sorted(nx.simple_cycles(G))
 96: (8)                    assert len(cc) == 2
 97: (8)                    ca = [[1, 2, 3], [10, 20, 30]]
 98: (8)                    for c in cc:
 99: (12)                       assert any(self.is_cyclic_permutation(c, rc) for rc in ca)
 100: (4)                def test_simple_cycles_empty(self):
 101: (8)                    G = nx.DiGraph()
 102: (8)                    assert list(nx.simple_cycles(G)) == []
 103: (4)                def worst_case_graph(self, k):
 104: (8)                    G = nx.DiGraph()
 105: (8)                    for n in range(2, k + 2):
 106: (12)                       G.add_edge(1, n)
 107: (12)                       G.add_edge(n, k + 2)
 108: (8)                    G.add_edge(2 * k + 1, 1)
 109: (8)                    for n in range(k + 2, 2 * k + 2):
 110: (12)                       G.add_edge(n, 2 * k + 2)
 111: (12)                       G.add_edge(n, n + 1)
 112: (8)                    G.add_edge(2 * k + 3, k + 2)
 113: (8)                    for n in range(2 * k + 3, 3 * k + 3):
 114: (12)                       G.add_edge(2 * k + 2, n)
 115: (12)                       G.add_edge(n, 3 * k + 3)
 116: (8)                    G.add_edge(3 * k + 3, 2 * k + 2)
 117: (8)                    return G
 118: (4)                def test_worst_case_graph(self):
 119: (8)                    for k in range(3, 10):
 120: (12)                       G = self.worst_case_graph(k)
 121: (12)                       l = len(list(nx.simple_cycles(G)))
 122: (12)                       assert l == 3 * k
 123: (4)                def test_recursive_simple_and_not(self):
 124: (8)                    for k in range(2, 10):
 125: (12)                       G = self.worst_case_graph(k)
 126: (12)                       cc = sorted(nx.simple_cycles(G))
 127: (12)                       rcc = sorted(nx.recursive_simple_cycles(G))
 128: (12)                       assert len(cc) == len(rcc)
 129: (12)                       for c in cc:
 130: (16)                           assert any(self.is_cyclic_permutation(c, r) for r in rcc)
 131: (12)                       for rc in rcc:
 132: (16)                           assert any(self.is_cyclic_permutation(rc, c) for c in cc)
 133: (4)                def test_simple_graph_with_reported_bug(self):
 134: (8)                    G = nx.DiGraph()
 135: (8)                    edges = [
 136: (12)                       (0, 2),
 137: (12)                       (0, 3),
 138: (12)                       (1, 0),
 139: (12)                       (1, 3),
 140: (12)                       (2, 1),
 141: (12)                       (2, 4),
 142: (12)                       (3, 2),
 143: (12)                       (3, 4),
 144: (12)                       (4, 0),
 145: (12)                       (4, 1),
 146: (12)                       (4, 5),
 147: (12)                       (5, 0),
 148: (12)                       (5, 1),
 149: (12)                       (5, 2),
 150: (12)                       (5, 3),
 151: (8)                    ]
 152: (8)                    G.add_edges_from(edges)
 153: (8)                    cc = sorted(nx.simple_cycles(G))
 154: (8)                    assert len(cc) == 26
 155: (8)                    rcc = sorted(nx.recursive_simple_cycles(G))
 156: (8)                    assert len(cc) == len(rcc)
 157: (8)                    for c in cc:
 158: (12)                       assert any(self.is_cyclic_permutation(c, rc) for rc in rcc)
 159: (8)                    for rc in rcc:
 160: (12)                       assert any(self.is_cyclic_permutation(rc, c) for c in cc)
 161: (0)            def pairwise(iterable):
 162: (4)                a, b = tee(iterable)
```

```
163: (4)                next(b, None)
164: (4)                return zip(a, b)
165: (0)            def cycle_edges(c):
166: (4)                return pairwise(chain(c, islice(c, 1)))
167: (0)            def directed_cycle_edgeset(c):
168: (4)                return frozenset(cycle_edges(c))
169: (0)            def undirected_cycle_edgeset(c):
170: (4)                if len(c) == 1:
171: (8)                    return frozenset(cycle_edges(c))
172: (4)                return frozenset(map(frozenset, cycle_edges(c)))
173: (0)            def multigraph_cycle_edgeset(c):
174: (4)                if len(c) <= 2:
175: (8)                    return frozenset(cycle_edges(c))
176: (4)                else:
177: (8)                    return frozenset(map(frozenset, cycle_edges(c)))
178: (0)            class TestCycleEnumeration:
179: (4)                @staticmethod
180: (4)                def K(n):
181: (8)                    return nx.complete_graph(n)
182: (4)                @staticmethod
183: (4)                def D(n):
184: (8)                    return nx.complete_graph(n).to_directed()
185: (4)                @staticmethod
186: (4)                def edgeset_function(g):
187: (8)                    if g.is_directed():
188: (12)                       return directed_cycle_edgeset
189: (8)                    elif g.is_multigraph():
190: (12)                       return multigraph_cycle_edgeset
191: (8)                    else:
192: (12)                       return undirected_cycle_edgeset
193: (4)                def check_cycle(self, g, c, es, cache, source, original_c, length_bound,
chordless):
194: (8)                    if length_bound is not None and len(c) > length_bound:
195: (12)                       raise RuntimeError(
196: (16)                           f"computed cycle {original_c} exceeds length bound
{length_bound}"
197: (12)                       )
198: (8)                    if source == "computed":
199: (12)                       if es in cache:
200: (16)                           raise RuntimeError(
201: (20)                               f"computed cycle {original_c} has already been found!"
202: (16)                           )
203: (12)                       else:
204: (16)                           cache[es] = tuple(original_c)
205: (8)                    else:
206: (12)                       if es in cache:
207: (16)                           cache.pop(es)
208: (12)                       else:
209: (16)                           raise RuntimeError(f"expected cycle {original_c} was not
computed")
210: (8)                    if not all(g.has_edge(*e) for e in es):
211: (12)                       raise RuntimeError(
212: (16)                           f"{source} claimed cycle {original_c} is not a cycle of g"
213: (12)                       )
214: (8)                    if chordless and len(g.subgraph(c).edges) > len(c):
215: (12)                       raise RuntimeError(f"{source} cycle {original_c} is not
chordless")
216: (4)                def check_cycle_algorithm(
217: (8)                    self,
218: (8)                    g,
219: (8)                    expected_cycles,
220: (8)                    length_bound=None,
221: (8)                    chordless=False,
222: (8)                    algorithm=None,
223: (4)                ):
224: (8)                    if algorithm is None:
225: (12)                       algorithm = nx.chordless_cycles if chordless else nx.simple_cycles
226: (8)                    relabel = list(range(len(g)))
227: (8)                    shuffle(relabel)
```

```
228: (8)                    label = dict(zip(g, relabel))
229: (8)                    unlabel = dict(zip(relabel, g))
230: (8)                    h = nx.relabel_nodes(g, label, copy=True)
231: (8)                    edgeset = self.edgeset_function(h)
232: (8)                    params = {}
233: (8)                    if length_bound is not None:
234: (12)                       params["length_bound"] = length_bound
235: (8)                    cycle_cache = {}
236: (8)                    for c in algorithm(h, **params):
237: (12)                       original_c = [unlabel[x] for x in c]
238: (12)                       es = edgeset(c)
239: (12)                       self.check_cycle(
240: (16)                           h, c, es, cycle_cache, "computed", original_c, length_bound,
chordless
241: (12)                       )
242: (8)                    if isinstance(expected_cycles, int):
243: (12)                       if len(cycle_cache) != expected_cycles:
244: (16)                           raise RuntimeError(
245: (20)                               f"expected {expected_cycles} cycles, got
{len(cycle_cache)}"
246: (16)                           )
247: (12)                       return
248: (8)                    for original_c in expected_cycles:
249: (12)                       c = [label[x] for x in original_c]
250: (12)                       es = edgeset(c)
251: (12)                       self.check_cycle(
252: (16)                           h, c, es, cycle_cache, "expected", original_c, length_bound,
chordless
253: (12)                       )
254: (8)                    if len(cycle_cache):
255: (12)                       for c in cycle_cache.values():
256: (16)                           raise RuntimeError(
257: (20)                               f"computed cycle {c} is valid but not in the expected
cycle set!"
258: (16)                           )
259: (4)                def check_cycle_enumeration_integer_sequence(
260: (8)                    self,
261: (8)                    g_family,
262: (8)                    cycle_counts,
263: (8)                    length_bound=None,
264: (8)                    chordless=False,
265: (8)                    algorithm=None,
266: (4)                ):
267: (8)                    for g, num_cycles in zip(g_family, cycle_counts):
268: (12)                       self.check_cycle_algorithm(
269: (16)                           g,
270: (16)                           num_cycles,
271: (16)                           length_bound=length_bound,
272: (16)                           chordless=chordless,
273: (16)                           algorithm=algorithm,
274: (12)                       )
275: (4)                def test_directed_chordless_cycle_digons(self):
276: (8)                    g = nx.DiGraph()
277: (8)                    nx.add_cycle(g, range(5))
278: (8)                    nx.add_cycle(g, range(5)[::-1])
279: (8)                    g.add_edge(0, 0)
280: (8)                    expected_cycles = [(0,), (1, 2), (2, 3), (3, 4)]
281: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
282: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True,
length_bound=2)
283: (8)                    expected_cycles = [c for c in expected_cycles if len(c) < 2]
284: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True,
length_bound=1)
285: (4)                def test_directed_chordless_cycle_undirected(self):
286: (8)                    g = nx.DiGraph([(1, 2), (2, 3), (3, 4), (4, 5), (5, 0), (5, 1), (0,
2)])
287: (8)                    expected_cycles = [(0, 2, 3, 4, 5), (1, 2, 3, 4, 5)]
288: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
289: (8)                    g = nx.DiGraph()
```

```
290: (8)                    nx.add_cycle(g, range(5))
291: (8)                    nx.add_cycle(g, range(4, 9))
292: (8)                    g.add_edge(7, 3)
293: (8)                    expected_cycles = [(0, 1, 2, 3, 4), (3, 4, 5, 6, 7), (4, 5, 6, 7, 8)]
294: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
295: (8)                    g.add_edge(3, 7)
296: (8)                    expected_cycles = [(0, 1, 2, 3, 4), (3, 7), (4, 5, 6, 7, 8)]
297: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
298: (8)                    expected_cycles = [(3, 7)]
299: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True,
length_bound=4)
300: (8)                    g.remove_edge(7, 3)
301: (8)                    expected_cycles = [(0, 1, 2, 3, 4), (4, 5, 6, 7, 8)]
302: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
303: (8)                    g = nx.DiGraph((i, j) for i in range(10) for j in range(i))
304: (8)                    expected_cycles = []
305: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
306: (4)                def test_chordless_cycles_directed(self):
307: (8)                    G = nx.DiGraph()
308: (8)                    nx.add_cycle(G, range(5))
309: (8)                    nx.add_cycle(G, range(4, 12))
310: (8)                    expected = [[*range(5)], [*range(4, 12)]]
311: (8)                    self.check_cycle_algorithm(G, expected, chordless=True)
312: (8)                    self.check_cycle_algorithm(
313: (12)                       G, [c for c in expected if len(c) <= 5], length_bound=5,
chordless=True
314: (8)                    )
315: (8)                    G.add_edge(7, 3)
316: (8)                    expected.append([*range(3, 8)])
317: (8)                    self.check_cycle_algorithm(G, expected, chordless=True)
318: (8)                    self.check_cycle_algorithm(
319: (12)                       G, [c for c in expected if len(c) <= 5], length_bound=5,
chordless=True
320: (8)                    )
321: (8)                    G.add_edge(3, 7)
322: (8)                    expected[-1] = [7, 3]
323: (8)                    self.check_cycle_algorithm(G, expected, chordless=True)
324: (8)                    self.check_cycle_algorithm(
325: (12)                       G, [c for c in expected if len(c) <= 5], length_bound=5,
chordless=True
326: (8)                    )
327: (8)                    expected.pop()
328: (8)                    G.remove_edge(7, 3)
329: (8)                    self.check_cycle_algorithm(G, expected, chordless=True)
330: (8)                    self.check_cycle_algorithm(
331: (12)                       G, [c for c in expected if len(c) <= 5], length_bound=5,
chordless=True
332: (8)                    )
333: (4)                def test_directed_chordless_cycle_diclique(self):
334: (8)                    g_family = [self.D(n) for n in range(10)]
335: (8)                    expected_cycles = [(n * n - n) // 2 for n in range(10)]
336: (8)                    self.check_cycle_enumeration_integer_sequence(
337: (12)                       g_family, expected_cycles, chordless=True
338: (8)                    )
339: (8)                    expected_cycles = [(n * n - n) // 2 for n in range(10)]
340: (8)                    self.check_cycle_enumeration_integer_sequence(
341: (12)                       g_family, expected_cycles, length_bound=2
342: (8)                    )
343: (4)                def test_directed_chordless_loop_blockade(self):
344: (8)                    g = nx.DiGraph((i, i) for i in range(10))
345: (8)                    nx.add_cycle(g, range(10))
346: (8)                    expected_cycles = [(i,) for i in range(10)]
347: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
348: (8)                    self.check_cycle_algorithm(g, expected_cycles, length_bound=1)
349: (8)                    g = nx.MultiDiGraph(g)
350: (8)                    g.add_edges_from((i, i) for i in range(0, 10, 2))
351: (8)                    expected_cycles = [(i,) for i in range(1, 10, 2)]
352: (8)                    self.check_cycle_algorithm(g, expected_cycles, chordless=True)
353: (4)                def test_simple_cycles_notable_clique_sequences(self):
```

```
(8)                 g_family = [self.K(n) for n in range(2, 12)]
(8)                 expected = [0, 1, 4, 10, 20, 35, 56, 84, 120, 165, 220]
(8)                 self.check_cycle_enumeration_integer_sequence(
(12)                    g_family, expected, length_bound=3
(8)                 )
(8)                 def triangles(g, **kwargs):
(12)                    yield from (c for c in nx.simple_cycles(g, **kwargs) if len(c) ==
3)
(8)                 g_family = [self.D(n) for n in range(2, 12)]
(8)                 expected = [2 * e for e in expected]
(8)                 self.check_cycle_enumeration_integer_sequence(
(12)                    g_family, expected, length_bound=3, algorithm=triangles
(8)                 )
(8)                 def four_cycles(g, **kwargs):
(12)                    yield from (c for c in nx.simple_cycles(g, **kwargs) if len(c) ==
4)
(8)                 expected = [0, 0, 0, 3, 15, 45, 105, 210, 378, 630, 990]
(8)                 g_family = [self.K(n) for n in range(1, 12)]
(8)                 self.check_cycle_enumeration_integer_sequence(
(12)                    g_family, expected, length_bound=4, algorithm=four_cycles
(8)                 )
(8)                 expected = [2 * e for e in expected]
(8)                 g_family = [self.D(n) for n in range(1, 15)]
(8)                 self.check_cycle_enumeration_integer_sequence(
(12)                    g_family, expected, length_bound=4, algorithm=four_cycles
(8)                 )
(8)                 expected = [0, 1, 5, 20, 84, 409, 2365]
(8)                 g_family = [self.D(n) for n in range(1, 8)]
(8)                 self.check_cycle_enumeration_integer_sequence(g_family, expected)
(8)                 expected = [0, 0, 0, 1, 7, 37, 197, 1172]
(8)                 g_family = [self.K(n) for n in range(8)]
(8)                 self.check_cycle_enumeration_integer_sequence(g_family, expected)
(4)             def test_directed_chordless_cycle_parallel_multiedges(self):
(8)                 g = nx.MultiGraph()
(8)                 nx.add_cycle(g, range(5))
(8)                 expected = [[*range(5)]]
(8)                 self.check_cycle_algorithm(g, expected, chordless=True)
(8)                 nx.add_cycle(g, range(5))
(8)                 expected = [*cycle_edges(range(5))]
(8)                 self.check_cycle_algorithm(g, expected, chordless=True)
(8)                 nx.add_cycle(g, range(5))
(8)                 expected = []
(8)                 self.check_cycle_algorithm(g, expected, chordless=True)
(8)                 g = nx.MultiDiGraph()
(8)                 nx.add_cycle(g, range(5))
(8)                 expected = [[*range(5)]]
(8)                 self.check_cycle_algorithm(g, expected, chordless=True)
(8)                 nx.add_cycle(g, range(5))
(8)                 self.check_cycle_algorithm(g, [], chordless=True)
(8)                 nx.add_cycle(g, range(5))
(8)                 self.check_cycle_algorithm(g, [], chordless=True)
(8)                 g = nx.MultiDiGraph()
(8)                 nx.add_cycle(g, range(5))
(8)                 nx.add_cycle(g, range(5)[::-1])
(8)                 expected = [*cycle_edges(range(5))]
(8)                 self.check_cycle_algorithm(g, expected, chordless=True)
(8)                 nx.add_cycle(g, range(5))
(8)                 self.check_cycle_algorithm(g, [], chordless=True)
(4)             def test_chordless_cycles_graph(self):
(8)                 G = nx.Graph()
(8)                 nx.add_cycle(G, range(5))
(8)                 nx.add_cycle(G, range(4, 12))
(8)                 expected = [[*range(5)], [*range(4, 12)]]
(8)                 self.check_cycle_algorithm(G, expected, chordless=True)
(8)                 self.check_cycle_algorithm(
(12)                    G, [c for c in expected if len(c) <= 5], length_bound=5,
chordless=True
(8)                 )
(8)                 G.add_edge(7, 3)
```

```
(8)                 expected.append([*range(3, 8)])
(8)                 expected.append([4, 3, 7, 8, 9, 10, 11])
(8)                 self.check_cycle_algorithm(G, expected, chordless=True)
(8)                 self.check_cycle_algorithm(
(12)                    G, [c for c in expected if len(c) <= 5], length_bound=5,
chordless=True
(8)                 )
(4)             def test_chordless_cycles_giant_hamiltonian(self):
(8)                 n = 1000
(8)                 assert n % 2 == 0
(8)                 G = nx.Graph()
(8)                 for v in range(n):
(12)                    if not v % 2:
(16)                        G.add_edge(v, (v + 2) % n)
(12)                    G.add_edge(v, (v + 1) % n)
(8)                 expected = [[*range(0, n, 2)]] + [
(12)                    [x % n for x in range(i, i + 3)] for i in range(0, n, 2)
(8)                 ]
(8)                 self.check_cycle_algorithm(G, expected, chordless=True)
(8)                 self.check_cycle_algorithm(
(12)                    G, [c for c in expected if len(c) <= 3], length_bound=3,
chordless=True
(8)                 )
(8)                 n = 100
(8)                 assert n % 2 == 0
(8)                 G = nx.DiGraph()
(8)                 for v in range(n):
(12)                    G.add_edge(v, (v + 1) % n)
(12)                    if not v % 2:
(16)                        G.add_edge((v + 2) % n, v)
(8)                 expected = [[*range(n - 2, -2, -2)]] + [
(12)                    [x % n for x in range(i, i + 3)] for i in range(0, n, 2)
(8)                 ]
(8)                 self.check_cycle_algorithm(G, expected, chordless=True)
(8)                 self.check_cycle_algorithm(
(12)                    G, [c for c in expected if len(c) <= 3], length_bound=3,
chordless=True
(8)                 )
(4)             def test_simple_cycles_acyclic_tournament(self):
(8)                 n = 10
(8)                 G = nx.DiGraph((x, y) for x in range(n) for y in range(x))
(8)                 self.check_cycle_algorithm(G, [])
(8)                 self.check_cycle_algorithm(G, [], chordless=True)
(8)                 for k in range(n + 1):
(12)                    self.check_cycle_algorithm(G, [], length_bound=k)
(12)                    self.check_cycle_algorithm(G, [], length_bound=k, chordless=True)
(4)             def test_simple_cycles_graph(self):
(8)                 testG = nx.cycle_graph(8)
(8)                 cyc1 = tuple(range(8))
(8)                 self.check_cycle_algorithm(testG, [cyc1])
(8)                 testG.add_edge(4, -1)
(8)                 nx.add_path(testG, [3, -2, -3, -4])
(8)                 self.check_cycle_algorithm(testG, [cyc1])
(8)                 testG.update(nx.cycle_graph(range(8, 16)))
(8)                 cyc2 = tuple(range(8, 16))
(8)                 self.check_cycle_algorithm(testG, [cyc1, cyc2])
(8)                 testG.update(nx.cycle_graph(range(4, 12)))
(8)                 cyc3 = tuple(range(4, 12))
(8)                 expected = {
(12)                    (0, 1, 2, 3, 4, 5, 6, 7),  # cyc1
(12)                    (8, 9, 10, 11, 12, 13, 14, 15),  # cyc2
(12)                    (4, 5, 6, 7, 8, 9, 10, 11),  # cyc3
(12)                    (4, 5, 6, 7, 8, 15, 14, 13, 12, 11),  # cyc2 + cyc3
(12)                    (0, 1, 2, 3, 4, 11, 10, 9, 8, 7),  # cyc1 + cyc3
(12)                    (0, 1, 2, 3, 4, 11, 12, 13, 14, 15, 8, 7),  # cyc1 + cyc2 + cyc3
(8)                 }
(8)                 self.check_cycle_algorithm(testG, expected)
(8)                 assert len(expected) == (2**3 - 1) - 1  # 1 disjoint comb: cyc1 + cyc2
(8)                 testG = nx.cycle_graph(12)
```

```
(8)                 testG.update(nx.cycle_graph([12, 10, 13, 2, 14, 4, 15, 8]).edges)
(8)                 expected = (2**5 - 1) - 11  # 11 disjoint combinations
(8)                 self.check_cycle_algorithm(testG, expected)
(4)             def test_simple_cycles_bounded(self):
(8)                 d = nx.DiGraph()
(8)                 expected = []
(8)                 for n in range(10):
(12)                    nx.add_cycle(d, range(n))
(12)                    expected.append(n)
(12)                    for k, e in enumerate(expected):
(16)                        self.check_cycle_algorithm(d, e, length_bound=k)
(8)                 g = nx.Graph()
(8)                 top = 0
(8)                 expected = []
(8)                 for n in range(10):
(12)                    expected.append(n if n < 2 else n - 1)
(12)                    if n == 2:
(16)                        continue
(12)                    nx.add_cycle(g, range(top, top + n))
(12)                    top += n
(12)                    for k, e in enumerate(expected):
(16)                        self.check_cycle_algorithm(g, e, length_bound=k)
(4)             def test_simple_cycles_bound_corner_cases(self):
(8)                 G = nx.cycle_graph(4)
(8)                 DG = nx.cycle_graph(4, create_using=nx.DiGraph)
(8)                 assert list(nx.simple_cycles(G, length_bound=0)) == []
(8)                 assert list(nx.simple_cycles(DG, length_bound=0)) == []
(8)                 assert list(nx.chordless_cycles(G, length_bound=0)) == []
(8)                 assert list(nx.chordless_cycles(DG, length_bound=0)) == []
(4)             def test_simple_cycles_bound_error(self):
(8)                 with pytest.raises(ValueError):
(12)                    G = nx.DiGraph()
(12)                    for c in nx.simple_cycles(G, -1):
(16)                        assert False
(8)                 with pytest.raises(ValueError):
(12)                    G = nx.Graph()
(12)                    for c in nx.simple_cycles(G, -1):
(16)                        assert False
(8)                 with pytest.raises(ValueError):
(12)                    G = nx.Graph()
(12)                    for c in nx.chordless_cycles(G, -1):
(16)                        assert False
(8)                 with pytest.raises(ValueError):
(12)                    G = nx.DiGraph()
(12)                    for c in nx.chordless_cycles(G, -1):
(16)                        assert False
(4)             def test_chordless_cycles_clique(self):
(8)                 g_family = [self.K(n) for n in range(2, 15)]
(8)                 expected = [0, 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286, 364]
(8)                 self.check_cycle_enumeration_integer_sequence(
(12)                    g_family, expected, chordless=True
(8)                 )
(8)                 expected = [(n * n - n) // 2 for n in range(15)]
(8)                 g_family = [self.D(n) for n in range(15)]
(8)                 self.check_cycle_enumeration_integer_sequence(
(12)                    g_family, expected, chordless=True
(8)                 )
(0)         class TestFindCycle:
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 cls.nodes = [0, 1, 2, 3]
(8)                 cls.edges = [(-1, 0), (0, 1), (1, 0), (1, 0), (2, 1), (3, 1)]
(4)             def test_graph_nocycle(self):
(8)                 G = nx.Graph(self.edges)
(8)                 pytest.raises(nx.exception.NetworkXNoCycle, nx.find_cycle, G,
self.nodes)
(4)             def test_graph_cycle(self):
(8)                 G = nx.Graph(self.edges)
(8)                 G.add_edge(2, 0)
```

```
(8)                     x = list(nx.find_cycle(G, self.nodes))
(8)                     x_ = [(0, 1), (1, 2), (2, 0)]
(8)                     assert x == x_
(4)                 def test_graph_orientation_none(self):
(8)                     G = nx.Graph(self.edges)
(8)                     G.add_edge(2, 0)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation=None))
(8)                     x_ = [(0, 1), (1, 2), (2, 0)]
(8)                     assert x == x_
(4)                 def test_graph_orientation_original(self):
(8)                     G = nx.Graph(self.edges)
(8)                     G.add_edge(2, 0)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation="original"))
(8)                     x_ = [(0, 1, FORWARD), (1, 2, FORWARD), (2, 0, FORWARD)]
(8)                     assert x == x_
(4)                 def test_digraph(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes))
(8)                     x_ = [(0, 1), (1, 0)]
(8)                     assert x == x_
(4)                 def test_digraph_orientation_none(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation=None))
(8)                     x_ = [(0, 1), (1, 0)]
(8)                     assert x == x_
(4)                 def test_digraph_orientation_original(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation="original"))
(8)                     x_ = [(0, 1, FORWARD), (1, 0, FORWARD)]
(8)                     assert x == x_
(4)                 def test_multigraph(self):
(8)                     G = nx.MultiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes))
(8)                     x_ = [(0, 1, 0), (1, 0, 1)]  # or (1, 0, 2)
(8)                     assert x[0] == x_[0]
(8)                     assert x[1][:2] == x_[1][:2]
(4)                 def test_multidigraph(self):
(8)                     G = nx.MultiDiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes))
(8)                     x_ = [(0, 1, 0), (1, 0, 0)]  # (1, 0, 1)
(8)                     assert x[0] == x_[0]
(8)                     assert x[1][:2] == x_[1][:2]
(4)                 def test_digraph_ignore(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation="ignore"))
(8)                     x_ = [(0, 1, FORWARD), (1, 0, FORWARD)]
(8)                     assert x == x_
(4)                 def test_digraph_reverse(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation="reverse"))
(8)                     x_ = [(1, 0, REVERSE), (0, 1, REVERSE)]
(8)                     assert x == x_
(4)                 def test_multidigraph_ignore(self):
(8)                     G = nx.MultiDiGraph(self.edges)
(8)                     x = list(nx.find_cycle(G, self.nodes, orientation="ignore"))
(8)                     x_ = [(0, 1, 0, FORWARD), (1, 0, 0, FORWARD)]  # or (1, 0, 1, 1)
(8)                     assert x[0] == x_[0]
(8)                     assert x[1][:2] == x_[1][:2]
(8)                     assert x[1][3] == x_[1][3]
(4)                 def test_multidigraph_ignore2(self):
(8)                     G = nx.MultiDiGraph([(0, 1), (1, 2), (1, 2)])
(8)                     x = list(nx.find_cycle(G, [0, 1, 2], orientation="ignore"))
(8)                     x_ = [(1, 2, 0, FORWARD), (1, 2, 1, REVERSE)]
(8)                     assert x == x_
(4)                 def test_multidigraph_original(self):
(8)                     G = nx.MultiDiGraph([(0, 1), (1, 2), (2, 3), (4, 2)])
(8)                     pytest.raises(
(12)                        nx.exception.NetworkXNoCycle,
(12)                        nx.find_cycle,
```

```
(12)                        G,
(12)                        [0, 1, 2, 3, 4],
(12)                        orientation="original",
(8)                     )
(4)                 def test_dag(self):
(8)                     G = nx.DiGraph([(0, 1), (0, 2), (1, 2)])
(8)                     pytest.raises(
(12)                        nx.exception.NetworkXNoCycle, nx.find_cycle, G,
orientation="original"
(8)                     )
(8)                     x = list(nx.find_cycle(G, orientation="ignore"))
(8)                     assert x == [(0, 1, FORWARD), (1, 2, FORWARD), (0, 2, REVERSE)]
(4)                 def test_prev_explored(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edges_from([(1, 0), (2, 0), (1, 2), (2, 1)])
(8)                     pytest.raises(nx.NetworkXNoCycle, nx.find_cycle, G, source=0)
(8)                     x = list(nx.find_cycle(G, 1))
(8)                     x_ = [(1, 2), (2, 1)]
(8)                     assert x == x_
(8)                     x = list(nx.find_cycle(G, 2))
(8)                     x_ = [(2, 1), (1, 2)]
(8)                     assert x == x_
(8)                     x = list(nx.find_cycle(G))
(8)                     x_ = [(1, 2), (2, 1)]
(8)                     assert x == x_
(4)                 def test_no_cycle(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edges_from([(1, 2), (2, 0), (3, 1), (3, 2)])
(8)                     pytest.raises(nx.NetworkXNoCycle, nx.find_cycle, G, source=0)
(8)                     pytest.raises(nx.NetworkXNoCycle, nx.find_cycle, G)
(0)             def assert_basis_equal(a, b):
(4)                 assert sorted(a) == sorted(b)
(0)             class TestMinimumCycleBasis:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     T = nx.Graph()
(8)                     nx.add_cycle(T, [1, 2, 3, 4], weight=1)
(8)                     T.add_edge(2, 4, weight=5)
(8)                     cls.diamond_graph = T
(4)                 def test_unweighted_diamond(self):
(8)                     mcb = nx.minimum_cycle_basis(self.diamond_graph)
(8)                     assert_basis_equal(mcb, [[2, 4, 1], [3, 4, 2]])
(4)                 def test_weighted_diamond(self):
(8)                     mcb = nx.minimum_cycle_basis(self.diamond_graph, weight="weight")
(8)                     assert_basis_equal(mcb, [[2, 4, 1], [4, 3, 2, 1]])
(4)                 def test_dimensionality(self):
(8)                     ntrial = 10
(8)                     for seed in range(1234, 1234 + ntrial):
(12)                        rg = nx.erdos_renyi_graph(10, 0.3, seed=seed)
(12)                        nnodes = rg.number_of_nodes()
(12)                        nedges = rg.number_of_edges()
(12)                        ncomp = nx.number_connected_components(rg)
(12)                        mcb = nx.minimum_cycle_basis(rg)
(12)                        assert len(mcb) == nedges - nnodes + ncomp
(12)                        check_independent(mcb)
(4)                 def test_complete_graph(self):
(8)                     cg = nx.complete_graph(5)
(8)                     mcb = nx.minimum_cycle_basis(cg)
(8)                     assert all(len(cycle) == 3 for cycle in mcb)
(8)                     check_independent(mcb)
(4)                 def test_tree_graph(self):
(8)                     tg = nx.balanced_tree(3, 3)
(8)                     assert not nx.minimum_cycle_basis(tg)
(4)                 def test_petersen_graph(self):
(8)                     G = nx.petersen_graph()
(8)                     mcb = list(nx.minimum_cycle_basis(G))
(8)                     expected = [
(12)                        [4, 9, 7, 5, 0],
(12)                        [1, 2, 3, 4, 0],
```

```
691: (12)                        [1, 6, 8, 5, 0],
692: (12)                        [4, 3, 8, 5, 0],
693: (12)                        [1, 6, 9, 4, 0],
694: (12)                        [1, 2, 7, 5, 0],
695: (8)                    ]
696: (8)                    assert len(mcb) == len(expected)
697: (8)                    assert all(c in expected for c in mcb)
698: (8)                    for c in mcb:
699: (12)                        assert all(G.has_edge(u, v) for u, v in nx.utils.pairwise(c,
cyclic=True))
700: (8)                    check_independent(mcb)
701: (4)                def test_gh6787_variable_weighted_complete_graph(self):
702: (8)                    N = 8
703: (8)                    cg = nx.complete_graph(N)
704: (8)                    cg.add_weighted_edges_from([(u, v, 9) for u, v in cg.edges])
705: (8)                    cg.add_weighted_edges_from([(u, v, 1) for u, v in
nx.cycle_graph(N).edges])
706: (8)                    mcb = nx.minimum_cycle_basis(cg, weight="weight")
707: (8)                    check_independent(mcb)
708: (4)                def test_gh6787_and_edge_attribute_names(self):
709: (8)                    G = nx.cycle_graph(4)
710: (8)                    G.add_weighted_edges_from([(0, 2, 10), (1, 3, 10)], weight="dist")
711: (8)                    expected = [[1, 3, 0], [3, 2, 1, 0], [1, 2, 0]]
712: (8)                    mcb = list(nx.minimum_cycle_basis(G, weight="dist"))
713: (8)                    assert len(mcb) == len(expected)
714: (8)                    assert all(c in expected for c in mcb)
715: (8)                    expected = [[1, 3, 0], [1, 2, 0], [3, 2, 0]]
716: (8)                    mcb = list(nx.minimum_cycle_basis(G))
717: (8)                    assert len(mcb) == len(expected)
718: (8)                    assert all(c in expected for c in mcb)
719: (0)            class TestGirth:
720: (4)                @pytest.mark.parametrize(
721: (8)                    ("G", "expected"),
722: (8)                    (
723: (12)                        (nx.chvatal_graph(), 4),
724: (12)                        (nx.tutte_graph(), 4),
725: (12)                        (nx.petersen_graph(), 5),
726: (12)                        (nx.heawood_graph(), 6),
727: (12)                        (nx.pappus_graph(), 6),
728: (12)                        (nx.random_tree(10, seed=42), inf),
729: (12)                        (nx.empty_graph(10), inf),
730: (12)                        (nx.Graph(chain(cycle_edges(range(5)), cycle_edges(range(6,
10)))), 4),
731: (12)                        (
732: (16)                            nx.Graph(
733: (20)                                [
734: (24)                                    (0, 6),
735: (24)                                    (0, 8),
736: (24)                                    (0, 9),
737: (24)                                    (1, 8),
738: (24)                                    (2, 8),
739: (24)                                    (2, 9),
740: (24)                                    (4, 9),
741: (24)                                    (5, 9),
742: (24)                                    (6, 8),
743: (24)                                    (6, 9),
744: (24)                                    (7, 8),
745: (20)                                ]
746: (16)                            ),
747: (16)                            3,
748: (12)                        ),
749: (8)                    ),
750: (4)                )
751: (4)                def test_girth(self, G, expected):
752: (8)                    assert nx.girth(G) == expected

----------------------------------------

File 319 - test_dag.py:
```

```
(0)  from collections import deque
(0)  from itertools import combinations, permutations
(0)  import pytest
(0)  import networkx as nx
(0)  from networkx.utils import edges_equal, pairwise
(0)  def _consume(iterator):
(4)      "Consume the iterator entirely."
(4)      deque(iterator, maxlen=0)
(0)  class TestDagLongestPath:
(4)      """Unit tests computing the longest path in a directed acyclic graph."""
(4)      def test_empty(self):
(8)          G = nx.DiGraph()
(8)          assert nx.dag_longest_path(G) == []
(4)      def test_unweighted1(self):
(8)          edges = [(1, 2), (2, 3), (2, 4), (3, 5), (5, 6), (3, 7)]
(8)          G = nx.DiGraph(edges)
(8)          assert nx.dag_longest_path(G) == [1, 2, 3, 5, 6]
(4)      def test_unweighted2(self):
(8)          edges = [(1, 2), (2, 3), (3, 4), (4, 5), (1, 3), (1, 5), (3, 5)]
(8)          G = nx.DiGraph(edges)
(8)          assert nx.dag_longest_path(G) == [1, 2, 3, 4, 5]
(4)      def test_weighted(self):
(8)          G = nx.DiGraph()
(8)          edges = [(1, 2, -5), (2, 3, 1), (3, 4, 1), (4, 5, 0), (3, 5, 4), (1,
6, 2)]

(8)          G.add_weighted_edges_from(edges)
(8)          assert nx.dag_longest_path(G) == [2, 3, 5]
(4)      def test_undirected_not_implemented(self):
(8)          G = nx.Graph()
(8)          pytest.raises(nx.NetworkXNotImplemented, nx.dag_longest_path, G)
(4)      def test_unorderable_nodes(self):
(8)          """Tests that computing the longest path does not depend on
(8)          nodes being orderable.
(8)          For more information, see issue #1989.
(8)          """
(8)          nodes = [object() for n in range(4)]
(8)          G = nx.DiGraph()
(8)          G.add_edge(nodes[0], nodes[1])
(8)          G.add_edge(nodes[0], nodes[2])
(8)          G.add_edge(nodes[2], nodes[3])
(8)          G.add_edge(nodes[1], nodes[3])
(8)          nx.dag_longest_path(G)
(4)      def test_multigraph_unweighted(self):
(8)          edges = [(1, 2), (2, 3), (2, 3), (3, 4), (4, 5), (1, 3), (1, 5), (3,
5)]

(8)          G = nx.MultiDiGraph(edges)
(8)          assert nx.dag_longest_path(G) == [1, 2, 3, 4, 5]
(4)      def test_multigraph_weighted(self):
(8)          G = nx.MultiDiGraph()
(8)          edges = [
(12)             (1, 2, 2),
(12)             (2, 3, 2),
(12)             (1, 3, 1),
(12)             (1, 3, 5),
(12)             (1, 3, 2),
(8)          ]
(8)          G.add_weighted_edges_from(edges)
(8)          assert nx.dag_longest_path(G) == [1, 3]
(4)      def test_multigraph_weighted_default_weight(self):
(8)          G = nx.MultiDiGraph([(1, 2), (2, 3)])  # Unweighted edges
(8)          G.add_weighted_edges_from([(1, 3, 1), (1, 3, 5), (1, 3, 2)])
(8)          assert nx.dag_longest_path(G) == [1, 3]
(8)          assert nx.dag_longest_path(G, default_weight=3) == [1, 2, 3]
(0)  class TestDagLongestPathLength:
(4)      """Unit tests for computing the length of a longest path in a
(4)      directed acyclic graph.
(4)      """
(4)      def test_unweighted(self):
```

```
67: (8)                     edges = [(1, 2), (2, 3), (2, 4), (3, 5), (5, 6), (5, 7)]
68: (8)                     G = nx.DiGraph(edges)
69: (8)                     assert nx.dag_longest_path_length(G) == 4
70: (8)                     edges = [(1, 2), (2, 3), (3, 4), (4, 5), (1, 3), (1, 5), (3, 5)]
71: (8)                     G = nx.DiGraph(edges)
72: (8)                     assert nx.dag_longest_path_length(G) == 4
73: (8)                     G = nx.DiGraph()
74: (8)                     G.add_node(1)
75: (8)                     assert nx.dag_longest_path_length(G) == 0
76: (4)                 def test_undirected_not_implemented(self):
77: (8)                     G = nx.Graph()
78: (8)                     pytest.raises(nx.NetworkXNotImplemented, nx.dag_longest_path_length,
G)
79: (4)                 def test_weighted(self):
80: (8)                     edges = [(1, 2, -5), (2, 3, 1), (3, 4, 1), (4, 5, 0), (3, 5, 4), (1,
6, 2)]
81: (8)                     G = nx.DiGraph()
82: (8)                     G.add_weighted_edges_from(edges)
83: (8)                     assert nx.dag_longest_path_length(G) == 5
84: (4)                 def test_multigraph_unweighted(self):
85: (8)                     edges = [(1, 2), (2, 3), (2, 3), (3, 4), (4, 5), (1, 3), (1, 5), (3,
5)]
86: (8)                     G = nx.MultiDiGraph(edges)
87: (8)                     assert nx.dag_longest_path_length(G) == 4
88: (4)                 def test_multigraph_weighted(self):
89: (8)                     G = nx.MultiDiGraph()
90: (8)                     edges = [
91: (12)                        (1, 2, 2),
92: (12)                        (2, 3, 2),
93: (12)                        (1, 3, 1),
94: (12)                        (1, 3, 5),
95: (12)                        (1, 3, 2),
96: (8)                     ]
97: (8)                     G.add_weighted_edges_from(edges)
98: (8)                     assert nx.dag_longest_path_length(G) == 5
99: (0)             class TestDAG:
100: (4)                @classmethod
101: (4)                def setup_class(cls):
102: (8)                    pass
103: (4)                def test_topological_sort1(self):
104: (8)                    DG = nx.DiGraph([(1, 2), (1, 3), (2, 3)])
105: (8)                    for algorithm in [nx.topological_sort,
nx.lexicographical_topological_sort]:
106: (12)                       assert tuple(algorithm(DG)) == (1, 2, 3)
107: (8)                    DG.add_edge(3, 2)
108: (8)                    for algorithm in [nx.topological_sort,
nx.lexicographical_topological_sort]:
109: (12)                       pytest.raises(nx.NetworkXUnfeasible, _consume, algorithm(DG))
110: (8)                    DG.remove_edge(2, 3)
111: (8)                    for algorithm in [nx.topological_sort,
nx.lexicographical_topological_sort]:
112: (12)                       assert tuple(algorithm(DG)) == (1, 3, 2)
113: (8)                    DG.remove_edge(3, 2)
114: (8)                    assert tuple(nx.topological_sort(DG)) in {(1, 2, 3), (1, 3, 2)}
115: (8)                    assert tuple(nx.lexicographical_topological_sort(DG)) == (1, 2, 3)
116: (4)                def test_is_directed_acyclic_graph(self):
117: (8)                    G = nx.generators.complete_graph(2)
118: (8)                    assert not nx.is_directed_acyclic_graph(G)
119: (8)                    assert not nx.is_directed_acyclic_graph(G.to_directed())
120: (8)                    assert not nx.is_directed_acyclic_graph(nx.Graph([(3, 4), (4, 5)]))
121: (8)                    assert nx.is_directed_acyclic_graph(nx.DiGraph([(3, 4), (4, 5)]))
122: (4)                def test_topological_sort2(self):
123: (8)                    DG = nx.DiGraph(
124: (12)                       {
125: (16)                           1: [2],
126: (16)                           2: [3],
127: (16)                           3: [4],
128: (16)                           4: [5],
129: (16)                           5: [1],
```

```
130: (16)                           11: [12],
131: (16)                           12: [13],
132: (16)                           13: [14],
133: (16)                           14: [15],
134: (12)                       }
135: (8)                    )
136: (8)                    pytest.raises(nx.NetworkXUnfeasible, _consume,
nx.topological_sort(DG))
137: (8)                    assert not nx.is_directed_acyclic_graph(DG)
138: (8)                    DG.remove_edge(1, 2)
139: (8)                    _consume(nx.topological_sort(DG))
140: (8)                    assert nx.is_directed_acyclic_graph(DG)
141: (4)                def test_topological_sort3(self):
142: (8)                    DG = nx.DiGraph()
143: (8)                    DG.add_edges_from([(1, i) for i in range(2, 5)])
144: (8)                    DG.add_edges_from([(2, i) for i in range(5, 9)])
145: (8)                    DG.add_edges_from([(6, i) for i in range(9, 12)])
146: (8)                    DG.add_edges_from([(4, i) for i in range(12, 15)])
147: (8)                    def validate(order):
148: (12)                       assert isinstance(order, list)
149: (12)                       assert set(order) == set(DG)
150: (12)                       for u, v in combinations(order, 2):
151: (16)                           assert not nx.has_path(DG, v, u)
152: (8)                    validate(list(nx.topological_sort(DG)))
153: (8)                    DG.add_edge(14, 1)
154: (8)                    pytest.raises(nx.NetworkXUnfeasible, _consume,
nx.topological_sort(DG))
155: (4)                def test_topological_sort4(self):
156: (8)                    G = nx.Graph()
157: (8)                    G.add_edge(1, 2)
158: (8)                    pytest.raises(nx.NetworkXError, _consume, nx.topological_sort(G))
159: (4)                def test_topological_sort5(self):
160: (8)                    G = nx.DiGraph()
161: (8)                    G.add_edge(0, 1)
162: (8)                    assert list(nx.topological_sort(G)) == [0, 1]
163: (4)                def test_topological_sort6(self):
164: (8)                    for algorithm in [nx.topological_sort,
nx.lexicographical_topological_sort]:
165: (12)                       def runtime_error():
166: (16)                           DG = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
167: (16)                           first = True
168: (16)                           for x in algorithm(DG):
169: (20)                               if first:
170: (24)                                   first = False
171: (24)                                   DG.add_edge(5 - x, 5)
172: (12)                       def unfeasible_error():
173: (16)                           DG = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
174: (16)                           first = True
175: (16)                           for x in algorithm(DG):
176: (20)                               if first:
177: (24)                                   first = False
178: (24)                                   DG.remove_node(4)
179: (12)                       def runtime_error2():
180: (16)                           DG = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
181: (16)                           first = True
182: (16)                           for x in algorithm(DG):
183: (20)                               if first:
184: (24)                                   first = False
185: (24)                                   DG.remove_node(2)
186: (12)                       pytest.raises(RuntimeError, runtime_error)
187: (12)                       pytest.raises(RuntimeError, runtime_error2)
188: (12)                       pytest.raises(nx.NetworkXUnfeasible, unfeasible_error)
189: (4)                def test_all_topological_sorts_1(self):
190: (8)                    DG = nx.DiGraph([(1, 2), (2, 3), (3, 4), (4, 5)])
191: (8)                    assert list(nx.all_topological_sorts(DG)) == [[1, 2, 3, 4, 5]]
192: (4)                def test_all_topological_sorts_2(self):
193: (8)                    DG = nx.DiGraph([(1, 3), (2, 1), (2, 4), (4, 3), (4, 5)])
194: (8)                    assert sorted(nx.all_topological_sorts(DG)) == [
195: (12)                       [2, 1, 4, 3, 5],
```

```
196: (12)                        [2, 1, 4, 5, 3],
197: (12)                        [2, 4, 1, 3, 5],
198: (12)                        [2, 4, 1, 5, 3],
199: (12)                        [2, 4, 5, 1, 3],
200: (8)                    ]
201: (4)                def test_all_topological_sorts_3(self):
202: (8)                    def unfeasible():
203: (12)                        DG = nx.DiGraph([(1, 2), (2, 3), (3, 4), (4, 2), (4, 5)])
204: (12)                        list(nx.all_topological_sorts(DG))
205: (8)                    def not_implemented():
206: (12)                        G = nx.Graph([(1, 2), (2, 3)])
207: (12)                        list(nx.all_topological_sorts(G))
208: (8)                    def not_implemented_2():
209: (12)                        G = nx.MultiGraph([(1, 2), (1, 2), (2, 3)])
210: (12)                        list(nx.all_topological_sorts(G))
211: (8)                    pytest.raises(nx.NetworkXUnfeasible, unfeasible)
212: (8)                    pytest.raises(nx.NetworkXNotImplemented, not_implemented)
213: (8)                    pytest.raises(nx.NetworkXNotImplemented, not_implemented_2)
214: (4)                def test_all_topological_sorts_4(self):
215: (8)                    DG = nx.DiGraph()
216: (8)                    for i in range(7):
217: (12)                        DG.add_node(i)
218: (8)                    assert sorted(map(list, permutations(DG.nodes))) == sorted(
219: (12)                        nx.all_topological_sorts(DG)
220: (8)                    )
221: (4)                def test_all_topological_sorts_multigraph_1(self):
222: (8)                    DG = nx.MultiDiGraph([(1, 2), (1, 2), (2, 3), (3, 4), (3, 5), (3, 5),
(3, 5)])
223: (8)                    assert sorted(nx.all_topological_sorts(DG)) == sorted(
224: (12)                        [[1, 2, 3, 4, 5], [1, 2, 3, 5, 4]]
225: (8)                    )
226: (4)                def test_all_topological_sorts_multigraph_2(self):
227: (8)                    N = 9
228: (8)                    edges = []
229: (8)                    for i in range(1, N):
230: (12)                        edges.extend([(i, i + 1)] * i)
231: (8)                    DG = nx.MultiDiGraph(edges)
232: (8)                    assert list(nx.all_topological_sorts(DG)) == [list(range(1, N + 1))]
233: (4)                def test_ancestors(self):
234: (8)                    G = nx.DiGraph()
235: (8)                    ancestors = nx.algorithms.dag.ancestors
236: (8)                    G.add_edges_from([(1, 2), (1, 3), (4, 2), (4, 3), (4, 5), (2, 6), (5,
6)])
237: (8)                    assert ancestors(G, 6) == {1, 2, 4, 5}
238: (8)                    assert ancestors(G, 3) == {1, 4}
239: (8)                    assert ancestors(G, 1) == set()
240: (8)                    pytest.raises(nx.NetworkXError, ancestors, G, 8)
241: (4)                def test_descendants(self):
242: (8)                    G = nx.DiGraph()
243: (8)                    descendants = nx.algorithms.dag.descendants
244: (8)                    G.add_edges_from([(1, 2), (1, 3), (4, 2), (4, 3), (4, 5), (2, 6), (5,
6)])
245: (8)                    assert descendants(G, 1) == {2, 3, 6}
246: (8)                    assert descendants(G, 4) == {2, 3, 5, 6}
247: (8)                    assert descendants(G, 3) == set()
248: (8)                    pytest.raises(nx.NetworkXError, descendants, G, 8)
249: (4)                def test_transitive_closure(self):
250: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
251: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
252: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
253: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (2, 4)])
254: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)]
255: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
256: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
257: (8)                    solution = [(1, 2), (2, 1), (2, 3), (3, 2), (1, 3), (3, 1)]
258: (8)                    soln = sorted(solution + [(n, n) for n in G])
259: (8)                    assert edges_equal(sorted(nx.transitive_closure(G).edges()), soln)
260: (8)                    G = nx.Graph([(1, 2), (2, 3), (3, 4)])
261: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

```
262: (8)                    assert edges_equal(sorted(nx.transitive_closure(G).edges()), solution)
263: (8)                    G = nx.MultiGraph([(1, 2), (2, 3), (3, 4)])
264: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
265: (8)                    assert edges_equal(sorted(nx.transitive_closure(G).edges()), solution)
266: (8)                    G = nx.MultiDiGraph([(1, 2), (2, 3), (3, 4)])
267: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
268: (8)                    assert edges_equal(sorted(nx.transitive_closure(G).edges()), solution)
269: (8)                    G = nx.DiGraph([(1, 2, {"a": 3}), (2, 3, {"b": 0}), (3, 4)])
270: (8)                    H = nx.transitive_closure(G)
271: (8)                    for u, v in G.edges():
272: (12)                       assert G.get_edge_data(u, v) == H.get_edge_data(u, v)
273: (8)                    k = 10
274: (8)                    G = nx.DiGraph((i, i + 1, {"f": "b", "weight": i}) for i in range(k))
275: (8)                    H = nx.transitive_closure(G)
276: (8)                    for u, v in G.edges():
277: (12)                       assert G.get_edge_data(u, v) == H.get_edge_data(u, v)
278: (8)                    G = nx.Graph()
279: (8)                    with pytest.raises(nx.NetworkXError):
280: (12)                       nx.transitive_closure(G, reflexive="wrong input")
281: (4)                def test_reflexive_transitive_closure(self):
282: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
283: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
284: (8)                    soln = sorted(solution + [(n, n) for n in G])
285: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
286: (8)                    assert edges_equal(nx.transitive_closure(G, False).edges(), solution)
287: (8)                    assert edges_equal(nx.transitive_closure(G, True).edges(), soln)
288: (8)                    assert edges_equal(nx.transitive_closure(G, None).edges(), solution)
289: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (2, 4)])
290: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)]
291: (8)                    soln = sorted(solution + [(n, n) for n in G])
292: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
293: (8)                    assert edges_equal(nx.transitive_closure(G, False).edges(), solution)
294: (8)                    assert edges_equal(nx.transitive_closure(G, True).edges(), soln)
295: (8)                    assert edges_equal(nx.transitive_closure(G, None).edges(), solution)
296: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
297: (8)                    solution = sorted([(1, 2), (2, 1), (2, 3), (3, 2), (1, 3), (3, 1)])
298: (8)                    soln = sorted(solution + [(n, n) for n in G])
299: (8)                    assert edges_equal(sorted(nx.transitive_closure(G).edges()), soln)
300: (8)                    assert edges_equal(sorted(nx.transitive_closure(G, False).edges()),
soln)
301: (8)                    assert edges_equal(sorted(nx.transitive_closure(G, None).edges()),
solution)
302: (8)                    assert edges_equal(sorted(nx.transitive_closure(G, True).edges()),
soln)
303: (8)                    G = nx.Graph([(1, 2), (2, 3), (3, 4)])
304: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
305: (8)                    soln = sorted(solution + [(n, n) for n in G])
306: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
307: (8)                    assert edges_equal(nx.transitive_closure(G, False).edges(), solution)
308: (8)                    assert edges_equal(nx.transitive_closure(G, True).edges(), soln)
309: (8)                    assert edges_equal(nx.transitive_closure(G, None).edges(), solution)
310: (8)                    G = nx.MultiGraph([(1, 2), (2, 3), (3, 4)])
311: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
312: (8)                    soln = sorted(solution + [(n, n) for n in G])
313: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
314: (8)                    assert edges_equal(nx.transitive_closure(G, False).edges(), solution)
315: (8)                    assert edges_equal(nx.transitive_closure(G, True).edges(), soln)
316: (8)                    assert edges_equal(nx.transitive_closure(G, None).edges(), solution)
317: (8)                    G = nx.MultiDiGraph([(1, 2), (2, 3), (3, 4)])
318: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
319: (8)                    soln = sorted(solution + [(n, n) for n in G])
320: (8)                    assert edges_equal(nx.transitive_closure(G).edges(), solution)
321: (8)                    assert edges_equal(nx.transitive_closure(G, False).edges(), solution)
322: (8)                    assert edges_equal(nx.transitive_closure(G, True).edges(), soln)
323: (8)                    assert edges_equal(nx.transitive_closure(G, None).edges(), solution)
324: (4)                def test_transitive_closure_dag(self):
325: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
326: (8)                    transitive_closure = nx.algorithms.dag.transitive_closure_dag
327: (8)                    solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

```
(8)                     assert edges_equal(transitive_closure(G).edges(), solution)
(8)                     G = nx.DiGraph([(1, 2), (2, 3), (2, 4)])
(8)                     solution = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)]
(8)                     assert edges_equal(transitive_closure(G).edges(), solution)
(8)                     G = nx.Graph([(1, 2), (2, 3), (3, 4)])
(8)                     pytest.raises(nx.NetworkXNotImplemented, transitive_closure, G)
(8)                     G = nx.DiGraph([(1, 2, {"a": 3}), (2, 3, {"b": 0}), (3, 4)])
(8)                     H = transitive_closure(G)
(8)                     for u, v in G.edges():
(12)                        assert G.get_edge_data(u, v) == H.get_edge_data(u, v)
(8)                     k = 10
(8)                     G = nx.DiGraph((i, i + 1, {"foo": "bar", "weight": i}) for i in
range(k))
(8)                     H = transitive_closure(G)
(8)                     for u, v in G.edges():
(12)                        assert G.get_edge_data(u, v) == H.get_edge_data(u, v)
(4)                 def test_transitive_reduction(self):
(8)                     G = nx.DiGraph([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])
(8)                     transitive_reduction = nx.algorithms.dag.transitive_reduction
(8)                     solution = [(1, 2), (2, 3), (3, 4)]
(8)                     assert edges_equal(transitive_reduction(G).edges(), solution)
(8)                     G = nx.DiGraph([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)])
(8)                     transitive_reduction = nx.algorithms.dag.transitive_reduction
(8)                     solution = [(1, 2), (2, 3), (2, 4)]
(8)                     assert edges_equal(transitive_reduction(G).edges(), solution)
(8)                     G = nx.Graph([(1, 2), (2, 3), (3, 4)])
(8)                     pytest.raises(nx.NetworkXNotImplemented, transitive_reduction, G)
(4)                 def _check_antichains(self, solution, result):
(8)                     sol = [frozenset(a) for a in solution]
(8)                     res = [frozenset(a) for a in result]
(8)                     assert set(sol) == set(res)
(4)                 def test_antichains(self):
(8)                     antichains = nx.algorithms.dag.antichains
(8)                     G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
(8)                     solution = [[], [4], [3], [2], [1]]
(8)                     self._check_antichains(list(antichains(G)), solution)
(8)                     G = nx.DiGraph([(1, 2), (2, 3), (2, 4), (3, 5), (5, 6), (5, 7)])
(8)                     solution = [
(12)                        [],
(12)                        [4],
(12)                        [7],
(12)                        [7, 4],
(12)                        [6],
(12)                        [6, 4],
(12)                        [6, 7],
(12)                        [6, 7, 4],
(12)                        [5],
(12)                        [5, 4],
(12)                        [3],
(12)                        [3, 4],
(12)                        [2],
(12)                        [1],
(8)                     ]
(8)                     self._check_antichains(list(antichains(G)), solution)
(8)                     G = nx.DiGraph([(1, 2), (1, 3), (3, 4), (3, 5), (5, 6)])
(8)                     solution = [
(12)                        [],
(12)                        [6],
(12)                        [5],
(12)                        [4],
(12)                        [4, 6],
(12)                        [4, 5],
(12)                        [3],
(12)                        [2],
(12)                        [2, 6],
(12)                        [2, 5],
(12)                        [2, 4],
(12)                        [2, 4, 6],
(12)                        [2, 4, 5],
```

```
396: (12)                        [2, 3],
397: (12)                        [1],
398: (8)                    ]
399: (8)                    self._check_antichains(list(antichains(G)), solution)
400: (8)                    G = nx.DiGraph({0: [1, 2], 1: [4], 2: [3], 3: [4]})
401: (8)                    solution = [[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]
402: (8)                    self._check_antichains(list(antichains(G)), solution)
403: (8)                    G = nx.DiGraph()
404: (8)                    self._check_antichains(list(antichains(G)), [[]])
405: (8)                    G = nx.DiGraph()
406: (8)                    G.add_nodes_from([0, 1, 2])
407: (8)                    solution = [[], [0], [1], [1, 0], [2], [2, 0], [2, 1], [2, 1, 0]]
408: (8)                    self._check_antichains(list(antichains(G)), solution)
409: (8)                    def f(x):
410: (12)                       return list(antichains(x))
411: (8)                    G = nx.Graph([(1, 2), (2, 3), (3, 4)])
412: (8)                    pytest.raises(nx.NetworkXNotImplemented, f, G)
413: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
414: (8)                    pytest.raises(nx.NetworkXUnfeasible, f, G)
415: (4)                def test_lexicographical_topological_sort(self):
416: (8)                    G = nx.DiGraph([(1, 2), (2, 3), (1, 4), (1, 5), (2, 6)])
417: (8)                    assert list(nx.lexicographical_topological_sort(G)) == [1, 2, 3, 4, 5,
6]
418: (8)                    assert list(nx.lexicographical_topological_sort(G, key=lambda x: x))
== [
419: (12)                       1,
420: (12)                       2,
421: (12)                       3,
422: (12)                       4,
423: (12)                       5,
424: (12)                       6,
425: (8)                    ]
426: (8)                    assert list(nx.lexicographical_topological_sort(G, key=lambda x: -x))
== [
427: (12)                       1,
428: (12)                       5,
429: (12)                       4,
430: (12)                       2,
431: (12)                       6,
432: (12)                       3,
433: (8)                    ]
434: (4)                def test_lexicographical_topological_sort2(self):
435: (8)                    """
436: (8)                    Check the case of two or more nodes with same key value.
437: (8)                    Want to avoid exception raised due to comparing nodes directly.
438: (8)                    See Issue #3493
439: (8)                    """
440: (8)                    class Test_Node:
441: (12)                       def __init__(self, n):
442: (16)                           self.label = n
443: (16)                           self.priority = 1
444: (12)                       def __repr__(self):
445: (16)                           return f"Node({self.label})"
446: (8)                    def sorting_key(node):
447: (12)                       return node.priority
448: (8)                    test_nodes = [Test_Node(n) for n in range(4)]
449: (8)                    G = nx.DiGraph()
450: (8)                    edges = [(0, 1), (0, 2), (0, 3), (2, 3)]
451: (8)                    G.add_edges_from((test_nodes[a], test_nodes[b]) for a, b in edges)
452: (8)                    sorting = list(nx.lexicographical_topological_sort(G,
key=sorting_key))
453: (8)                    assert sorting == test_nodes
454: (0)            def test_topological_generations():
455: (4)                G = nx.DiGraph(
456: (8)                    {1: [2, 3], 2: [4, 5], 3: [7], 4: [], 5: [6, 7], 6: [], 7: []}
457: (4)                ).reverse()
458: (4)                generations = [sorted(gen) for gen in nx.topological_generations(G)]
459: (4)                expected = [[4, 6, 7], [3, 5], [2], [1]]
460: (4)                assert generations == expected
```

```
(4)                 MG = nx.MultiDiGraph(G.edges)
(4)                 MG.add_edge(2, 1)
(4)                 generations = [sorted(gen) for gen in nx.topological_generations(MG)]
(4)                 assert generations == expected
(0)             def test_topological_generations_empty():
(4)                 G = nx.DiGraph()
(4)                 assert list(nx.topological_generations(G)) == []
(0)             def test_topological_generations_cycle():
(4)                 G = nx.DiGraph([[2, 1], [3, 1], [1, 2]])
(4)                 with pytest.raises(nx.NetworkXUnfeasible):
(8)                     list(nx.topological_generations(G))
(0)             def test_is_aperiodic_cycle():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [1, 2, 3, 4])
(4)                 assert not nx.is_aperiodic(G)
(0)             def test_is_aperiodic_cycle2():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [1, 2, 3, 4])
(4)                 nx.add_cycle(G, [3, 4, 5, 6, 7])
(4)                 assert nx.is_aperiodic(G)
(0)             def test_is_aperiodic_cycle3():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [1, 2, 3, 4])
(4)                 nx.add_cycle(G, [3, 4, 5, 6])
(4)                 assert not nx.is_aperiodic(G)
(0)             def test_is_aperiodic_cycle4():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [1, 2, 3, 4])
(4)                 G.add_edge(1, 3)
(4)                 assert nx.is_aperiodic(G)
(0)             def test_is_aperiodic_selfloop():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [1, 2, 3, 4])
(4)                 G.add_edge(1, 1)
(4)                 assert nx.is_aperiodic(G)
(0)             def test_is_aperiodic_undirected_raises():
(4)                 G = nx.Graph()
(4)                 pytest.raises(nx.NetworkXError, nx.is_aperiodic, G)
(0)             def test_is_aperiodic_empty_graph():
(4)                 G = nx.empty_graph(create_using=nx.DiGraph)
(4)                 with pytest.raises(nx.NetworkXPointlessConcept, match="Graph has no
nodes."):
(8)                     nx.is_aperiodic(G)
(0)             def test_is_aperiodic_bipartite():
(4)                 G = nx.DiGraph(nx.davis_southern_women_graph())
(4)                 assert not nx.is_aperiodic(G)
(0)             def test_is_aperiodic_rary_tree():
(4)                 G = nx.full_rary_tree(3, 27, create_using=nx.DiGraph())
(4)                 assert not nx.is_aperiodic(G)
(0)             def test_is_aperiodic_disconnected():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [1, 2, 3, 4])
(4)                 nx.add_cycle(G, [5, 6, 7, 8])
(4)                 assert not nx.is_aperiodic(G)
(4)                 G.add_edge(1, 3)
(4)                 G.add_edge(5, 7)
(4)                 assert nx.is_aperiodic(G)
(0)             def test_is_aperiodic_disconnected2():
(4)                 G = nx.DiGraph()
(4)                 nx.add_cycle(G, [0, 1, 2])
(4)                 G.add_edge(3, 3)
(4)                 assert not nx.is_aperiodic(G)
(0)             class TestDagToBranching:
(4)                 """Unit tests for the :func:`networkx.dag_to_branching` function."""
(4)                 def test_single_root(self):
(8)                     """Tests that a directed acyclic graph with a single degree
(8)                     zero node produces an arborescence.
(8)                     """
(8)                     G = nx.DiGraph([(0, 1), (0, 2), (1, 3), (2, 3)])
```

```
(8)                     B = nx.dag_to_branching(G)
(8)                     expected = nx.DiGraph([(0, 1), (1, 3), (0, 2), (2, 4)])
(8)                     assert nx.is_arborescence(B)
(8)                     assert nx.is_isomorphic(B, expected)
(4)                 def test_multiple_roots(self):
(8)                     """Tests that a directed acyclic graph with multiple degree zero
(8)                     nodes creates an arborescence with multiple (weakly) connected
(8)                     components.
(8)                     """
(8)                     G = nx.DiGraph([(0, 1), (0, 2), (1, 3), (2, 3), (5, 2)])
(8)                     B = nx.dag_to_branching(G)
(8)                     expected = nx.DiGraph([(0, 1), (1, 3), (0, 2), (2, 4), (5, 6), (6,
7)])
(8)                     assert nx.is_branching(B)
(8)                     assert not nx.is_arborescence(B)
(8)                     assert nx.is_isomorphic(B, expected)
(4)                 def test_already_arborescence(self):
(8)                     """Tests that a directed acyclic graph that is already an
(8)                     arborescence produces an isomorphic arborescence as output.
(8)                     """
(8)                     A = nx.balanced_tree(2, 2, create_using=nx.DiGraph())
(8)                     B = nx.dag_to_branching(A)
(8)                     assert nx.is_isomorphic(A, B)
(4)                 def test_already_branching(self):
(8)                     """Tests that a directed acyclic graph that is already a
(8)                     branching produces an isomorphic branching as output.
(8)                     """
(8)                     T1 = nx.balanced_tree(2, 2, create_using=nx.DiGraph())
(8)                     T2 = nx.balanced_tree(2, 2, create_using=nx.DiGraph())
(8)                     G = nx.disjoint_union(T1, T2)
(8)                     B = nx.dag_to_branching(G)
(8)                     assert nx.is_isomorphic(G, B)
(4)                 def test_not_acyclic(self):
(8)                     """Tests that a non-acyclic graph causes an exception."""
(8)                     with pytest.raises(nx.HasACycle):
(12)                        G = nx.DiGraph(pairwise("abc", cyclic=True))
(12)                        nx.dag_to_branching(G)
(4)                 def test_undirected(self):
(8)                     with pytest.raises(nx.NetworkXNotImplemented):
(12)                        nx.dag_to_branching(nx.Graph())
(4)                 def test_multigraph(self):
(8)                     with pytest.raises(nx.NetworkXNotImplemented):
(12)                        nx.dag_to_branching(nx.MultiGraph())
(4)                 def test_multidigraph(self):
(8)                     with pytest.raises(nx.NetworkXNotImplemented):
(12)                        nx.dag_to_branching(nx.MultiDiGraph())
(0)             def test_ancestors_descendants_undirected():
(4)                 """Regression test to ensure ancestors and descendants work as expected on
(4)                 undirected graphs."""
(4)                 G = nx.path_graph(5)
(4)                 nx.ancestors(G, 2) == nx.descendants(G, 2) == {0, 1, 3, 4}
(0)             def test_compute_v_structures_raise():
(4)                 G = nx.Graph()
(4)                 pytest.raises(nx.NetworkXNotImplemented, nx.compute_v_structures, G)
(0)             def test_compute_v_structures():
(4)                 edges = [(0, 1), (0, 2), (3, 2)]
(4)                 G = nx.DiGraph(edges)
(4)                 v_structs = set(nx.compute_v_structures(G))
(4)                 assert len(v_structs) == 1
(4)                 assert (0, 2, 3) in v_structs
(4)                 edges = [("A", "B"), ("C", "B"), ("B", "D"), ("D", "E"), ("G", "E")]
(4)                 G = nx.DiGraph(edges)
(4)                 v_structs = set(nx.compute_v_structures(G))
(4)                 assert len(v_structs) == 2
```

File 320 - test_d_separation.py:

```
(0)  from itertools import combinations
(0)  import pytest
(0)  import networkx as nx
(0)  def path_graph():
(4)      """Return a path graph of length three."""
(4)      G = nx.path_graph(3, create_using=nx.DiGraph)
(4)      G.graph["name"] = "path"
(4)      nx.freeze(G)
(4)      return G
(0)  def fork_graph():
(4)      """Return a three node fork graph."""
(4)      G = nx.DiGraph(name="fork")
(4)      G.add_edges_from([(0, 1), (0, 2)])
(4)      nx.freeze(G)
(4)      return G
(0)  def collider_graph():
(4)      """Return a collider/v-structure graph with three nodes."""
(4)      G = nx.DiGraph(name="collider")
(4)      G.add_edges_from([(0, 2), (1, 2)])
(4)      nx.freeze(G)
(4)      return G
(0)  def naive_bayes_graph():
(4)      """Return a simply Naive Bayes PGM graph."""
(4)      G = nx.DiGraph(name="naive_bayes")
(4)      G.add_edges_from([(0, 1), (0, 2), (0, 3), (0, 4)])
(4)      nx.freeze(G)
(4)      return G
(0)  def asia_graph():
(4)      """Return the 'Asia' PGM graph."""
(4)      G = nx.DiGraph(name="asia")
(4)      G.add_edges_from(
(8)          [
(12)             ("asia", "tuberculosis"),
(12)             ("smoking", "cancer"),
(12)             ("smoking", "bronchitis"),
(12)             ("tuberculosis", "either"),
(12)             ("cancer", "either"),
(12)             ("either", "xray"),
(12)             ("either", "dyspnea"),
(12)             ("bronchitis", "dyspnea"),
(8)          ]
(4)      )
(4)      nx.freeze(G)
(4)      return G
(0)  @pytest.fixture(name="path_graph")
(0)  def path_graph_fixture():
(4)      return path_graph()
(0)  @pytest.fixture(name="fork_graph")
(0)  def fork_graph_fixture():
(4)      return fork_graph()
(0)  @pytest.fixture(name="collider_graph")
(0)  def collider_graph_fixture():
(4)      return collider_graph()
(0)  @pytest.fixture(name="naive_bayes_graph")
(0)  def naive_bayes_graph_fixture():
(4)      return naive_bayes_graph()
(0)  @pytest.fixture(name="asia_graph")
(0)  def asia_graph_fixture():
(4)      return asia_graph()
(0)  @pytest.fixture()
(0)  def large_collider_graph():
(4)      edge_list = [("A", "B"), ("C", "B"), ("B", "D"), ("D", "E"), ("B", "F"),
("G", "E")]
(4)      G = nx.DiGraph(edge_list)
(4)      return G
(0)  @pytest.fixture()
(0)  def chain_and_fork_graph():
(4)      edge_list = [("A", "B"), ("B", "C"), ("B", "D"), ("D", "C")]
(4)      G = nx.DiGraph(edge_list)
```

```
 69: (4)                 return G
 70: (0)             @pytest.fixture()
 71: (0)             def no_separating_set_graph():
 72: (4)                 edge_list = [("A", "B")]
 73: (4)                 G = nx.DiGraph(edge_list)
 74: (4)                 return G
 75: (0)             @pytest.fixture()
 76: (0)             def large_no_separating_set_graph():
 77: (4)                 edge_list = [("A", "B"), ("C", "A"), ("C", "B")]
 78: (4)                 G = nx.DiGraph(edge_list)
 79: (4)                 return G
 80: (0)             @pytest.fixture()
 81: (0)             def collider_trek_graph():
 82: (4)                 edge_list = [("A", "B"), ("C", "B"), ("C", "D")]
 83: (4)                 G = nx.DiGraph(edge_list)
 84: (4)                 return G
 85: (0)             @pytest.mark.parametrize(
 86: (4)                 "graph",
 87: (4)                 [path_graph(), fork_graph(), collider_graph(), naive_bayes_graph(),
asia_graph()],
 88: (0)             )
 89: (0)             def test_markov_condition(graph):
 90: (4)                 """Test that the Markov condition holds for each PGM graph."""
 91: (4)                 for node in graph.nodes:
 92: (8)                     parents = set(graph.predecessors(node))
 93: (8)                     non_descendants = graph.nodes - nx.descendants(graph, node) - {node} -
parents
 94: (8)                     assert nx.is_d_separator(graph, {node}, non_descendants, parents)
 95: (0)             def test_path_graph_dsep(path_graph):
 96: (4)                 """Example-based test of d-separation for path_graph."""
 97: (4)                 assert nx.is_d_separator(path_graph, {0}, {2}, {1})
 98: (4)                 assert not nx.is_d_separator(path_graph, {0}, {2}, set())
 99: (0)             def test_fork_graph_dsep(fork_graph):
100: (4)                 """Example-based test of d-separation for fork_graph."""
101: (4)                 assert nx.is_d_separator(fork_graph, {1}, {2}, {0})
102: (4)                 assert not nx.is_d_separator(fork_graph, {1}, {2}, set())
103: (0)             def test_collider_graph_dsep(collider_graph):
104: (4)                 """Example-based test of d-separation for collider_graph."""
105: (4)                 assert nx.is_d_separator(collider_graph, {0}, {1}, set())
106: (4)                 assert not nx.is_d_separator(collider_graph, {0}, {1}, {2})
107: (0)             def test_naive_bayes_dsep(naive_bayes_graph):
108: (4)                 """Example-based test of d-separation for naive_bayes_graph."""
109: (4)                 for u, v in combinations(range(1, 5), 2):
110: (8)                     assert nx.is_d_separator(naive_bayes_graph, {u}, {v}, {0})
111: (8)                     assert not nx.is_d_separator(naive_bayes_graph, {u}, {v}, set())
112: (0)             def test_asia_graph_dsep(asia_graph):
113: (4)                 """Example-based test of d-separation for asia_graph."""
114: (4)                 assert nx.is_d_separator(
115: (8)                     asia_graph, {"asia", "smoking"}, {"dyspnea", "xray"}, {"bronchitis",
"either"}
116: (4)                 )
117: (4)                 assert nx.is_d_separator(
118: (8)                     asia_graph, {"tuberculosis", "cancer"}, {"bronchitis"}, {"smoking",
"xray"}
119: (4)                 )
120: (0)             def test_undirected_graphs_are_not_supported():
121: (4)                 """
122: (4)                 Test that undirected graphs are not supported.
123: (4)                 d-separation and its related algorithms do not apply in
124: (4)                 the case of undirected graphs.
125: (4)                 """
126: (4)                 g = nx.path_graph(3, nx.Graph)
127: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
128: (8)                     nx.is_d_separator(g, {0}, {1}, {2})
129: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
130: (8)                     nx.is_minimal_d_separator(g, {0}, {1}, {2})
131: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
132: (8)                     nx.find_minimal_d_separator(g, {0}, {1})
133: (0)             def test_cyclic_graphs_raise_error():
```

```
134: (4)                """
135: (4)                Test that cycle graphs should cause erroring.
136: (4)                This is because PGMs assume a directed acyclic graph.
137: (4)                """
138: (4)                g = nx.cycle_graph(3, nx.DiGraph)
139: (4)                with pytest.raises(nx.NetworkXError):
140: (8)                    nx.is_d_separator(g, {0}, {1}, {2})
141: (4)                with pytest.raises(nx.NetworkXError):
142: (8)                    nx.find_minimal_d_separator(g, {0}, {1})
143: (4)                with pytest.raises(nx.NetworkXError):
144: (8)                    nx.is_minimal_d_separator(g, {0}, {1}, {2})
145: (0)            def test_invalid_nodes_raise_error(asia_graph):
146: (4)                """
147: (4)                Test that graphs that have invalid nodes passed in raise errors.
148: (4)                """
149: (4)                with pytest.raises(nx.NodeNotFound):
150: (8)                    nx.is_d_separator(asia_graph, {0}, {1}, {2})
151: (4)                with pytest.raises(nx.NodeNotFound):
152: (8)                    nx.is_d_separator(asia_graph, 0, 1, 2)
153: (4)                with pytest.raises(nx.NodeNotFound):
154: (8)                    nx.is_minimal_d_separator(asia_graph, {0}, {1}, {2})
155: (4)                with pytest.raises(nx.NodeNotFound):
156: (8)                    nx.is_minimal_d_separator(asia_graph, 0, 1, 2)
157: (4)                with pytest.raises(nx.NodeNotFound):
158: (8)                    nx.find_minimal_d_separator(asia_graph, {0}, {1})
159: (4)                with pytest.raises(nx.NodeNotFound):
160: (8)                    nx.find_minimal_d_separator(asia_graph, 0, 1)
161: (0)            def test_nondisjoint_node_sets_raise_error(collider_graph):
162: (4)                """
163: (4)                Test that error is raised when node sets aren't disjoint.
164: (4)                """
165: (4)                with pytest.raises(nx.NetworkXError):
166: (8)                    nx.is_d_separator(collider_graph, 0, 1, 0)
167: (4)                with pytest.raises(nx.NetworkXError):
168: (8)                    nx.is_d_separator(collider_graph, 0, 2, 0)
169: (4)                with pytest.raises(nx.NetworkXError):
170: (8)                    nx.is_d_separator(collider_graph, 0, 0, 1)
171: (4)                with pytest.raises(nx.NetworkXError):
172: (8)                    nx.is_d_separator(collider_graph, 1, 0, 0)
173: (4)                with pytest.raises(nx.NetworkXError):
174: (8)                    nx.find_minimal_d_separator(collider_graph, 0, 0)
175: (4)                with pytest.raises(nx.NetworkXError):
176: (8)                    nx.find_minimal_d_separator(collider_graph, 0, 1, included=0)
177: (4)                with pytest.raises(nx.NetworkXError):
178: (8)                    nx.find_minimal_d_separator(collider_graph, 1, 0, included=0)
179: (4)                with pytest.raises(nx.NetworkXError):
180: (8)                    nx.is_minimal_d_separator(collider_graph, 0, 0, set())
181: (4)                with pytest.raises(nx.NetworkXError):
182: (8)                    nx.is_minimal_d_separator(collider_graph, 0, 1, set(), included=0)
183: (4)                with pytest.raises(nx.NetworkXError):
184: (8)                    nx.is_minimal_d_separator(collider_graph, 1, 0, set(), included=0)
185: (0)            def test_is_minimal_d_separator(
186: (4)                large_collider_graph,
187: (4)                chain_and_fork_graph,
188: (4)                no_separating_set_graph,
189: (4)                large_no_separating_set_graph,
190: (4)                collider_trek_graph,
191: (0)            ):
192: (4)                assert not nx.is_d_separator(large_collider_graph, {"B"}, {"E"}, set())
193: (4)                Zmin = nx.find_minimal_d_separator(large_collider_graph, "B", "E")
194: (4)                assert nx.is_d_separator(large_collider_graph, "B", "E", Zmin)
195: (4)                assert nx.is_minimal_d_separator(large_collider_graph, "B", "E", Zmin)
196: (4)                assert nx.is_minimal_d_separator(large_collider_graph, {"A", "B"}, {"G",
"E"}, Zmin)
197: (4)                assert Zmin == {"D"}
198: (4)                assert not nx.is_d_separator(chain_and_fork_graph, {"A"}, {"C"}, set())
199: (4)                Zmin = nx.find_minimal_d_separator(chain_and_fork_graph, "A", "C")
200: (4)                assert nx.is_minimal_d_separator(chain_and_fork_graph, "A", "C", Zmin)
201: (4)                assert Zmin == {"B"}
```

```
202: (4)                Znotmin = Zmin.union({"D"})
203: (4)                assert not nx.is_minimal_d_separator(chain_and_fork_graph, "A", "C",
Znotmin)
204: (4)                assert not nx.is_d_separator(no_separating_set_graph, {"A"}, {"B"}, set())
205: (4)                assert nx.find_minimal_d_separator(no_separating_set_graph, "A", "B") is
None
206: (4)                assert not nx.is_d_separator(large_no_separating_set_graph, {"A"}, {"B"},
{"C"})
207: (4)                assert nx.find_minimal_d_separator(large_no_separating_set_graph, "A",
"B") is None
208: (4)                assert nx.find_minimal_d_separator(collider_trek_graph, "A", "D",
included="B") == {
209: (8)                    "B",
210: (8)                    "C",
211: (4)                }
212: (4)                assert (
213: (8)                    nx.find_minimal_d_separator(
214: (12)                       collider_trek_graph, "A", "D", included="B", restricted="B"
215: (8)                    )
216: (8)                    is None
217: (4)                )
218: (0)            def test_is_minimal_d_separator_checks_dsep():
219: (4)                """Test that is_minimal_d_separator checks for d-separation as well."""
220: (4)                g = nx.DiGraph()
221: (4)                g.add_edges_from(
222: (8)                    [
223: (12)                       ("A", "B"),
224: (12)                       ("A", "E"),
225: (12)                       ("B", "C"),
226: (12)                       ("B", "D"),
227: (12)                       ("D", "C"),
228: (12)                       ("D", "F"),
229: (12)                       ("E", "D"),
230: (12)                       ("E", "F"),
231: (8)                    ]
232: (4)                )
233: (4)                assert not nx.is_d_separator(g, {"C"}, {"F"}, {"D"})
234: (4)                assert not nx.is_minimal_d_separator(g, "C", "F", {"D"})
235: (4)                assert not nx.is_minimal_d_separator(g, "C", "F", set())
236: (0)            def test__reachable(large_collider_graph):
237: (4)                reachable = nx.algorithms.d_separation._reachable
238: (4)                g = large_collider_graph
239: (4)                x = {"F", "D"}
240: (4)                ancestors = {"A", "B", "C", "D", "F"}
241: (4)                assert reachable(g, x, ancestors, {"B"}) == {"B", "F", "D"}
242: (4)                assert reachable(g, x, ancestors, set()) == ancestors
243: (0)            def test_deprecations():
244: (4)                G = nx.DiGraph([(0, 1), (1, 2)])
245: (4)                with pytest.deprecated_call():
246: (8)                    nx.d_separated(G, 0, 2, {1})
247: (4)                with pytest.deprecated_call():
248: (8)                    z = nx.minimal_d_separator(G, 0, 2)
```

File 321 - test_distance_regular.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx import is_strongly_regular
4: (0)              @pytest.mark.parametrize(
5: (4)                  "f", (nx.is_distance_regular, nx.intersection_array,
nx.is_strongly_regular)
6: (0)              )
7: (0)              @pytest.mark.parametrize("graph_constructor", (nx.DiGraph, nx.MultiGraph))
8: (0)              def test_raises_on_directed_and_multigraphs(f, graph_constructor):
9: (4)                  G = graph_constructor([(0, 1), (1, 2)])
10: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
11: (8)                     f(G)
```

```
12: (0)             class TestDistanceRegular:
13: (4)                 def test_is_distance_regular(self):
14: (8)                     assert nx.is_distance_regular(nx.icosahedral_graph())
15: (8)                     assert nx.is_distance_regular(nx.petersen_graph())
16: (8)                     assert nx.is_distance_regular(nx.cubical_graph())
17: (8)                     assert nx.is_distance_regular(nx.complete_bipartite_graph(3, 3))
18: (8)                     assert nx.is_distance_regular(nx.tetrahedral_graph())
19: (8)                     assert nx.is_distance_regular(nx.dodecahedral_graph())
20: (8)                     assert nx.is_distance_regular(nx.pappus_graph())
21: (8)                     assert nx.is_distance_regular(nx.heawood_graph())
22: (8)                     assert nx.is_distance_regular(nx.cycle_graph(3))
23: (8)                     assert not nx.is_distance_regular(nx.path_graph(4))
24: (4)                 def test_not_connected(self):
25: (8)                     G = nx.cycle_graph(4)
26: (8)                     nx.add_cycle(G, [5, 6, 7])
27: (8)                     assert not nx.is_distance_regular(G)
28: (4)                 def test_global_parameters(self):
29: (8)                     b, c = nx.intersection_array(nx.cycle_graph(5))
30: (8)                     g = nx.global_parameters(b, c)
31: (8)                     assert list(g) == [(0, 0, 2), (1, 0, 1), (1, 1, 0)]
32: (8)                     b, c = nx.intersection_array(nx.cycle_graph(3))
33: (8)                     g = nx.global_parameters(b, c)
34: (8)                     assert list(g) == [(0, 0, 2), (1, 1, 0)]
35: (4)                 def test_intersection_array(self):
36: (8)                     b, c = nx.intersection_array(nx.cycle_graph(5))
37: (8)                     assert b == [2, 1]
38: (8)                     assert c == [1, 1]
39: (8)                     b, c = nx.intersection_array(nx.dodecahedral_graph())
40: (8)                     assert b == [3, 2, 1, 1, 1]
41: (8)                     assert c == [1, 1, 1, 2, 3]
42: (8)                     b, c = nx.intersection_array(nx.icosahedral_graph())
43: (8)                     assert b == [5, 2, 1]
44: (8)                     assert c == [1, 2, 5]
45: (0)             @pytest.mark.parametrize("f", (nx.is_distance_regular,
nx.is_strongly_regular))
46: (0)             def test_empty_graph_raises(f):
47: (4)                 G = nx.Graph()
48: (4)                 with pytest.raises(nx.NetworkXPointlessConcept, match="Graph has no
nodes"):
49: (8)                     f(G)
50: (0)             class TestStronglyRegular:
51: (4)                 """Unit tests for the :func:`~networkx.is_strongly_regular`
52: (4)                 function.
53: (4)                 """
54: (4)                 def test_cycle_graph(self):
55: (8)                     """Tests that the cycle graph on five vertices is strongly
56: (8)                     regular.
57: (8)                     """
58: (8)                     G = nx.cycle_graph(5)
59: (8)                     assert is_strongly_regular(G)
60: (4)                 def test_petersen_graph(self):
61: (8)                     """Tests that the Petersen graph is strongly regular."""
62: (8)                     G = nx.petersen_graph()
63: (8)                     assert is_strongly_regular(G)
64: (4)                 def test_path_graph(self):
65: (8)                     """Tests that the path graph is not strongly regular."""
66: (8)                     G = nx.path_graph(4)
67: (8)                     assert not is_strongly_regular(G)
```

File 322 - test_distance_measures.py:

```
1: (0)              from random import Random
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx import convert_node_labels_to_integers as cnlti
5: (0)              from networkx.algorithms.distance_measures import _extrema_bounding
6: (0)              def test__extrema_bounding_invalid_compute_kwarg():
```

```
(4)                 G = nx.path_graph(3)
(4)                 with pytest.raises(ValueError, match="compute must be one of"):
(8)                     _extrema_bounding(G, compute="spam")
(0)             class TestDistance:
(4)                 def setup_method(self):
(8)                     G = cnlti(nx.grid_2d_graph(4, 4), first_label=1, ordering="sorted")
(8)                     self.G = G
(4)                 def test_eccentricity(self):
(8)                     assert nx.eccentricity(self.G, 1) == 6
(8)                     e = nx.eccentricity(self.G)
(8)                     assert e[1] == 6
(8)                     sp = dict(nx.shortest_path_length(self.G))
(8)                     e = nx.eccentricity(self.G, sp=sp)
(8)                     assert e[1] == 6
(8)                     e = nx.eccentricity(self.G, v=1)
(8)                     assert e == 6
(8)                     e = nx.eccentricity(self.G, v=[1, 1])
(8)                     assert e[1] == 6
(8)                     e = nx.eccentricity(self.G, v=[1, 2])
(8)                     assert e[1] == 6
(8)                     G = nx.path_graph(1)
(8)                     e = nx.eccentricity(G)
(8)                     assert e[0] == 0
(8)                     e = nx.eccentricity(G, v=0)
(8)                     assert e == 0
(8)                     pytest.raises(nx.NetworkXError, nx.eccentricity, G, 1)
(8)                     G = nx.empty_graph()
(8)                     e = nx.eccentricity(G)
(8)                     assert e == {}
(4)                 def test_diameter(self):
(8)                     assert nx.diameter(self.G) == 6
(4)                 def test_radius(self):
(8)                     assert nx.radius(self.G) == 4
(4)                 def test_periphery(self):
(8)                     assert set(nx.periphery(self.G)) == {1, 4, 13, 16}
(4)                 def test_center(self):
(8)                     assert set(nx.center(self.G)) == {6, 7, 10, 11}
(4)                 def test_bound_diameter(self):
(8)                     assert nx.diameter(self.G, usebounds=True) == 6
(4)                 def test_bound_radius(self):
(8)                     assert nx.radius(self.G, usebounds=True) == 4
(4)                 def test_bound_periphery(self):
(8)                     result = {1, 4, 13, 16}
(8)                     assert set(nx.periphery(self.G, usebounds=True)) == result
(4)                 def test_bound_center(self):
(8)                     result = {6, 7, 10, 11}
(8)                     assert set(nx.center(self.G, usebounds=True)) == result
(4)                 def test_radius_exception(self):
(8)                     G = nx.Graph()
(8)                     G.add_edge(1, 2)
(8)                     G.add_edge(3, 4)
(8)                     pytest.raises(nx.NetworkXError, nx.diameter, G)
(4)                 def test_eccentricity_infinite(self):
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G = nx.Graph([(1, 2), (3, 4)])
(12)                        e = nx.eccentricity(G)
(4)                 def test_eccentricity_undirected_not_connected(self):
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G = nx.Graph([(1, 2), (3, 4)])
(12)                        e = nx.eccentricity(G, sp=1)
(4)                 def test_eccentricity_directed_weakly_connected(self):
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        DG = nx.DiGraph([(1, 2), (1, 3)])
(12)                        nx.eccentricity(DG)
(0)             class TestWeightedDistance:
(4)                 def setup_method(self):
(8)                     G = nx.Graph()
(8)                     G.add_edge(0, 1, weight=0.6, cost=0.6, high_cost=6)
(8)                     G.add_edge(0, 2, weight=0.2, cost=0.2, high_cost=2)
```

```
(8)                 G.add_edge(2, 3, weight=0.1, cost=0.1, high_cost=1)
(8)                 G.add_edge(2, 4, weight=0.7, cost=0.7, high_cost=7)
(8)                 G.add_edge(2, 5, weight=0.9, cost=0.9, high_cost=9)
(8)                 G.add_edge(1, 5, weight=0.3, cost=0.3, high_cost=3)
(8)                 self.G = G
(8)                 self.weight_fn = lambda v, u, e: 2
(4)             def test_eccentricity_weight_None(self):
(8)                 assert nx.eccentricity(self.G, 1, weight=None) == 3
(8)                 e = nx.eccentricity(self.G, weight=None)
(8)                 assert e[1] == 3
(8)                 e = nx.eccentricity(self.G, v=1, weight=None)
(8)                 assert e == 3
(8)                 e = nx.eccentricity(self.G, v=[1, 1], weight=None)
(8)                 assert e[1] == 3
(8)                 e = nx.eccentricity(self.G, v=[1, 2], weight=None)
(8)                 assert e[1] == 3
(4)             def test_eccentricity_weight_attr(self):
(8)                 assert nx.eccentricity(self.G, 1, weight="weight") == 1.5
(8)                 e = nx.eccentricity(self.G, weight="weight")
(8)                 assert (
(12)                    e
(12)                    == nx.eccentricity(self.G, weight="cost")
(12)                    != nx.eccentricity(self.G, weight="high_cost")
(8)                 )
(8)                 assert e[1] == 1.5
(8)                 e = nx.eccentricity(self.G, v=1, weight="weight")
(8)                 assert e == 1.5
(8)                 e = nx.eccentricity(self.G, v=[1, 1], weight="weight")
(8)                 assert e[1] == 1.5
(8)                 e = nx.eccentricity(self.G, v=[1, 2], weight="weight")
(8)                 assert e[1] == 1.5
(4)             def test_eccentricity_weight_fn(self):
(8)                 assert nx.eccentricity(self.G, 1, weight=self.weight_fn) == 6
(8)                 e = nx.eccentricity(self.G, weight=self.weight_fn)
(8)                 assert e[1] == 6
(8)                 e = nx.eccentricity(self.G, v=1, weight=self.weight_fn)
(8)                 assert e == 6
(8)                 e = nx.eccentricity(self.G, v=[1, 1], weight=self.weight_fn)
(8)                 assert e[1] == 6
(8)                 e = nx.eccentricity(self.G, v=[1, 2], weight=self.weight_fn)
(8)                 assert e[1] == 6
(4)             def test_diameter_weight_None(self):
(8)                 assert nx.diameter(self.G, weight=None) == 3
(4)             def test_diameter_weight_attr(self):
(8)                 assert (
(12)                    nx.diameter(self.G, weight="weight")
(12)                    == nx.diameter(self.G, weight="cost")
(12)                    == 1.6
(12)                    != nx.diameter(self.G, weight="high_cost")
(8)                 )
(4)             def test_diameter_weight_fn(self):
(8)                 assert nx.diameter(self.G, weight=self.weight_fn) == 6
(4)             def test_radius_weight_None(self):
(8)                 assert pytest.approx(nx.radius(self.G, weight=None)) == 2
(4)             def test_radius_weight_attr(self):
(8)                 assert (
(12)                    pytest.approx(nx.radius(self.G, weight="weight"))
(12)                    == pytest.approx(nx.radius(self.G, weight="cost"))
(12)                    == 0.9
(12)                    != nx.radius(self.G, weight="high_cost")
(8)                 )
(4)             def test_radius_weight_fn(self):
(8)                 assert nx.radius(self.G, weight=self.weight_fn) == 4
(4)             def test_periphery_weight_None(self):
(8)                 for v in set(nx.periphery(self.G, weight=None)):
(12)                    assert nx.eccentricity(self.G, v, weight=None) == nx.diameter(
(16)                        self.G, weight=None
(12)                    )
(4)             def test_periphery_weight_attr(self):
```

```
145: (8)                    periphery = set(nx.periphery(self.G, weight="weight"))
146: (8)                    assert (
147: (12)                       periphery
148: (12)                       == set(nx.periphery(self.G, weight="cost"))
149: (12)                       == set(nx.periphery(self.G, weight="high_cost"))
150: (8)                    )
151: (8)                    for v in periphery:
152: (12)                       assert (
153: (16)                           nx.eccentricity(self.G, v, weight="high_cost")
154: (16)                           != nx.eccentricity(self.G, v, weight="weight")
155: (16)                           == nx.eccentricity(self.G, v, weight="cost")
156: (16)                           == nx.diameter(self.G, weight="weight")
157: (16)                           == nx.diameter(self.G, weight="cost")
158: (16)                           != nx.diameter(self.G, weight="high_cost")
159: (12)                       )
160: (12)                       assert nx.eccentricity(self.G, v, weight="high_cost") ==
nx.diameter(
161: (16)                           self.G, weight="high_cost"
162: (12)                       )
163: (4)                def test_periphery_weight_fn(self):
164: (8)                    for v in set(nx.periphery(self.G, weight=self.weight_fn)):
165: (12)                       assert nx.eccentricity(self.G, v, weight=self.weight_fn) ==
nx.diameter(
166: (16)                           self.G, weight=self.weight_fn
167: (12)                       )
168: (4)                def test_center_weight_None(self):
169: (8)                    for v in set(nx.center(self.G, weight=None)):
170: (12)                       assert pytest.approx(nx.eccentricity(self.G, v, weight=None)) ==
nx.radius(
171: (16)                           self.G, weight=None
172: (12)                       )
173: (4)                def test_center_weight_attr(self):
174: (8)                    center = set(nx.center(self.G, weight="weight"))
175: (8)                    assert (
176: (12)                       center
177: (12)                       == set(nx.center(self.G, weight="cost"))
178: (12)                       != set(nx.center(self.G, weight="high_cost"))
179: (8)                    )
180: (8)                    for v in center:
181: (12)                       assert (
182: (16)                           nx.eccentricity(self.G, v, weight="high_cost")
183: (16)                           != pytest.approx(nx.eccentricity(self.G, v, weight="weight"))
184: (16)                           == pytest.approx(nx.eccentricity(self.G, v, weight="cost"))
185: (16)                           == nx.radius(self.G, weight="weight")
186: (16)                           == nx.radius(self.G, weight="cost")
187: (16)                           != nx.radius(self.G, weight="high_cost")
188: (12)                       )
189: (12)                       assert nx.eccentricity(self.G, v, weight="high_cost") ==
nx.radius(
190: (16)                           self.G, weight="high_cost"
191: (12)                       )
192: (4)                def test_center_weight_fn(self):
193: (8)                    for v in set(nx.center(self.G, weight=self.weight_fn)):
194: (12)                       assert nx.eccentricity(self.G, v, weight=self.weight_fn) ==
nx.radius(
195: (16)                           self.G, weight=self.weight_fn
196: (12)                       )
197: (4)                def test_bound_diameter_weight_None(self):
198: (8)                    assert nx.diameter(self.G, usebounds=True, weight=None) == 3
199: (4)                def test_bound_diameter_weight_attr(self):
200: (8)                    assert (
201: (12)                       nx.diameter(self.G, usebounds=True, weight="high_cost")
202: (12)                       != nx.diameter(self.G, usebounds=True, weight="weight")
203: (12)                       == nx.diameter(self.G, usebounds=True, weight="cost")
204: (12)                       == 1.6
205: (12)                       != nx.diameter(self.G, usebounds=True, weight="high_cost")
206: (8)                    )
207: (8)                    assert nx.diameter(self.G, usebounds=True, weight="high_cost") ==
nx.diameter(
```

```
208: (12)                        self.G, usebounds=True, weight="high_cost"
209: (8)                     )
210: (4)                 def test_bound_diameter_weight_fn(self):
211: (8)                     assert nx.diameter(self.G, usebounds=True, weight=self.weight_fn) == 6
212: (4)                 def test_bound_radius_weight_None(self):
213: (8)                     assert pytest.approx(nx.radius(self.G, usebounds=True, weight=None))
== 2
214: (4)                 def test_bound_radius_weight_attr(self):
215: (8)                     assert (
216: (12)                        nx.radius(self.G, usebounds=True, weight="high_cost")
217: (12)                        != pytest.approx(nx.radius(self.G, usebounds=True,
weight="weight"))
218: (12)                        == pytest.approx(nx.radius(self.G, usebounds=True, weight="cost"))
219: (12)                        == 0.9
220: (12)                        != nx.radius(self.G, usebounds=True, weight="high_cost")
221: (8)                     )
222: (8)                     assert nx.radius(self.G, usebounds=True, weight="high_cost") ==
nx.radius(
223: (12)                        self.G, usebounds=True, weight="high_cost"
224: (8)                     )
225: (4)                 def test_bound_radius_weight_fn(self):
226: (8)                     assert nx.radius(self.G, usebounds=True, weight=self.weight_fn) == 4
227: (4)                 def test_bound_periphery_weight_None(self):
228: (8)                     result = {1, 3, 4}
229: (8)                     assert set(nx.periphery(self.G, usebounds=True, weight=None)) ==
result
230: (4)                 def test_bound_periphery_weight_attr(self):
231: (8)                     result = {4, 5}
232: (8)                     assert (
233: (12)                        set(nx.periphery(self.G, usebounds=True, weight="weight"))
234: (12)                        == set(nx.periphery(self.G, usebounds=True, weight="cost"))
235: (12)                        == result
236: (8)                     )
237: (4)                 def test_bound_periphery_weight_fn(self):
238: (8)                     result = {1, 3, 4}
239: (8)                     assert (
240: (12)                        set(nx.periphery(self.G, usebounds=True, weight=self.weight_fn))
== result
241: (8)                     )
242: (4)                 def test_bound_center_weight_None(self):
243: (8)                     result = {0, 2, 5}
244: (8)                     assert set(nx.center(self.G, usebounds=True, weight=None)) == result
245: (4)                 def test_bound_center_weight_attr(self):
246: (8)                     result = {0}
247: (8)                     assert (
248: (12)                        set(nx.center(self.G, usebounds=True, weight="weight"))
249: (12)                        == set(nx.center(self.G, usebounds=True, weight="cost"))
250: (12)                        == result
251: (8)                     )
252: (4)                 def test_bound_center_weight_fn(self):
253: (8)                     result = {0, 2, 5}
254: (8)                     assert set(nx.center(self.G, usebounds=True, weight=self.weight_fn))
== result
255: (0)             class TestResistanceDistance:
256: (4)                 @classmethod
257: (4)                 def setup_class(cls):
258: (8)                     global np
259: (8)                     np = pytest.importorskip("numpy")
260: (8)                     sp = pytest.importorskip("scipy")
261: (4)                 def setup_method(self):
262: (8)                     G = nx.Graph()
263: (8)                     G.add_edge(1, 2, weight=2)
264: (8)                     G.add_edge(2, 3, weight=4)
265: (8)                     G.add_edge(3, 4, weight=1)
266: (8)                     G.add_edge(1, 4, weight=3)
267: (8)                     self.G = G
268: (4)                 def test_resistance_distance_directed_graph(self):
269: (8)                     G = nx.DiGraph()
270: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
```

```
(12)                        nx.resistance_distance(G)
(4)                 def test_resistance_distance_empty(self):
(8)                     G = nx.Graph()
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.resistance_distance(G)
(4)                 def test_resistance_distance_not_connected(self):
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        self.G.add_node(5)
(12)                        nx.resistance_distance(self.G, 1, 5)
(4)                 def test_resistance_distance_nodeA_not_in_graph(self):
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.resistance_distance(self.G, 9, 1)
(4)                 def test_resistance_distance_nodeB_not_in_graph(self):
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.resistance_distance(self.G, 1, 9)
(4)                 def test_resistance_distance(self):
(8)                     rd = nx.resistance_distance(self.G, 1, 3, "weight", True)
(8)                     test_data = 1 / (1 / (2 + 4) + 1 / (1 + 3))
(8)                     assert round(rd, 5) == round(test_data, 5)
(4)                 def test_resistance_distance_noinv(self):
(8)                     rd = nx.resistance_distance(self.G, 1, 3, "weight", False)
(8)                     test_data = 1 / (1 / (1 / 2 + 1 / 4) + 1 / (1 / 1 + 1 / 3))
(8)                     assert round(rd, 5) == round(test_data, 5)
(4)                 def test_resistance_distance_no_weight(self):
(8)                     rd = nx.resistance_distance(self.G, 1, 3)
(8)                     assert round(rd, 5) == 1
(4)                 def test_resistance_distance_neg_weight(self):
(8)                     self.G[2][3]["weight"] = -4
(8)                     rd = nx.resistance_distance(self.G, 1, 3, "weight", True)
(8)                     test_data = 1 / (1 / (2 + -4) + 1 / (1 + 3))
(8)                     assert round(rd, 5) == round(test_data, 5)
(4)                 def test_multigraph(self):
(8)                     G = nx.MultiGraph()
(8)                     G.add_edge(1, 2, weight=2)
(8)                     G.add_edge(2, 3, weight=4)
(8)                     G.add_edge(3, 4, weight=1)
(8)                     G.add_edge(1, 4, weight=3)
(8)                     rd = nx.resistance_distance(G, 1, 3, "weight", True)
(8)                     assert np.isclose(rd, 1 / (1 / (2 + 4) + 1 / (1 + 3)))
(4)                 def test_resistance_distance_div0(self):
(8)                     with pytest.raises(ZeroDivisionError):
(12)                        self.G[1][2]["weight"] = 0
(12)                        nx.resistance_distance(self.G, 1, 3, "weight")
(4)                 def test_resistance_distance_same_node(self):
(8)                     assert nx.resistance_distance(self.G, 1, 1) == 0
(4)                 def test_resistance_distance_only_nodeA(self):
(8)                     rd = nx.resistance_distance(self.G, nodeA=1)
(8)                     test_data = {}
(8)                     test_data[1] = 0
(8)                     test_data[2] = 0.75
(8)                     test_data[3] = 1
(8)                     test_data[4] = 0.75
(8)                     assert type(rd) == dict
(8)                     assert sorted(rd.keys()) == sorted(test_data.keys())
(8)                     for key in rd:
(12)                        assert np.isclose(rd[key], test_data[key])
(4)                 def test_resistance_distance_only_nodeB(self):
(8)                     rd = nx.resistance_distance(self.G, nodeB=1)
(8)                     test_data = {}
(8)                     test_data[1] = 0
(8)                     test_data[2] = 0.75
(8)                     test_data[3] = 1
(8)                     test_data[4] = 0.75
(8)                     assert type(rd) == dict
(8)                     assert sorted(rd.keys()) == sorted(test_data.keys())
(8)                     for key in rd:
(12)                        assert np.isclose(rd[key], test_data[key])
(4)                 def test_resistance_distance_all(self):
(8)                     rd = nx.resistance_distance(self.G)
```

```
(8)                 assert type(rd) == dict
(8)                 assert round(rd[1][3], 5) == 1
(0)         class TestEffectiveGraphResistance:
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 global np
(8)                 np = pytest.importorskip("numpy")
(8)                 sp = pytest.importorskip("scipy")
(4)             def setup_method(self):
(8)                 G = nx.Graph()
(8)                 G.add_edge(1, 2, weight=2)
(8)                 G.add_edge(1, 3, weight=1)
(8)                 G.add_edge(2, 3, weight=4)
(8)                 self.G = G
(4)             def test_effective_graph_resistance_directed_graph(self):
(8)                 G = nx.DiGraph()
(8)                 with pytest.raises(nx.NetworkXNotImplemented):
(12)                    nx.effective_graph_resistance(G)
(4)             def test_effective_graph_resistance_empty(self):
(8)                 G = nx.Graph()
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    nx.effective_graph_resistance(G)
(4)             def test_effective_graph_resistance_not_connected(self):
(8)                 G = nx.Graph([(1, 2), (3, 4)])
(8)                 RG = nx.effective_graph_resistance(G)
(8)                 assert np.isinf(RG)
(4)             def test_effective_graph_resistance(self):
(8)                 RG = nx.effective_graph_resistance(self.G, "weight", True)
(8)                 rd12 = 1 / (1 / (1 + 4) + 1 / 2)
(8)                 rd13 = 1 / (1 / (1 + 2) + 1 / 4)
(8)                 rd23 = 1 / (1 / (2 + 4) + 1 / 1)
(8)                 assert np.isclose(RG, rd12 + rd13 + rd23)
(4)             def test_effective_graph_resistance_noinv(self):
(8)                 RG = nx.effective_graph_resistance(self.G, "weight", False)
(8)                 rd12 = 1 / (1 / (1 / 1 + 1 / 4) + 1 / (1 / 2))
(8)                 rd13 = 1 / (1 / (1 / 1 + 1 / 2) + 1 / (1 / 4))
(8)                 rd23 = 1 / (1 / (1 / 2 + 1 / 4) + 1 / (1 / 1))
(8)                 assert np.isclose(RG, rd12 + rd13 + rd23)
(4)             def test_effective_graph_resistance_no_weight(self):
(8)                 RG = nx.effective_graph_resistance(self.G)
(8)                 assert np.isclose(RG, 2)
(4)             def test_effective_graph_resistance_neg_weight(self):
(8)                 self.G[2][3]["weight"] = -4
(8)                 RG = nx.effective_graph_resistance(self.G, "weight", True)
(8)                 rd12 = 1 / (1 / (1 + -4) + 1 / 2)
(8)                 rd13 = 1 / (1 / (1 + 2) + 1 / (-4))
(8)                 rd23 = 1 / (1 / (2 + -4) + 1 / 1)
(8)                 assert np.isclose(RG, rd12 + rd13 + rd23)
(4)             def test_effective_graph_resistance_multigraph(self):
(8)                 G = nx.MultiGraph()
(8)                 G.add_edge(1, 2, weight=2)
(8)                 G.add_edge(1, 3, weight=1)
(8)                 G.add_edge(2, 3, weight=1)
(8)                 G.add_edge(2, 3, weight=3)
(8)                 RG = nx.effective_graph_resistance(G, "weight", True)
(8)                 edge23 = 1 / (1 / 1 + 1 / 3)
(8)                 rd12 = 1 / (1 / (1 + edge23) + 1 / 2)
(8)                 rd13 = 1 / (1 / (1 + 2) + 1 / edge23)
(8)                 rd23 = 1 / (1 / (2 + edge23) + 1 / 1)
(8)                 assert np.isclose(RG, rd12 + rd13 + rd23)
(4)             def test_effective_graph_resistance_div0(self):
(8)                 with pytest.raises(ZeroDivisionError):
(12)                    self.G[1][2]["weight"] = 0
(12)                    nx.effective_graph_resistance(self.G, "weight")
(4)             def test_effective_graph_resistance_complete_graph(self):
(8)                 N = 10
(8)                 G = nx.complete_graph(N)
(8)                 RG = nx.effective_graph_resistance(G)
(8)                 assert np.isclose(RG, N - 1)
```

```
    def test_effective_graph_resistance_path_graph(self):
        N = 10
        G = nx.path_graph(N)
        RG = nx.effective_graph_resistance(G)
        assert np.isclose(RG, (N - 1) * N * (N + 1) // 6)
class TestBarycenter:
    """Test :func:`networkx.algorithms.distance_measures.barycenter`."""
    def barycenter_as_subgraph(self, g, **kwargs):
        """Return the subgraph induced on the barycenter of g"""
        b = nx.barycenter(g, **kwargs)
        assert isinstance(b, list)
        assert set(b) <= set(g)
        return g.subgraph(b)
    def test_must_be_connected(self):
        pytest.raises(nx.NetworkXNoPath, nx.barycenter, nx.empty_graph(5))
    def test_sp_kwarg(self):
        K_5 = nx.complete_graph(5)
        sp = dict(nx.shortest_path_length(K_5))
        assert nx.barycenter(K_5, sp=sp) == list(K_5)
        for u, v, data in K_5.edges.data():
            data["weight"] = 1
        pytest.raises(ValueError, nx.barycenter, K_5, sp=sp, weight="weight")
        del sp[0][1]
        pytest.raises(nx.NetworkXNoPath, nx.barycenter, K_5, sp=sp)
    def test_trees(self):
        """The barycenter of a tree is a single vertex or an edge.
        See [West01]_, p. 78.
        """
        prng = Random(0xDEADBEEF)
        for i in range(50):
            RT = nx.random_labeled_tree(prng.randint(1, 75), seed=prng)
            b = self.barycenter_as_subgraph(RT)
            if len(b) == 2:
                assert b.size() == 1
            else:
                assert len(b) == 1
                assert b.size() == 0
    def test_this_one_specific_tree(self):
        """Test the tree pictured at the bottom of [West01]_, p. 78."""
        g = nx.Graph(
            {
                "a": ["b"],
                "b": ["a", "x"],
                "x": ["b", "y"],
                "y": ["x", "z"],
                "z": ["y", 0, 1, 2, 3, 4],
                0: ["z"],
                1: ["z"],
                2: ["z"],
                3: ["z"],
                4: ["z"],
            }
        )
        b = self.barycenter_as_subgraph(g, attr="barycentricity")
        assert list(b) == ["z"]
        assert not b.edges
        expected_barycentricity = {
            0: 23,
            1: 23,
            2: 23,
            3: 23,
            4: 23,
            "a": 35,
            "b": 27,
            "x": 21,
            "y": 17,
            "z": 15,
        }
        for node, barycentricity in expected_barycentricity.items():
```

```
(12)                        assert g.nodes[node]["barycentricity"] == barycentricity
(8)                     for edge in g.edges:
(12)                        g.edges[edge]["weight"] = 2
(8)                     b = self.barycenter_as_subgraph(g, weight="weight",
attr="barycentricity2")
(8)                     assert list(b) == ["z"]
(8)                     assert not b.edges
(8)                     for node, barycentricity in expected_barycentricity.items():
(12)                        assert g.nodes[node]["barycentricity2"] == barycentricity * 2
(0)             class TestKemenyConstant:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     global np
(8)                     np = pytest.importorskip("numpy")
(8)                     sp = pytest.importorskip("scipy")
(4)                 def setup_method(self):
(8)                     G = nx.Graph()
(8)                     w12 = 2
(8)                     w13 = 3
(8)                     w23 = 4
(8)                     G.add_edge(1, 2, weight=w12)
(8)                     G.add_edge(1, 3, weight=w13)
(8)                     G.add_edge(2, 3, weight=w23)
(8)                     self.G = G
(4)                 def test_kemeny_constant_directed(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge(1, 2)
(8)                     G.add_edge(1, 3)
(8)                     G.add_edge(2, 3)
(8)                     with pytest.raises(nx.NetworkXNotImplemented):
(12)                        nx.kemeny_constant(G)
(4)                 def test_kemeny_constant_not_connected(self):
(8)                     self.G.add_node(5)
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.kemeny_constant(self.G)
(4)                 def test_kemeny_constant_no_nodes(self):
(8)                     G = nx.Graph()
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.kemeny_constant(G)
(4)                 def test_kemeny_constant_negative_weight(self):
(8)                     G = nx.Graph()
(8)                     w12 = 2
(8)                     w13 = 3
(8)                     w23 = -10
(8)                     G.add_edge(1, 2, weight=w12)
(8)                     G.add_edge(1, 3, weight=w13)
(8)                     G.add_edge(2, 3, weight=w23)
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.kemeny_constant(G, weight="weight")
(4)                 def test_kemeny_constant(self):
(8)                     K = nx.kemeny_constant(self.G, weight="weight")
(8)                     w12 = 2
(8)                     w13 = 3
(8)                     w23 = 4
(8)                     test_data = (
(12)                        3
(12)                        / 2
(12)                        * (w12 + w13)
(12)                        * (w12 + w23)
(12)                        * (w13 + w23)
(12)                        / (
(16)                            w12**2 * (w13 + w23)
(16)                            + w13**2 * (w12 + w23)
(16)                            + w23**2 * (w12 + w13)
(16)                            + 3 * w12 * w13 * w23
(12)                        )
(8)                     )
(8)                     assert np.isclose(K, test_data)
(4)                 def test_kemeny_constant_no_weight(self):
```

```
(8)                 K = nx.kemeny_constant(self.G)
(8)                 assert np.isclose(K, 4 / 3)
(4)             def test_kemeny_constant_multigraph(self):
(8)                 G = nx.MultiGraph()
(8)                 w12_1 = 2
(8)                 w12_2 = 1
(8)                 w13 = 3
(8)                 w23 = 4
(8)                 G.add_edge(1, 2, weight=w12_1)
(8)                 G.add_edge(1, 2, weight=w12_2)
(8)                 G.add_edge(1, 3, weight=w13)
(8)                 G.add_edge(2, 3, weight=w23)
(8)                 K = nx.kemeny_constant(G, weight="weight")
(8)                 w12 = w12_1 + w12_2
(8)                 test_data = (
(12)                    3
(12)                    / 2
(12)                    * (w12 + w13)
(12)                    * (w12 + w23)
(12)                    * (w13 + w23)
(12)                    / (
(16)                        w12**2 * (w13 + w23)
(16)                        + w13**2 * (w12 + w23)
(16)                        + w23**2 * (w12 + w13)
(16)                        + 3 * w12 * w13 * w23
(12)                    )
(8)                 )
(8)                 assert np.isclose(K, test_data)
(4)             def test_kemeny_constant_weight0(self):
(8)                 G = nx.Graph()
(8)                 w12 = 0
(8)                 w13 = 3
(8)                 w23 = 4
(8)                 G.add_edge(1, 2, weight=w12)
(8)                 G.add_edge(1, 3, weight=w13)
(8)                 G.add_edge(2, 3, weight=w23)
(8)                 K = nx.kemeny_constant(G, weight="weight")
(8)                 test_data = (
(12)                    3
(12)                    / 2
(12)                    * (w12 + w13)
(12)                    * (w12 + w23)
(12)                    * (w13 + w23)
(12)                    / (
(16)                        w12**2 * (w13 + w23)
(16)                        + w13**2 * (w12 + w23)
(16)                        + w23**2 * (w12 + w13)
(16)                        + 3 * w12 * w13 * w23
(12)                    )
(8)                 )
(8)                 assert np.isclose(K, test_data)
(4)             def test_kemeny_constant_selfloop(self):
(8)                 G = nx.Graph()
(8)                 w11 = 1
(8)                 w12 = 2
(8)                 w13 = 3
(8)                 w23 = 4
(8)                 G.add_edge(1, 1, weight=w11)
(8)                 G.add_edge(1, 2, weight=w12)
(8)                 G.add_edge(1, 3, weight=w13)
(8)                 G.add_edge(2, 3, weight=w23)
(8)                 K = nx.kemeny_constant(G, weight="weight")
(8)                 test_data = (
(12)                    (2 * w11 + 3 * w12 + 3 * w13)
(12)                    * (w12 + w23)
(12)                    * (w13 + w23)
(12)                    / (
(16)                        (w12 * w13 + w12 * w23 + w13 * w23)
(16)                        * (w11 + 2 * w12 + 2 * w13 + 2 * w23)
```

```
615: (12)                        )
616: (8)                    )
617: (8)                    assert np.isclose(K, test_data)
618: (4)                def test_kemeny_constant_complete_bipartite_graph(self):
619: (8)                    n1 = 5
620: (8)                    n2 = 4
621: (8)                    G = nx.complete_bipartite_graph(n1, n2)
622: (8)                    K = nx.kemeny_constant(G)
623: (8)                    assert np.isclose(K, n1 + n2 - 3 / 2)
624: (4)                def test_kemeny_constant_path_graph(self):
625: (8)                    n = 10
626: (8)                    G = nx.path_graph(n)
627: (8)                    K = nx.kemeny_constant(G)
628: (8)                    assert np.isclose(K, n**2 / 3 - 2 * n / 3 + 1 / 2)

----------------------------------------

File 323 - test_dominance.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestImmediateDominators:
4: (4)                  def test_exceptions(self):
5: (8)                      G = nx.Graph()
6: (8)                      G.add_node(0)
7: (8)                      pytest.raises(nx.NetworkXNotImplemented, nx.immediate_dominators, G,
0)
8: (8)                      G = nx.MultiGraph(G)
9: (8)                      pytest.raises(nx.NetworkXNotImplemented, nx.immediate_dominators, G,
0)
10: (8)                     G = nx.DiGraph([[0, 0]])
11: (8)                     pytest.raises(nx.NetworkXError, nx.immediate_dominators, G, 1)
12: (4)                 def test_singleton(self):
13: (8)                     G = nx.DiGraph()
14: (8)                     G.add_node(0)
15: (8)                     assert nx.immediate_dominators(G, 0) == {0: 0}
16: (8)                     G.add_edge(0, 0)
17: (8)                     assert nx.immediate_dominators(G, 0) == {0: 0}
18: (4)                 def test_path(self):
19: (8)                     n = 5
20: (8)                     G = nx.path_graph(n, create_using=nx.DiGraph())
21: (8)                     assert nx.immediate_dominators(G, 0) == {i: max(i - 1, 0) for i in
range(n)}
22: (4)                 def test_cycle(self):
23: (8)                     n = 5
24: (8)                     G = nx.cycle_graph(n, create_using=nx.DiGraph())
25: (8)                     assert nx.immediate_dominators(G, 0) == {i: max(i - 1, 0) for i in
range(n)}
26: (4)                 def test_unreachable(self):
27: (8)                     n = 5
28: (8)                     assert n > 1
29: (8)                     G = nx.path_graph(n, create_using=nx.DiGraph())
30: (8)                     assert nx.immediate_dominators(G, n // 2) == {
31: (12)                        i: max(i - 1, n // 2) for i in range(n // 2, n)
32: (8)                     }
33: (4)                 def test_irreducible1(self):
34: (8)                     edges = [(1, 2), (2, 1), (3, 2), (4, 1), (5, 3), (5, 4)]
35: (8)                     G = nx.DiGraph(edges)
36: (8)                     assert nx.immediate_dominators(G, 5) == {i: 5 for i in range(1, 6)}
37: (4)                 def test_irreducible2(self):
38: (8)                     edges = [(1, 2), (2, 1), (2, 3), (3, 2), (4, 2), (4, 3), (5, 1), (6,
4), (6, 5)]
39: (8)                     G = nx.DiGraph(edges)
40: (8)                     result = nx.immediate_dominators(G, 6)
41: (8)                     assert result == {i: 6 for i in range(1, 7)}
42: (4)                 def test_domrel_png(self):
43: (8)                     edges = [(1, 2), (2, 3), (2, 4), (2, 6), (3, 5), (4, 5), (5, 2)]
44: (8)                     G = nx.DiGraph(edges)
45: (8)                     result = nx.immediate_dominators(G, 1)
```

```
46: (8)                     assert result == {1: 1, 2: 1, 3: 2, 4: 2, 5: 2, 6: 2}
47: (8)                     result = nx.immediate_dominators(G.reverse(copy=False), 6)
48: (8)                     assert result == {1: 2, 2: 6, 3: 5, 4: 5, 5: 2, 6: 6}
49: (4)                 def test_boost_example(self):
50: (8)                     edges = [(0, 1), (1, 2), (1, 3), (2, 7), (3, 4), (4, 5), (4, 6), (5,
7), (6, 4)]
51: (8)                     G = nx.DiGraph(edges)
52: (8)                     result = nx.immediate_dominators(G, 0)
53: (8)                     assert result == {0: 0, 1: 0, 2: 1, 3: 1, 4: 3, 5: 4, 6: 4, 7: 1}
54: (8)                     result = nx.immediate_dominators(G.reverse(copy=False), 7)
55: (8)                     assert result == {0: 1, 1: 7, 2: 7, 3: 4, 4: 5, 5: 7, 6: 4, 7: 7}
56: (0)             class TestDominanceFrontiers:
57: (4)                 def test_exceptions(self):
58: (8)                     G = nx.Graph()
59: (8)                     G.add_node(0)
60: (8)                     pytest.raises(nx.NetworkXNotImplemented, nx.dominance_frontiers, G, 0)
61: (8)                     G = nx.MultiGraph(G)
62: (8)                     pytest.raises(nx.NetworkXNotImplemented, nx.dominance_frontiers, G, 0)
63: (8)                     G = nx.DiGraph([[0, 0]])
64: (8)                     pytest.raises(nx.NetworkXError, nx.dominance_frontiers, G, 1)
65: (4)                 def test_singleton(self):
66: (8)                     G = nx.DiGraph()
67: (8)                     G.add_node(0)
68: (8)                     assert nx.dominance_frontiers(G, 0) == {0: set()}
69: (8)                     G.add_edge(0, 0)
70: (8)                     assert nx.dominance_frontiers(G, 0) == {0: set()}
71: (4)                 def test_path(self):
72: (8)                     n = 5
73: (8)                     G = nx.path_graph(n, create_using=nx.DiGraph())
74: (8)                     assert nx.dominance_frontiers(G, 0) == {i: set() for i in range(n)}
75: (4)                 def test_cycle(self):
76: (8)                     n = 5
77: (8)                     G = nx.cycle_graph(n, create_using=nx.DiGraph())
78: (8)                     assert nx.dominance_frontiers(G, 0) == {i: set() for i in range(n)}
79: (4)                 def test_unreachable(self):
80: (8)                     n = 5
81: (8)                     assert n > 1
82: (8)                     G = nx.path_graph(n, create_using=nx.DiGraph())
83: (8)                     assert nx.dominance_frontiers(G, n // 2) == {i: set() for i in range(n
// 2, n)}
84: (4)                 def test_irreducible1(self):
85: (8)                     edges = [(1, 2), (2, 1), (3, 2), (4, 1), (5, 3), (5, 4)]
86: (8)                     G = nx.DiGraph(edges)
87: (8)                     assert dict(nx.dominance_frontiers(G, 5).items()) == {
88: (12)                        1: {2},
89: (12)                        2: {1},
90: (12)                        3: {2},
91: (12)                        4: {1},
92: (12)                        5: set(),
93: (8)                     }
94: (4)                 def test_irreducible2(self):
95: (8)                     edges = [(1, 2), (2, 1), (2, 3), (3, 2), (4, 2), (4, 3), (5, 1), (6,
4), (6, 5)]
96: (8)                     G = nx.DiGraph(edges)
97: (8)                     assert nx.dominance_frontiers(G, 6) == {
98: (12)                        1: {2},
99: (12)                        2: {1, 3},
100: (12)                       3: {2},
101: (12)                       4: {2, 3},
102: (12)                       5: {1},
103: (12)                       6: set(),
104: (8)                    }
105: (4)                dom def test_domrel_png(self):
106: (8)                    edges = [(1, 2), (2, 3), (2, 4), (2, 6), (3, 5), (4, 5), (5, 2)]
107: (8)                    G = nx.DiGraph(edges)
108: (8)                    assert nx.dominance_frontiers(G, 1) == {
109: (12)                       1: set(),
110: (12)                       2: {2},
111: (12)                       3: {5},
```

```
112: (12)                        4: {5},
113: (12)                        5: {2},
114: (12)                        6: set(),
115: (8)                     }
116: (8)                     result = nx.dominance_frontiers(G.reverse(copy=False), 6)
117: (8)                     assert result == {1: set(), 2: {2}, 3: {2}, 4: {2}, 5: {2}, 6: set()}
118: (4)                 def test_boost_example(self):
119: (8)                     edges = [(0, 1), (1, 2), (1, 3), (2, 7), (3, 4), (4, 5), (4, 6), (5,
7), (6, 4)]
120: (8)                     G = nx.DiGraph(edges)
121: (8)                     assert nx.dominance_frontiers(G, 0) == {
122: (12)                        0: set(),
123: (12)                        1: set(),
124: (12)                        2: {7},
125: (12)                        3: {7},
126: (12)                        4: {4, 7},
127: (12)                        5: {7},
128: (12)                        6: {4},
129: (12)                        7: set(),
130: (8)                     }
131: (8)                     result = nx.dominance_frontiers(G.reverse(copy=False), 7)
132: (8)                     expected = {
133: (12)                        0: set(),
134: (12)                        1: set(),
135: (12)                        2: {1},
136: (12)                        3: {1},
137: (12)                        4: {1, 4},
138: (12)                        5: {1},
139: (12)                        6: {4},
140: (12)                        7: set(),
141: (8)                     }
142: (8)                     assert result == expected
143: (4)                 def test_discard_issue(self):
144: (8)                     g = nx.DiGraph()
145: (8)                     g.add_edges_from(
146: (12)                        [
147: (16)                            ("b0", "b1"),
148: (16)                            ("b1", "b2"),
149: (16)                            ("b2", "b3"),
150: (16)                            ("b3", "b1"),
151: (16)                            ("b1", "b5"),
152: (16)                            ("b5", "b6"),
153: (16)                            ("b5", "b8"),
154: (16)                            ("b6", "b7"),
155: (16)                            ("b8", "b7"),
156: (16)                            ("b7", "b3"),
157: (16)                            ("b3", "b4"),
158: (12)                        ]
159: (8)                     )
160: (8)                     df = nx.dominance_frontiers(g, "b0")
161: (8)                     assert df == {
162: (12)                        "b4": set(),
163: (12)                        "b5": {"b3"},
164: (12)                        "b6": {"b7"},
165: (12)                        "b7": {"b3"},
166: (12)                        "b0": set(),
167: (12)                        "b1": {"b1"},
168: (12)                        "b2": {"b3"},
169: (12)                        "b3": {"b1"},
170: (12)                        "b8": {"b7"},
171: (8)                     }
172: (4)                 def test_loop(self):
173: (8)                     g = nx.DiGraph()
174: (8)                     g.add_edges_from([("a", "b"), ("b", "c"), ("b", "a")])
175: (8)                     df = nx.dominance_frontiers(g, "a")
176: (8)                     assert df == {"a": set(), "b": set(), "c": set()}
177: (4)                 def test_missing_immediate_doms(self):
178: (8)                     g = nx.DiGraph()
179: (8)                     edges = [
```

```
180: (12)                        ("entry_1", "b1"),
181: (12)                        ("b1", "b2"),
182: (12)                        ("b2", "b3"),
183: (12)                        ("b3", "exit"),
184: (12)                        ("entry_2", "b3"),
185: (8)                    ]
186: (8)                    g.add_edges_from(edges)
187: (8)                    nx.dominance_frontiers(g, "entry_1")
188: (4)                def test_loops_larger(self):
189: (8)                    g = nx.DiGraph()
190: (8)                    edges = [
191: (12)                        ("entry", "exit"),
192: (12)                        ("entry", "1"),
193: (12)                        ("1", "2"),
194: (12)                        ("2", "3"),
195: (12)                        ("3", "4"),
196: (12)                        ("4", "5"),
197: (12)                        ("5", "6"),
198: (12)                        ("6", "exit"),
199: (12)                        ("6", "2"),
200: (12)                        ("5", "3"),
201: (12)                        ("4", "4"),
202: (8)                    ]
203: (8)                    g.add_edges_from(edges)
204: (8)                    df = nx.dominance_frontiers(g, "entry")
205: (8)                    answer = {
206: (12)                        "entry": set(),
207: (12)                        "1": {"exit"},
208: (12)                        "2": {"exit", "2"},
209: (12)                        "3": {"exit", "3", "2"},
210: (12)                        "4": {"exit", "4", "3", "2"},
211: (12)                        "5": {"exit", "3", "2"},
212: (12)                        "6": {"exit", "2"},
213: (12)                        "exit": set(),
214: (8)                    }
215: (8)                    for n in df:
216: (12)                        assert set(df[n]) == set(answer[n])

----------------------------------------

File 324 - test_dominating.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_dominating_set():
4: (4)                  G = nx.gnp_random_graph(100, 0.1)
5: (4)                  D = nx.dominating_set(G)
6: (4)                  assert nx.is_dominating_set(G, D)
7: (4)                  D = nx.dominating_set(G, start_with=0)
8: (4)                  assert nx.is_dominating_set(G, D)
9: (0)              def test_complete():
10: (4)                 """In complete graphs each node is a dominating set.
11: (4)                 Thus the dominating set has to be of cardinality 1.
12: (4)                 """
13: (4)                 K4 = nx.complete_graph(4)
14: (4)                 assert len(nx.dominating_set(K4)) == 1
15: (4)                 K5 = nx.complete_graph(5)
16: (4)                 assert len(nx.dominating_set(K5)) == 1
17: (0)             def test_raise_dominating_set():
18: (4)                 with pytest.raises(nx.NetworkXError):
19: (8)                     G = nx.path_graph(4)
20: (8)                     D = nx.dominating_set(G, start_with=10)
21: (0)             def test_is_dominating_set():
22: (4)                 G = nx.path_graph(4)
23: (4)                 d = {1, 3}
24: (4)                 assert nx.is_dominating_set(G, d)
25: (4)                 d = {0, 2}
26: (4)                 assert nx.is_dominating_set(G, d)
27: (4)                 d = {1}
```

```
28: (4)                 assert not nx.is_dominating_set(G, d)
29: (0)             def test_wikipedia_is_dominating_set():
30: (4)                 """Example from https://en.wikipedia.org/wiki/Dominating_set"""
31: (4)                 G = nx.cycle_graph(4)
32: (4)                 G.add_edges_from([(0, 4), (1, 4), (2, 5)])
33: (4)                 assert nx.is_dominating_set(G, {4, 3, 5})
34: (4)                 assert nx.is_dominating_set(G, {0, 2})
35: (4)                 assert nx.is_dominating_set(G, {1, 2})
```

File 325 - test_efficiency.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.efficiency` module."""
2: (0)              import networkx as nx
3: (0)              class TestEfficiency:
4: (4)                  def setup_method(self):
5: (8)                      self.G1 = nx.Graph()
6: (8)                      self.G1.add_nodes_from([1, 2, 3])
7: (8)                      self.G2 = nx.cycle_graph(4)
8: (8)                      self.G3 = nx.lollipop_graph(3, 1)
9: (4)                  def test_efficiency_disconnected_nodes(self):
10: (8)                     """
11: (8)                     When nodes are disconnected, efficiency is 0
12: (8)                     """
13: (8)                     assert nx.efficiency(self.G1, 1, 2) == 0
14: (4)                 def test_local_efficiency_disconnected_graph(self):
15: (8)                     """
16: (8)                     In a disconnected graph the efficiency is 0
17: (8)                     """
18: (8)                     assert nx.local_efficiency(self.G1) == 0
19: (4)                 def test_efficiency(self):
20: (8)                     assert nx.efficiency(self.G2, 0, 1) == 1
21: (8)                     assert nx.efficiency(self.G2, 0, 2) == 1 / 2
22: (4)                 def test_global_efficiency(self):
23: (8)                     assert nx.global_efficiency(self.G2) == 5 / 6
24: (4)                 def test_global_efficiency_complete_graph(self):
25: (8)                     """
26: (8)                     Tests that the average global efficiency of the complete graph is one.
27: (8)                     """
28: (8)                     for n in range(2, 10):
29: (12)                        G = nx.complete_graph(n)
30: (12)                        assert nx.global_efficiency(G) == 1
31: (4)                 def test_local_efficiency_complete_graph(self):
32: (8)                     """
33: (8)                     Test that the local efficiency for a complete graph with at least 3
34: (8)                     nodes should be one. For a graph with only 2 nodes, the induced
35: (8)                     subgraph has no edges.
36: (8)                     """
37: (8)                     for n in range(3, 10):
38: (12)                        G = nx.complete_graph(n)
39: (12)                        assert nx.local_efficiency(G) == 1
40: (4)                 def test_using_ego_graph(self):
41: (8)                     """
42: (8)                     Test that the ego graph is used when computing local efficiency.
43: (8)                     For more information, see GitHub issue #2710.
44: (8)                     """
45: (8)                     assert nx.local_efficiency(self.G3) == 7 / 12
```

File 326 - test_euler.py:

```
1: (0)              import collections
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              @pytest.mark.parametrize("f", (nx.is_eulerian, nx.is_semieulerian))
5: (0)              def test_empty_graph_raises(f):
6: (4)                  G = nx.Graph()
```

```
 7: (4)                 with pytest.raises(nx.NetworkXPointlessConcept, match="Connectivity is
 undefined"):
 8: (8)                     f(G)
 9: (0)             class TestIsEulerian:
 10: (4)                def test_is_eulerian(self):
 11: (8)                    assert nx.is_eulerian(nx.complete_graph(5))
 12: (8)                    assert nx.is_eulerian(nx.complete_graph(7))
 13: (8)                    assert nx.is_eulerian(nx.hypercube_graph(4))
 14: (8)                    assert nx.is_eulerian(nx.hypercube_graph(6))
 15: (8)                    assert not nx.is_eulerian(nx.complete_graph(4))
 16: (8)                    assert not nx.is_eulerian(nx.complete_graph(6))
 17: (8)                    assert not nx.is_eulerian(nx.hypercube_graph(3))
 18: (8)                    assert not nx.is_eulerian(nx.hypercube_graph(5))
 19: (8)                    assert not nx.is_eulerian(nx.petersen_graph())
 20: (8)                    assert not nx.is_eulerian(nx.path_graph(4))
 21: (4)                def test_is_eulerian2(self):
 22: (8)                    G = nx.Graph()
 23: (8)                    G.add_nodes_from([1, 2, 3])
 24: (8)                    assert not nx.is_eulerian(G)
 25: (8)                    G = nx.DiGraph()
 26: (8)                    G.add_nodes_from([1, 2, 3])
 27: (8)                    assert not nx.is_eulerian(G)
 28: (8)                    G = nx.MultiDiGraph()
 29: (8)                    G.add_edge(1, 2)
 30: (8)                    G.add_edge(2, 3)
 31: (8)                    G.add_edge(2, 3)
 32: (8)                    G.add_edge(3, 1)
 33: (8)                    assert not nx.is_eulerian(G)
 34: (0)             class TestEulerianCircuit:
 35: (4)                def test_eulerian_circuit_cycle(self):
 36: (8)                    G = nx.cycle_graph(4)
 37: (8)                    edges = list(nx.eulerian_circuit(G, source=0))
 38: (8)                    nodes = [u for u, v in edges]
 39: (8)                    assert nodes == [0, 3, 2, 1]
 40: (8)                    assert edges == [(0, 3), (3, 2), (2, 1), (1, 0)]
 41: (8)                    edges = list(nx.eulerian_circuit(G, source=1))
 42: (8)                    nodes = [u for u, v in edges]
 43: (8)                    assert nodes == [1, 2, 3, 0]
 44: (8)                    assert edges == [(1, 2), (2, 3), (3, 0), (0, 1)]
 45: (8)                    G = nx.complete_graph(3)
 46: (8)                    edges = list(nx.eulerian_circuit(G, source=0))
 47: (8)                    nodes = [u for u, v in edges]
 48: (8)                    assert nodes == [0, 2, 1]
 49: (8)                    assert edges == [(0, 2), (2, 1), (1, 0)]
 50: (8)                    edges = list(nx.eulerian_circuit(G, source=1))
 51: (8)                    nodes = [u for u, v in edges]
 52: (8)                    assert nodes == [1, 2, 0]
 53: (8)                    assert edges == [(1, 2), (2, 0), (0, 1)]
 54: (4)                def test_eulerian_circuit_digraph(self):
 55: (8)                    G = nx.DiGraph()
 56: (8)                    nx.add_cycle(G, [0, 1, 2, 3])
 57: (8)                    edges = list(nx.eulerian_circuit(G, source=0))
 58: (8)                    nodes = [u for u, v in edges]
 59: (8)                    assert nodes == [0, 1, 2, 3]
 60: (8)                    assert edges == [(0, 1), (1, 2), (2, 3), (3, 0)]
 61: (8)                    edges = list(nx.eulerian_circuit(G, source=1))
 62: (8)                    nodes = [u for u, v in edges]
 63: (8)                    assert nodes == [1, 2, 3, 0]
 64: (8)                    assert edges == [(1, 2), (2, 3), (3, 0), (0, 1)]
 65: (4)                def test_multigraph(self):
 66: (8)                    G = nx.MultiGraph()
 67: (8)                    nx.add_cycle(G, [0, 1, 2, 3])
 68: (8)                    G.add_edge(1, 2)
 69: (8)                    G.add_edge(1, 2)
 70: (8)                    edges = list(nx.eulerian_circuit(G, source=0))
 71: (8)                    nodes = [u for u, v in edges]
 72: (8)                    assert nodes == [0, 3, 2, 1, 2, 1]
 73: (8)                    assert edges == [(0, 3), (3, 2), (2, 1), (1, 2), (2, 1), (1, 0)]
 74: (4)                def test_multigraph_with_keys(self):
```

```
75: (8)                     G = nx.MultiGraph()
76: (8)                     nx.add_cycle(G, [0, 1, 2, 3])
77: (8)                     G.add_edge(1, 2)
78: (8)                     G.add_edge(1, 2)
79: (8)                     edges = list(nx.eulerian_circuit(G, source=0, keys=True))
80: (8)                     nodes = [u for u, v, k in edges]
81: (8)                     assert nodes == [0, 3, 2, 1, 2, 1]
82: (8)                     assert edges[:2] == [(0, 3, 0), (3, 2, 0)]
83: (8)                     assert collections.Counter(edges[2:5]) == collections.Counter(
84: (12)                        [(2, 1, 0), (1, 2, 1), (2, 1, 2)]
85: (8)                     )
86: (8)                     assert edges[5:] == [(1, 0, 0)]
87: (4)                 def test_not_eulerian(self):
88: (8)                     with pytest.raises(nx.NetworkXError):
89: (12)                        f = list(nx.eulerian_circuit(nx.complete_graph(4)))
90: (0)             class TestIsSemiEulerian:
91: (4)                 def test_is_semieulerian(self):
92: (8)                     assert nx.is_semieulerian(nx.path_graph(4))
93: (8)                     G = nx.path_graph(6, create_using=nx.DiGraph)
94: (8)                     assert nx.is_semieulerian(G)
95: (8)                     assert not nx.is_semieulerian(nx.complete_graph(5))
96: (8)                     assert not nx.is_semieulerian(nx.complete_graph(7))
97: (8)                     assert not nx.is_semieulerian(nx.hypercube_graph(4))
98: (8)                     assert not nx.is_semieulerian(nx.hypercube_graph(6))
99: (0)             class TestHasEulerianPath:
100: (4)                def test_has_eulerian_path_cyclic(self):
101: (8)                    assert nx.has_eulerian_path(nx.complete_graph(5))
102: (8)                    assert nx.has_eulerian_path(nx.complete_graph(7))
103: (8)                    assert nx.has_eulerian_path(nx.hypercube_graph(4))
104: (8)                    assert nx.has_eulerian_path(nx.hypercube_graph(6))
105: (4)                def test_has_eulerian_path_non_cyclic(self):
106: (8)                    assert nx.has_eulerian_path(nx.path_graph(4))
107: (8)                    G = nx.path_graph(6, create_using=nx.DiGraph)
108: (8)                    assert nx.has_eulerian_path(G)
109: (4)                def test_has_eulerian_path_directed_graph(self):
110: (8)                    G = nx.DiGraph()
111: (8)                    G.add_edges_from([(0, 1), (1, 2), (0, 2)])
112: (8)                    assert not nx.has_eulerian_path(G)
113: (8)                    G = nx.DiGraph()
114: (8)                    G.add_edges_from([(0, 1), (1, 2), (2, 0)])
115: (8)                    assert nx.has_eulerian_path(G)
116: (8)                    G.add_node(3)
117: (8)                    assert not nx.has_eulerian_path(G)
118: (4)                @pytest.mark.parametrize("G", (nx.Graph(), nx.DiGraph()))
119: (4)                def test_has_eulerian_path_not_weakly_connected(self, G):
120: (8)                    G.add_edges_from([(0, 1), (2, 3), (3, 2)])
121: (8)                    assert not nx.has_eulerian_path(G)
122: (4)                @pytest.mark.parametrize("G", (nx.Graph(), nx.DiGraph()))
123: (4)                def test_has_eulerian_path_unbalancedins_more_than_one(self, G):
124: (8)                    G.add_edges_from([(0, 1), (2, 3)])
125: (8)                    assert not nx.has_eulerian_path(G)
126: (0)            class TestFindPathStart:
127: (4)                def testfind_path_start(self):
128: (8)                    find_path_start = nx.algorithms.euler._find_path_start
129: (8)                    G = nx.path_graph(6, create_using=nx.DiGraph)
130: (8)                    assert find_path_start(G) == 0
131: (8)                    edges = [(0, 1), (1, 2), (2, 0), (4, 0)]
132: (8)                    assert find_path_start(nx.DiGraph(edges)) == 4
133: (8)                    edges = [(0, 1), (1, 2), (2, 3), (2, 4)]
134: (8)                    assert find_path_start(nx.DiGraph(edges)) is None
135: (0)            class TestEulerianPath:
136: (4)                def test_eulerian_path(self):
137: (8)                    x = [(4, 0), (0, 1), (1, 2), (2, 0)]
138: (8)                    for e1, e2 in zip(x, nx.eulerian_path(nx.DiGraph(x))):
139: (12)                       assert e1 == e2
140: (4)                def test_eulerian_path_straight_link(self):
141: (8)                    G = nx.DiGraph()
142: (8)                    result = [(1, 2), (2, 3), (3, 4), (4, 5)]
143: (8)                    G.add_edges_from(result)
```

```
144: (8)                    assert result == list(nx.eulerian_path(G))
145: (8)                    assert result == list(nx.eulerian_path(G, source=1))
146: (8)                    with pytest.raises(nx.NetworkXError):
147: (12)                       list(nx.eulerian_path(G, source=3))
148: (8)                    with pytest.raises(nx.NetworkXError):
149: (12)                       list(nx.eulerian_path(G, source=4))
150: (8)                    with pytest.raises(nx.NetworkXError):
151: (12)                       list(nx.eulerian_path(G, source=5))
152: (4)                def test_eulerian_path_multigraph(self):
153: (8)                    G = nx.MultiDiGraph()
154: (8)                    result = [(2, 1), (1, 2), (2, 1), (1, 2), (2, 3), (3, 4), (4, 3)]
155: (8)                    G.add_edges_from(result)
156: (8)                    assert result == list(nx.eulerian_path(G))
157: (8)                    assert result == list(nx.eulerian_path(G, source=2))
158: (8)                    with pytest.raises(nx.NetworkXError):
159: (12)                       list(nx.eulerian_path(G, source=3))
160: (8)                    with pytest.raises(nx.NetworkXError):
161: (12)                       list(nx.eulerian_path(G, source=4))
162: (4)                def test_eulerian_path_eulerian_circuit(self):
163: (8)                    G = nx.DiGraph()
164: (8)                    result = [(1, 2), (2, 3), (3, 4), (4, 1)]
165: (8)                    result2 = [(2, 3), (3, 4), (4, 1), (1, 2)]
166: (8)                    result3 = [(3, 4), (4, 1), (1, 2), (2, 3)]
167: (8)                    G.add_edges_from(result)
168: (8)                    assert result == list(nx.eulerian_path(G))
169: (8)                    assert result == list(nx.eulerian_path(G, source=1))
170: (8)                    assert result2 == list(nx.eulerian_path(G, source=2))
171: (8)                    assert result3 == list(nx.eulerian_path(G, source=3))
172: (4)                def test_eulerian_path_undirected(self):
173: (8)                    G = nx.Graph()
174: (8)                    result = [(1, 2), (2, 3), (3, 4), (4, 5)]
175: (8)                    result2 = [(5, 4), (4, 3), (3, 2), (2, 1)]
176: (8)                    G.add_edges_from(result)
177: (8)                    assert list(nx.eulerian_path(G)) in (result, result2)
178: (8)                    assert result == list(nx.eulerian_path(G, source=1))
179: (8)                    assert result2 == list(nx.eulerian_path(G, source=5))
180: (8)                    with pytest.raises(nx.NetworkXError):
181: (12)                       list(nx.eulerian_path(G, source=3))
182: (8)                    with pytest.raises(nx.NetworkXError):
183: (12)                       list(nx.eulerian_path(G, source=2))
184: (4)                def test_eulerian_path_multigraph_undirected(self):
185: (8)                    G = nx.MultiGraph()
186: (8)                    result = [(2, 1), (1, 2), (2, 1), (1, 2), (2, 3), (3, 4)]
187: (8)                    G.add_edges_from(result)
188: (8)                    assert result == list(nx.eulerian_path(G))
189: (8)                    assert result == list(nx.eulerian_path(G, source=2))
190: (8)                    with pytest.raises(nx.NetworkXError):
191: (12)                       list(nx.eulerian_path(G, source=3))
192: (8)                    with pytest.raises(nx.NetworkXError):
193: (12)                       list(nx.eulerian_path(G, source=1))
194: (4)                @pytest.mark.parametrize(
195: (8)                    ("graph_type", "result"),
196: (8)                    (
197: (12)                       (nx.MultiGraph, [(0, 1, 0), (1, 0, 1)]),
198: (12)                       (nx.MultiDiGraph, [(0, 1, 0), (1, 0, 0)]),
199: (8)                    ),
200: (4)                )
201: (4)                def test_eulerian_with_keys(self, graph_type, result):
202: (8)                    G = graph_type([(0, 1), (1, 0)])
203: (8)                    answer = nx.eulerian_path(G, keys=True)
204: (8)                    assert list(answer) == result
205: (0)            class TestEulerize:
206: (4)                def test_disconnected(self):
207: (8)                    with pytest.raises(nx.NetworkXError):
208: (12)                       G = nx.from_edgelist([(0, 1), (2, 3)])
209: (12)                       nx.eulerize(G)
210: (4)                def test_null_graph(self):
211: (8)                    with pytest.raises(nx.NetworkXPointlessConcept):
212: (12)                       nx.eulerize(nx.Graph())
```

```
213: (4)                def test_null_multigraph(self):
214: (8)                    with pytest.raises(nx.NetworkXPointlessConcept):
215: (12)                       nx.eulerize(nx.MultiGraph())
216: (4)                def test_on_empty_graph(self):
217: (8)                    with pytest.raises(nx.NetworkXError):
218: (12)                       nx.eulerize(nx.empty_graph(3))
219: (4)                def test_on_eulerian(self):
220: (8)                    G = nx.cycle_graph(3)
221: (8)                    H = nx.eulerize(G)
222: (8)                    assert nx.is_isomorphic(G, H)
223: (4)                def test_on_eulerian_multigraph(self):
224: (8)                    G = nx.MultiGraph(nx.cycle_graph(3))
225: (8)                    G.add_edge(0, 1)
226: (8)                    H = nx.eulerize(G)
227: (8)                    assert nx.is_eulerian(H)
228: (4)                def test_on_complete_graph(self):
229: (8)                    G = nx.complete_graph(4)
230: (8)                    assert nx.is_eulerian(nx.eulerize(G))
231: (8)                    assert nx.is_eulerian(nx.eulerize(nx.MultiGraph(G)))
232: (4)                def test_on_non_eulerian_graph(self):
233: (8)                    G = nx.cycle_graph(18)
234: (8)                    G.add_edge(0, 18)
235: (8)                    G.add_edge(18, 19)
236: (8)                    G.add_edge(17, 19)
237: (8)                    G.add_edge(4, 20)
238: (8)                    G.add_edge(20, 21)
239: (8)                    G.add_edge(21, 22)
240: (8)                    G.add_edge(22, 23)
241: (8)                    G.add_edge(23, 24)
242: (8)                    G.add_edge(24, 25)
243: (8)                    G.add_edge(25, 26)
244: (8)                    G.add_edge(26, 27)
245: (8)                    G.add_edge(27, 28)
246: (8)                    G.add_edge(28, 13)
247: (8)                    assert not nx.is_eulerian(G)
248: (8)                    G = nx.eulerize(G)
249: (8)                    assert nx.is_eulerian(G)
250: (8)                    assert nx.number_of_edges(G) == 39

----------------------------------------

File 327 - test_graph_hashing.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.generators import directed
4: (0)              def test_empty_graph_hash():
5: (4)                  """
6: (4)                  empty graphs should give hashes regardless of other params
7: (4)                  """
8: (4)                  G1 = nx.empty_graph()
9: (4)                  G2 = nx.empty_graph()
10: (4)                 h1 = nx.weisfeiler_lehman_graph_hash(G1)
11: (4)                 h2 = nx.weisfeiler_lehman_graph_hash(G2)
12: (4)                 h3 = nx.weisfeiler_lehman_graph_hash(G2, edge_attr="edge_attr1")
13: (4)                 h4 = nx.weisfeiler_lehman_graph_hash(G2, node_attr="node_attr1")
14: (4)                 h5 = nx.weisfeiler_lehman_graph_hash(
15: (8)                     G2, edge_attr="edge_attr1", node_attr="node_attr1"
16: (4)                 )
17: (4)                 h6 = nx.weisfeiler_lehman_graph_hash(G2, iterations=10)
18: (4)                 assert h1 == h2
19: (4)                 assert h1 == h3
20: (4)                 assert h1 == h4
21: (4)                 assert h1 == h5
22: (4)                 assert h1 == h6
23: (0)             def test_directed():
24: (4)                 """
25: (4)                 A directed graph with no bi-directional edges should yield different a
graph hash
```

```
26: (4)                 to the same graph taken as undirected if there are no hash collisions.
27: (4)                 """
28: (4)                 r = 10
29: (4)                 for i in range(r):
30: (8)                     G_directed = nx.gn_graph(10 + r, seed=100 + i)
31: (8)                     G_undirected = nx.to_undirected(G_directed)
32: (8)                     h_directed = nx.weisfeiler_lehman_graph_hash(G_directed)
33: (8)                     h_undirected = nx.weisfeiler_lehman_graph_hash(G_undirected)
34: (8)                     assert h_directed != h_undirected
35: (0)             def test_reversed():
36: (4)                 """
37: (4)                 A directed graph with no bi-directional edges should yield different a
graph hash
38: (4)                 to the same graph taken with edge directions reversed if there are no hash
collisions.
39: (4)                 Here we test a cycle graph which is the minimal counterexample
40: (4)                 """
41: (4)                 G = nx.cycle_graph(5, create_using=nx.DiGraph)
42: (4)                 nx.set_node_attributes(G, {n: str(n) for n in G.nodes()}, name="label")
43: (4)                 G_reversed = G.reverse()
44: (4)                 h = nx.weisfeiler_lehman_graph_hash(G, node_attr="label")
45: (4)                 h_reversed = nx.weisfeiler_lehman_graph_hash(G_reversed,
node_attr="label")
46: (4)                 assert h != h_reversed
47: (0)             def test_isomorphic():
48: (4)                 """
49: (4)                 graph hashes should be invariant to node-relabeling (when the output is
reindexed
50: (4)                 by the same mapping)
51: (4)                 """
52: (4)                 n, r = 100, 10
53: (4)                 p = 1.0 / r
54: (4)                 for i in range(1, r + 1):
55: (8)                     G1 = nx.erdos_renyi_graph(n, p * i, seed=200 + i)
56: (8)                     G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
57: (8)                     g1_hash = nx.weisfeiler_lehman_graph_hash(G1)
58: (8)                     g2_hash = nx.weisfeiler_lehman_graph_hash(G2)
59: (8)                     assert g1_hash == g2_hash
60: (0)             def test_isomorphic_edge_attr():
61: (4)                 """
62: (4)                 Isomorphic graphs with differing edge attributes should yield different
graph
63: (4)                 hashes if the 'edge_attr' argument is supplied and populated in the graph,
64: (4)                 and there are no hash collisions.
65: (4)                 The output should still be invariant to node-relabeling
66: (4)                 """
67: (4)                 n, r = 100, 10
68: (4)                 p = 1.0 / r
69: (4)                 for i in range(1, r + 1):
70: (8)                     G1 = nx.erdos_renyi_graph(n, p * i, seed=300 + i)
71: (8)                     for a, b in G1.edges:
72: (12)                        G1[a][b]["edge_attr1"] = f"{a}-{b}-1"
73: (12)                        G1[a][b]["edge_attr2"] = f"{a}-{b}-2"
74: (8)                     g1_hash_with_edge_attr1 = nx.weisfeiler_lehman_graph_hash(
75: (12)                        G1, edge_attr="edge_attr1"
76: (8)                     )
77: (8)                     g1_hash_with_edge_attr2 = nx.weisfeiler_lehman_graph_hash(
78: (12)                        G1, edge_attr="edge_attr2"
79: (8)                     )
80: (8)                     g1_hash_no_edge_attr = nx.weisfeiler_lehman_graph_hash(G1,
edge_attr=None)
81: (8)                     assert g1_hash_with_edge_attr1 != g1_hash_no_edge_attr
82: (8)                     assert g1_hash_with_edge_attr2 != g1_hash_no_edge_attr
83: (8)                     assert g1_hash_with_edge_attr1 != g1_hash_with_edge_attr2
84: (8)                     G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
85: (8)                     g2_hash_with_edge_attr1 = nx.weisfeiler_lehman_graph_hash(
86: (12)                        G2, edge_attr="edge_attr1"
87: (8)                     )
88: (8)                     g2_hash_with_edge_attr2 = nx.weisfeiler_lehman_graph_hash(
```

```
89: (12)                        G2, edge_attr="edge_attr2"
90: (8)                     )
91: (8)                     assert g1_hash_with_edge_attr1 == g2_hash_with_edge_attr1
92: (8)                     assert g1_hash_with_edge_attr2 == g2_hash_with_edge_attr2
93: (0)             def test_missing_edge_attr():
94: (4)                 """
95: (4)                 If the 'edge_attr' argument is supplied but is missing from an edge in the
graph,
96: (4)                 we should raise a KeyError
97: (4)                 """
98: (4)                 G = nx.Graph()
99: (4)                 G.add_edges_from([(1, 2, {"edge_attr1": "a"}), (1, 3, {})])
100: (4)                pytest.raises(KeyError, nx.weisfeiler_lehman_graph_hash, G,
edge_attr="edge_attr1")
101: (0)            def test_isomorphic_node_attr():
102: (4)                """
103: (4)                Isomorphic graphs with differing node attributes should yield different
graph
104: (4)                hashes if the 'node_attr' argument is supplied and populated in the graph,
and
105: (4)                there are no hash collisions.
106: (4)                The output should still be invariant to node-relabeling
107: (4)                """
108: (4)                n, r = 100, 10
109: (4)                p = 1.0 / r
110: (4)                for i in range(1, r + 1):
111: (8)                    G1 = nx.erdos_renyi_graph(n, p * i, seed=400 + i)
112: (8)                    for u in G1.nodes():
113: (12)                       G1.nodes[u]["node_attr1"] = f"{u}-1"
114: (12)                       G1.nodes[u]["node_attr2"] = f"{u}-2"
115: (8)                    g1_hash_with_node_attr1 = nx.weisfeiler_lehman_graph_hash(
116: (12)                       G1, node_attr="node_attr1"
117: (8)                    )
118: (8)                    g1_hash_with_node_attr2 = nx.weisfeiler_lehman_graph_hash(
119: (12)                       G1, node_attr="node_attr2"
120: (8)                    )
121: (8)                    g1_hash_no_node_attr = nx.weisfeiler_lehman_graph_hash(G1,
node_attr=None)
122: (8)                    assert g1_hash_with_node_attr1 != g1_hash_no_node_attr
123: (8)                    assert g1_hash_with_node_attr2 != g1_hash_no_node_attr
124: (8)                    assert g1_hash_with_node_attr1 != g1_hash_with_node_attr2
125: (8)                    G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
126: (8)                    g2_hash_with_node_attr1 = nx.weisfeiler_lehman_graph_hash(
127: (12)                       G2, node_attr="node_attr1"
128: (8)                    )
129: (8)                    g2_hash_with_node_attr2 = nx.weisfeiler_lehman_graph_hash(
130: (12)                       G2, node_attr="node_attr2"
131: (8)                    )
132: (8)                    assert g1_hash_with_node_attr1 == g2_hash_with_node_attr1
133: (8)                    assert g1_hash_with_node_attr2 == g2_hash_with_node_attr2
134: (0)            def test_missing_node_attr():
135: (4)                """
136: (4)                If the 'node_attr' argument is supplied but is missing from a node in the
graph,
137: (4)                we should raise a KeyError
138: (4)                """
139: (4)                G = nx.Graph()
140: (4)                G.add_nodes_from([(1, {"node_attr1": "a"}), (2, {})])
141: (4)                G.add_edges_from([(1, 2), (2, 3), (3, 1), (1, 4)])
142: (4)                pytest.raises(KeyError, nx.weisfeiler_lehman_graph_hash, G,
node_attr="node_attr1")
143: (0)            def test_isomorphic_edge_attr_and_node_attr():
144: (4)                """
145: (4)                Isomorphic graphs with differing node attributes should yield different
graph
146: (4)                hashes if the 'node_attr' and 'edge_attr' argument is supplied and
populated in
147: (4)                the graph, and there are no hash collisions.
148: (4)                The output should still be invariant to node-relabeling
```

```
149: (4)                """
150: (4)                n, r = 100, 10
151: (4)                p = 1.0 / r
152: (4)                for i in range(1, r + 1):
153: (8)                    G1 = nx.erdos_renyi_graph(n, p * i, seed=500 + i)
154: (8)                    for u in G1.nodes():
155: (12)                       G1.nodes[u]["node_attr1"] = f"{u}-1"
156: (12)                       G1.nodes[u]["node_attr2"] = f"{u}-2"
157: (8)                    for a, b in G1.edges:
158: (12)                       G1[a][b]["edge_attr1"] = f"{a}-{b}-1"
159: (12)                       G1[a][b]["edge_attr2"] = f"{a}-{b}-2"
160: (8)                    g1_hash_edge1_node1 = nx.weisfeiler_lehman_graph_hash(
161: (12)                       G1, edge_attr="edge_attr1", node_attr="node_attr1"
162: (8)                    )
163: (8)                    g1_hash_edge2_node2 = nx.weisfeiler_lehman_graph_hash(
164: (12)                       G1, edge_attr="edge_attr2", node_attr="node_attr2"
165: (8)                    )
166: (8)                    g1_hash_edge1_node2 = nx.weisfeiler_lehman_graph_hash(
167: (12)                       G1, edge_attr="edge_attr1", node_attr="node_attr2"
168: (8)                    )
169: (8)                    g1_hash_no_attr = nx.weisfeiler_lehman_graph_hash(G1)
170: (8)                    assert g1_hash_edge1_node1 != g1_hash_no_attr
171: (8)                    assert g1_hash_edge2_node2 != g1_hash_no_attr
172: (8)                    assert g1_hash_edge1_node1 != g1_hash_edge2_node2
173: (8)                    assert g1_hash_edge1_node2 != g1_hash_edge2_node2
174: (8)                    assert g1_hash_edge1_node2 != g1_hash_edge1_node1
175: (8)                    G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
176: (8)                    g2_hash_edge1_node1 = nx.weisfeiler_lehman_graph_hash(
177: (12)                       G2, edge_attr="edge_attr1", node_attr="node_attr1"
178: (8)                    )
179: (8)                    g2_hash_edge2_node2 = nx.weisfeiler_lehman_graph_hash(
180: (12)                       G2, edge_attr="edge_attr2", node_attr="node_attr2"
181: (8)                    )
182: (8)                    assert g1_hash_edge1_node1 == g2_hash_edge1_node1
183: (8)                    assert g1_hash_edge2_node2 == g2_hash_edge2_node2
184: (0)            def test_digest_size():
185: (4)                """
186: (4)                The hash string lengths should be as expected for a variety of graphs and
187: (4)                digest sizes
188: (4)                """
189: (4)                n, r = 100, 10
190: (4)                p = 1.0 / r
191: (4)                for i in range(1, r + 1):
192: (8)                    G = nx.erdos_renyi_graph(n, p * i, seed=1000 + i)
193: (8)                    h16 = nx.weisfeiler_lehman_graph_hash(G)
194: (8)                    h32 = nx.weisfeiler_lehman_graph_hash(G, digest_size=32)
195: (8)                    assert h16 != h32
196: (8)                    assert len(h16) == 16 * 2
197: (8)                    assert len(h32) == 32 * 2
198: (0)            def is_subiteration(a, b):
199: (4)                """
200: (4)                returns True if that each hash sequence in 'a' is a prefix for
201: (4)                the corresponding sequence indexed by the same node in 'b'.
202: (4)                """
203: (4)                return all(b[node][: len(hashes)] == hashes for node, hashes in a.items())
204: (0)            def hexdigest_sizes_correct(a, digest_size):
205: (4)                """
206: (4)                returns True if all hex digest sizes are the expected length in a
node:subgraph-hashes
207: (4)                dictionary. Hex digest string length == 2 * bytes digest length since each
pair of hex
208: (4)                digits encodes 1 byte (https://docs.python.org/3/library/hashlib.html)
209: (4)                """
210: (4)                hexdigest_size = digest_size * 2
211: (4)                list_digest_sizes_correct = lambda l: all(len(x) == hexdigest_size for x
in l)
212: (4)                return all(list_digest_sizes_correct(hashes) for hashes in a.values())
213: (0)            def test_empty_graph_subgraph_hash():
214: (4)                """ "
```

```
215: (4)                empty graphs should give empty dict subgraph hashes regardless of other
params
216: (4)                """
217: (4)                G = nx.empty_graph()
218: (4)                subgraph_hashes1 = nx.weisfeiler_lehman_subgraph_hashes(G)
219: (4)                subgraph_hashes2 = nx.weisfeiler_lehman_subgraph_hashes(G,
edge_attr="edge_attr")
220: (4)                subgraph_hashes3 = nx.weisfeiler_lehman_subgraph_hashes(G,
node_attr="edge_attr")
221: (4)                subgraph_hashes4 = nx.weisfeiler_lehman_subgraph_hashes(G, iterations=2)
222: (4)                subgraph_hashes5 = nx.weisfeiler_lehman_subgraph_hashes(G, digest_size=64)
223: (4)                assert subgraph_hashes1 == {}
224: (4)                assert subgraph_hashes2 == {}
225: (4)                assert subgraph_hashes3 == {}
226: (4)                assert subgraph_hashes4 == {}
227: (4)                assert subgraph_hashes5 == {}
228: (0)            def test_directed_subgraph_hash():
229: (4)                """
230: (4)                A directed graph with no bi-directional edges should yield different
subgraph hashes
231: (4)                to the same graph taken as undirected, if all hashes don't collide.
232: (4)                """
233: (4)                r = 10
234: (4)                for i in range(r):
235: (8)                    G_directed = nx.gn_graph(10 + r, seed=100 + i)
236: (8)                    G_undirected = nx.to_undirected(G_directed)
237: (8)                    directed_subgraph_hashes =
nx.weisfeiler_lehman_subgraph_hashes(G_directed)
238: (8)                    undirected_subgraph_hashes =
nx.weisfeiler_lehman_subgraph_hashes(G_undirected)
239: (8)                    assert directed_subgraph_hashes != undirected_subgraph_hashes
240: (0)            def test_reversed_subgraph_hash():
241: (4)                """
242: (4)                A directed graph with no bi-directional edges should yield different
subgraph hashes
243: (4)                to the same graph taken with edge directions reversed if there are no hash
collisions.
244: (4)                Here we test a cycle graph which is the minimal counterexample
245: (4)                """
246: (4)                G = nx.cycle_graph(5, create_using=nx.DiGraph)
247: (4)                nx.set_node_attributes(G, {n: str(n) for n in G.nodes()}, name="label")
248: (4)                G_reversed = G.reverse()
249: (4)                h = nx.weisfeiler_lehman_subgraph_hashes(G, node_attr="label")
250: (4)                h_reversed = nx.weisfeiler_lehman_subgraph_hashes(G_reversed,
node_attr="label")
251: (4)                assert h != h_reversed
252: (0)            def test_isomorphic_subgraph_hash():
253: (4)                """
254: (4)                the subgraph hashes should be invariant to node-relabeling when the output
is reindexed
255: (4)                by the same mapping and all hashes don't collide.
256: (4)                """
257: (4)                n, r = 100, 10
258: (4)                p = 1.0 / r
259: (4)                for i in range(1, r + 1):
260: (8)                    G1 = nx.erdos_renyi_graph(n, p * i, seed=200 + i)
261: (8)                    G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
262: (8)                    g1_subgraph_hashes = nx.weisfeiler_lehman_subgraph_hashes(G1)
263: (8)                    g2_subgraph_hashes = nx.weisfeiler_lehman_subgraph_hashes(G2)
264: (8)                    assert g1_subgraph_hashes == {-1 * k: v for k, v in
g2_subgraph_hashes.items()}
265: (0)            def test_isomorphic_edge_attr_subgraph_hash():
266: (4)                """
267: (4)                Isomorphic graphs with differing edge attributes should yield different
subgraph
268: (4)                hashes if the 'edge_attr' argument is supplied and populated in the graph,
and
269: (4)                all hashes don't collide.
270: (4)                The output should still be invariant to node-relabeling
```

```
271: (4)                """
272: (4)                n, r = 100, 10
273: (4)                p = 1.0 / r
274: (4)                for i in range(1, r + 1):
275: (8)                    G1 = nx.erdos_renyi_graph(n, p * i, seed=300 + i)
276: (8)                    for a, b in G1.edges:
277: (12)                       G1[a][b]["edge_attr1"] = f"{a}-{b}-1"
278: (12)                       G1[a][b]["edge_attr2"] = f"{a}-{b}-2"
279: (8)                    g1_hash_with_edge_attr1 = nx.weisfeiler_lehman_subgraph_hashes(
280: (12)                       G1, edge_attr="edge_attr1"
281: (8)                    )
282: (8)                    g1_hash_with_edge_attr2 = nx.weisfeiler_lehman_subgraph_hashes(
283: (12)                       G1, edge_attr="edge_attr2"
284: (8)                    )
285: (8)                    g1_hash_no_edge_attr = nx.weisfeiler_lehman_subgraph_hashes(G1,
edge_attr=None)
286: (8)                    assert g1_hash_with_edge_attr1 != g1_hash_no_edge_attr
287: (8)                    assert g1_hash_with_edge_attr2 != g1_hash_no_edge_attr
288: (8)                    assert g1_hash_with_edge_attr1 != g1_hash_with_edge_attr2
289: (8)                    G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
290: (8)                    g2_hash_with_edge_attr1 = nx.weisfeiler_lehman_subgraph_hashes(
291: (12)                       G2, edge_attr="edge_attr1"
292: (8)                    )
293: (8)                    g2_hash_with_edge_attr2 = nx.weisfeiler_lehman_subgraph_hashes(
294: (12)                       G2, edge_attr="edge_attr2"
295: (8)                    )
296: (8)                    assert g1_hash_with_edge_attr1 == {
297: (12)                       -1 * k: v for k, v in g2_hash_with_edge_attr1.items()
298: (8)                    }
299: (8)                    assert g1_hash_with_edge_attr2 == {
300: (12)                       -1 * k: v for k, v in g2_hash_with_edge_attr2.items()
301: (8)                    }
302: (0)            def test_missing_edge_attr_subgraph_hash():
303: (4)                """
304: (4)                If the 'edge_attr' argument is supplied but is missing from an edge in the
graph,
305: (4)                we should raise a KeyError
306: (4)                """
307: (4)                G = nx.Graph()
308: (4)                G.add_edges_from([(1, 2, {"edge_attr1": "a"}), (1, 3, {})])
309: (4)                pytest.raises(
310: (8)                    KeyError, nx.weisfeiler_lehman_subgraph_hashes, G,
edge_attr="edge_attr1"
311: (4)                )
312: (0)            def test_isomorphic_node_attr_subgraph_hash():
313: (4)                """
314: (4)                Isomorphic graphs with differing node attributes should yield different
subgraph
315: (4)                hashes if the 'node_attr' argument is supplied and populated in the graph,
and
316: (4)                all hashes don't collide.
317: (4)                The output should still be invariant to node-relabeling
318: (4)                """
319: (4)                n, r = 100, 10
320: (4)                p = 1.0 / r
321: (4)                for i in range(1, r + 1):
322: (8)                    G1 = nx.erdos_renyi_graph(n, p * i, seed=400 + i)
323: (8)                    for u in G1.nodes():
324: (12)                       G1.nodes[u]["node_attr1"] = f"{u}-1"
325: (12)                       G1.nodes[u]["node_attr2"] = f"{u}-2"
326: (8)                    g1_hash_with_node_attr1 = nx.weisfeiler_lehman_subgraph_hashes(
327: (12)                       G1, node_attr="node_attr1"
328: (8)                    )
329: (8)                    g1_hash_with_node_attr2 = nx.weisfeiler_lehman_subgraph_hashes(
330: (12)                       G1, node_attr="node_attr2"
331: (8)                    )
332: (8)                    g1_hash_no_node_attr = nx.weisfeiler_lehman_subgraph_hashes(G1,
node_attr=None)
333: (8)                    assert g1_hash_with_node_attr1 != g1_hash_no_node_attr
```

```
334: (8)                    assert g1_hash_with_node_attr2 != g1_hash_no_node_attr
335: (8)                    assert g1_hash_with_node_attr1 != g1_hash_with_node_attr2
336: (8)                    G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
337: (8)                    g2_hash_with_node_attr1 = nx.weisfeiler_lehman_subgraph_hashes(
338: (12)                       G2, node_attr="node_attr1"
339: (8)                    )
340: (8)                    g2_hash_with_node_attr2 = nx.weisfeiler_lehman_subgraph_hashes(
341: (12)                       G2, node_attr="node_attr2"
342: (8)                    )
343: (8)                    assert g1_hash_with_node_attr1 == {
344: (12)                       -1 * k: v for k, v in g2_hash_with_node_attr1.items()
345: (8)                    }
346: (8)                    assert g1_hash_with_node_attr2 == {
347: (12)                       -1 * k: v for k, v in g2_hash_with_node_attr2.items()
348: (8)                    }
349: (0)            def test_missing_node_attr_subgraph_hash():
350: (4)                """
351: (4)                If the 'node_attr' argument is supplied but is missing from a node in the
graph,
352: (4)                we should raise a KeyError
353: (4)                """
354: (4)                G = nx.Graph()
355: (4)                G.add_nodes_from([(1, {"node_attr1": "a"}), (2, {})])
356: (4)                G.add_edges_from([(1, 2), (2, 3), (3, 1), (1, 4)])
357: (4)                pytest.raises(
358: (8)                    KeyError, nx.weisfeiler_lehman_subgraph_hashes, G,
node_attr="node_attr1"
359: (4)                )
360: (0)            def test_isomorphic_edge_attr_and_node_attr_subgraph_hash():
361: (4)                """
362: (4)                Isomorphic graphs with differing node attributes should yield different
subgraph
363: (4)                hashes if the 'node_attr' and 'edge_attr' argument is supplied and
populated in
364: (4)                the graph, and all hashes don't collide
365: (4)                The output should still be invariant to node-relabeling
366: (4)                """
367: (4)                n, r = 100, 10
368: (4)                p = 1.0 / r
369: (4)                for i in range(1, r + 1):
370: (8)                    G1 = nx.erdos_renyi_graph(n, p * i, seed=500 + i)
371: (8)                    for u in G1.nodes():
372: (12)                       G1.nodes[u]["node_attr1"] = f"{u}-1"
373: (12)                       G1.nodes[u]["node_attr2"] = f"{u}-2"
374: (8)                    for a, b in G1.edges:
375: (12)                       G1[a][b]["edge_attr1"] = f"{a}-{b}-1"
376: (12)                       G1[a][b]["edge_attr2"] = f"{a}-{b}-2"
377: (8)                    g1_hash_edge1_node1 = nx.weisfeiler_lehman_subgraph_hashes(
378: (12)                       G1, edge_attr="edge_attr1", node_attr="node_attr1"
379: (8)                    )
380: (8)                    g1_hash_edge2_node2 = nx.weisfeiler_lehman_subgraph_hashes(
381: (12)                       G1, edge_attr="edge_attr2", node_attr="node_attr2"
382: (8)                    )
383: (8)                    g1_hash_edge1_node2 = nx.weisfeiler_lehman_subgraph_hashes(
384: (12)                       G1, edge_attr="edge_attr1", node_attr="node_attr2"
385: (8)                    )
386: (8)                    g1_hash_no_attr = nx.weisfeiler_lehman_subgraph_hashes(G1)
387: (8)                    assert g1_hash_edge1_node1 != g1_hash_no_attr
388: (8)                    assert g1_hash_edge2_node2 != g1_hash_no_attr
389: (8)                    assert g1_hash_edge1_node1 != g1_hash_edge2_node2
390: (8)                    assert g1_hash_edge1_node2 != g1_hash_edge2_node2
391: (8)                    assert g1_hash_edge1_node2 != g1_hash_edge1_node1
392: (8)                    G2 = nx.relabel_nodes(G1, {u: -1 * u for u in G1.nodes()})
393: (8)                    g2_hash_edge1_node1 = nx.weisfeiler_lehman_subgraph_hashes(
394: (12)                       G2, edge_attr="edge_attr1", node_attr="node_attr1"
395: (8)                    )
396: (8)                    g2_hash_edge2_node2 = nx.weisfeiler_lehman_subgraph_hashes(
397: (12)                       G2, edge_attr="edge_attr2", node_attr="node_attr2"
398: (8)                    )
```

```
399: (8)                    assert g1_hash_edge1_node1 == {
400: (12)                       -1 * k: v for k, v in g2_hash_edge1_node1.items()
401: (8)                    }
402: (8)                    assert g1_hash_edge2_node2 == {
403: (12)                       -1 * k: v for k, v in g2_hash_edge2_node2.items()
404: (8)                    }
405: (0)            def test_iteration_depth():
406: (4)                """
407: (4)                All nodes should have the correct number of subgraph hashes in the output
when
408: (4)                using degree as initial node labels
409: (4)                Subsequent iteration depths for the same graph should be additive for each
node
410: (4)                """
411: (4)                n, r = 100, 10
412: (4)                p = 1.0 / r
413: (4)                for i in range(1, r + 1):
414: (8)                    G = nx.erdos_renyi_graph(n, p * i, seed=600 + i)
415: (8)                    depth3 = nx.weisfeiler_lehman_subgraph_hashes(G, iterations=3)
416: (8)                    depth4 = nx.weisfeiler_lehman_subgraph_hashes(G, iterations=4)
417: (8)                    depth5 = nx.weisfeiler_lehman_subgraph_hashes(G, iterations=5)
418: (8)                    assert all(len(hashes) == 3 for hashes in depth3.values())
419: (8)                    assert all(len(hashes) == 4 for hashes in depth4.values())
420: (8)                    assert all(len(hashes) == 5 for hashes in depth5.values())
421: (8)                    assert is_subiteration(depth3, depth4)
422: (8)                    assert is_subiteration(depth4, depth5)
423: (8)                    assert is_subiteration(depth3, depth5)
424: (0)            def test_iteration_depth_edge_attr():
425: (4)                """
426: (4)                All nodes should have the correct number of subgraph hashes in the output
when
427: (4)                setting initial node labels empty and using an edge attribute when
aggregating
428: (4)                neighborhoods.
429: (4)                Subsequent iteration depths for the same graph should be additive for each
node
430: (4)                """
431: (4)                n, r = 100, 10
432: (4)                p = 1.0 / r
433: (4)                for i in range(1, r + 1):
434: (8)                    G = nx.erdos_renyi_graph(n, p * i, seed=700 + i)
435: (8)                    for a, b in G.edges:
436: (12)                       G[a][b]["edge_attr1"] = f"{a}-{b}-1"
437: (8)                    depth3 = nx.weisfeiler_lehman_subgraph_hashes(
438: (12)                       G, edge_attr="edge_attr1", iterations=3
439: (8)                    )
440: (8)                    depth4 = nx.weisfeiler_lehman_subgraph_hashes(
441: (12)                       G, edge_attr="edge_attr1", iterations=4
442: (8)                    )
443: (8)                    depth5 = nx.weisfeiler_lehman_subgraph_hashes(
444: (12)                       G, edge_attr="edge_attr1", iterations=5
445: (8)                    )
446: (8)                    assert all(len(hashes) == 3 for hashes in depth3.values())
447: (8)                    assert all(len(hashes) == 4 for hashes in depth4.values())
448: (8)                    assert all(len(hashes) == 5 for hashes in depth5.values())
449: (8)                    assert is_subiteration(depth3, depth4)
450: (8)                    assert is_subiteration(depth4, depth5)
451: (8)                    assert is_subiteration(depth3, depth5)
452: (0)            def test_iteration_depth_node_attr():
453: (4)                """
454: (4)                All nodes should have the correct number of subgraph hashes in the output
when
455: (4)                setting initial node labels to an attribute.
456: (4)                Subsequent iteration depths for the same graph should be additive for each
node
457: (4)                """
458: (4)                n, r = 100, 10
459: (4)                p = 1.0 / r
460: (4)                for i in range(1, r + 1):
```

```
461: (8)                    G = nx.erdos_renyi_graph(n, p * i, seed=800 + i)
462: (8)                    for u in G.nodes():
463: (12)                       G.nodes[u]["node_attr1"] = f"{u}-1"
464: (8)                    depth3 = nx.weisfeiler_lehman_subgraph_hashes(
465: (12)                       G, node_attr="node_attr1", iterations=3
466: (8)                    )
467: (8)                    depth4 = nx.weisfeiler_lehman_subgraph_hashes(
468: (12)                       G, node_attr="node_attr1", iterations=4
469: (8)                    )
470: (8)                    depth5 = nx.weisfeiler_lehman_subgraph_hashes(
471: (12)                       G, node_attr="node_attr1", iterations=5
472: (8)                    )
473: (8)                    assert all(len(hashes) == 3 for hashes in depth3.values())
474: (8)                    assert all(len(hashes) == 4 for hashes in depth4.values())
475: (8)                    assert all(len(hashes) == 5 for hashes in depth5.values())
476: (8)                    assert is_subiteration(depth3, depth4)
477: (8)                    assert is_subiteration(depth4, depth5)
478: (8)                    assert is_subiteration(depth3, depth5)
479: (0)            def test_iteration_depth_node_edge_attr():
480: (4)                """
481: (4)                All nodes should have the correct number of subgraph hashes in the output
when
482: (4)                setting initial node labels to an attribute and also using an edge
attribute when
483: (4)                aggregating neighborhoods.
484: (4)                Subsequent iteration depths for the same graph should be additive for each
node
485: (4)                """
486: (4)                n, r = 100, 10
487: (4)                p = 1.0 / r
488: (4)                for i in range(1, r + 1):
489: (8)                    G = nx.erdos_renyi_graph(n, p * i, seed=900 + i)
490: (8)                    for u in G.nodes():
491: (12)                       G.nodes[u]["node_attr1"] = f"{u}-1"
492: (8)                    for a, b in G.edges:
493: (12)                       G[a][b]["edge_attr1"] = f"{a}-{b}-1"
494: (8)                    depth3 = nx.weisfeiler_lehman_subgraph_hashes(
495: (12)                       G, edge_attr="edge_attr1", node_attr="node_attr1", iterations=3
496: (8)                    )
497: (8)                    depth4 = nx.weisfeiler_lehman_subgraph_hashes(
498: (12)                       G, edge_attr="edge_attr1", node_attr="node_attr1", iterations=4
499: (8)                    )
500: (8)                    depth5 = nx.weisfeiler_lehman_subgraph_hashes(
501: (12)                       G, edge_attr="edge_attr1", node_attr="node_attr1", iterations=5
502: (8)                    )
503: (8)                    assert all(len(hashes) == 3 for hashes in depth3.values())
504: (8)                    assert all(len(hashes) == 4 for hashes in depth4.values())
505: (8)                    assert all(len(hashes) == 5 for hashes in depth5.values())
506: (8)                    assert is_subiteration(depth3, depth4)
507: (8)                    assert is_subiteration(depth4, depth5)
508: (8)                    assert is_subiteration(depth3, depth5)
509: (0)            def test_digest_size_subgraph_hash():
510: (4)                """
511: (4)                The hash string lengths should be as expected for a variety of graphs and
512: (4)                digest sizes
513: (4)                """
514: (4)                n, r = 100, 10
515: (4)                p = 1.0 / r
516: (4)                for i in range(1, r + 1):
517: (8)                    G = nx.erdos_renyi_graph(n, p * i, seed=1000 + i)
518: (8)                    digest_size16_hashes = nx.weisfeiler_lehman_subgraph_hashes(G)
519: (8)                    digest_size32_hashes = nx.weisfeiler_lehman_subgraph_hashes(G,
digest_size=32)
520: (8)                    assert digest_size16_hashes != digest_size32_hashes
521: (8)                    assert hexdigest_sizes_correct(digest_size16_hashes, 16)
522: (8)                    assert hexdigest_sizes_correct(digest_size32_hashes, 32)
523: (0)            def test_initial_node_labels_subgraph_hash():
524: (4)                """
525: (4)                Including the hashed initial label prepends an extra hash to the lists
```

```
526: (4)                """
527: (4)                G = nx.path_graph(5)
528: (4)                nx.set_node_attributes(G, {i: int(0 < i < 4) for i in G}, "label")
529: (4)                without_initial_label = nx.weisfeiler_lehman_subgraph_hashes(G,
node_attr="label")
530: (4)                assert all(len(v) == 3 for v in without_initial_label.values())
531: (4)                assert len({v[0] for v in without_initial_label.values()}) == 3
532: (4)                with_initial_label = nx.weisfeiler_lehman_subgraph_hashes(
533: (8)                    G, node_attr="label", include_initial_labels=True
534: (4)                )
535: (4)                assert all(len(v) == 4 for v in with_initial_label.values())
536: (4)                assert len({v[0] for v in with_initial_label.values()}) == 2
537: (4)                for u in G:
538: (8)                    for a, b in zip(
539: (12)                       with_initial_label[u][1:], without_initial_label[u], strict=True
540: (8)                    ):
541: (12)                       assert a == b

----------------------------------------

File 328 - test_hybrid.py:

1: (0)              import networkx as nx
2: (0)              def test_2d_grid_graph():
3: (4)                  G = nx.grid_2d_graph(8, 8, periodic=True)
4: (4)                  assert nx.is_kl_connected(G, 3, 3)
5: (4)                  assert not nx.is_kl_connected(G, 5, 9)
6: (4)                  (H, graphOK) = nx.kl_connected_subgraph(G, 5, 9, same_as_graph=True)
7: (4)                  assert not graphOK
8: (0)              def test_small_graph():
9: (4)                  G = nx.Graph()
10: (4)                 G.add_edge(1, 2)
11: (4)                 G.add_edge(1, 3)
12: (4)                 G.add_edge(2, 3)
13: (4)                 assert nx.is_kl_connected(G, 2, 2)
14: (4)                 H = nx.kl_connected_subgraph(G, 2, 2)
15: (4)                 (H, graphOK) = nx.kl_connected_subgraph(
16: (8)                     G, 2, 2, low_memory=True, same_as_graph=True
17: (4)                 )
18: (4)                 assert graphOK

----------------------------------------

File 329 - test_graphical.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_valid_degree_sequence1():
4: (4)                  n = 100
5: (4)                  p = 0.3
6: (4)                  for i in range(10):
7: (8)                      G = nx.erdos_renyi_graph(n, p)
8: (8)                      deg = (d for n, d in G.degree())
9: (8)                      assert nx.is_graphical(deg, method="eg")
10: (8)                     assert nx.is_graphical(deg, method="hh")
11: (0)             def test_valid_degree_sequence2():
12: (4)                 n = 100
13: (4)                 for i in range(10):
14: (8)                     G = nx.barabasi_albert_graph(n, 1)
15: (8)                     deg = (d for n, d in G.degree())
16: (8)                     assert nx.is_graphical(deg, method="eg")
17: (8)                     assert nx.is_graphical(deg, method="hh")
18: (0)             def test_string_input():
19: (4)                 pytest.raises(nx.NetworkXException, nx.is_graphical, [], "foo")
20: (4)                 pytest.raises(nx.NetworkXException, nx.is_graphical, ["red"], "hh")
21: (4)                 pytest.raises(nx.NetworkXException, nx.is_graphical, ["red"], "eg")
22: (0)             def test_non_integer_input():
23: (4)                 pytest.raises(nx.NetworkXException, nx.is_graphical, [72.5], "eg")
24: (4)                 pytest.raises(nx.NetworkXException, nx.is_graphical, [72.5], "hh")
```

```
(0)  def test_negative_input():
(4)      assert not nx.is_graphical([-1], "hh")
(4)      assert not nx.is_graphical([-1], "eg")
(0)  class TestAtlas:
(4)      @classmethod
(4)      def setup_class(cls):
(8)          global atlas
(8)          from networkx.generators import atlas
(8)          cls.GAG = atlas.graph_atlas_g()
(4)      def test_atlas(self):
(8)          for graph in self.GAG:
(12)             deg = (d for n, d in graph.degree())
(12)             assert nx.is_graphical(deg, method="eg")
(12)             assert nx.is_graphical(deg, method="hh")
(0)  def test_small_graph_true():
(4)      z = [5, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
(4)      assert nx.is_graphical(z, method="hh")
(4)      assert nx.is_graphical(z, method="eg")
(4)      z = [10, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2]
(4)      assert nx.is_graphical(z, method="hh")
(4)      assert nx.is_graphical(z, method="eg")
(4)      z = [1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
(4)      assert nx.is_graphical(z, method="hh")
(4)      assert nx.is_graphical(z, method="eg")
(0)  def test_small_graph_false():
(4)      z = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
(4)      assert not nx.is_graphical(z, method="hh")
(4)      assert not nx.is_graphical(z, method="eg")
(4)      z = [6, 5, 4, 4, 2, 1, 1, 1]
(4)      assert not nx.is_graphical(z, method="hh")
(4)      assert not nx.is_graphical(z, method="eg")
(4)      z = [1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
(4)      assert not nx.is_graphical(z, method="hh")
(4)      assert not nx.is_graphical(z, method="eg")
(0)  def test_directed_degree_sequence():
(4)      n, r = 100, 10
(4)      p = 1.0 / r
(4)      for i in range(r):
(8)          G = nx.erdos_renyi_graph(n, p * (i + 1), None, True)
(8)          din = (d for n, d in G.in_degree())
(8)          dout = (d for n, d in G.out_degree())
(8)          assert nx.is_digraphical(din, dout)
(0)  def test_small_directed_sequences():
(4)      dout = [5, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
(4)      din = [3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1]
(4)      assert nx.is_digraphical(din, dout)
(4)      dout = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
(4)      din = [103, 102, 102, 102, 102, 102, 102, 102, 102, 102]
(4)      assert not nx.is_digraphical(din, dout)
(4)      dout = [1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
(4)      din = [2, 2, 2, 2, 2, 2, 2, 2, 1, 1]
(4)      assert nx.is_digraphical(din, dout)
(4)      din = [2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1]
(4)      assert not nx.is_digraphical(din, dout)
(4)      din = [2, 2, 2, -2, 2, 2, 2, 2, 1, 1, 4]
(4)      assert not nx.is_digraphical(din, dout)
(4)      din = dout = [1, 1, 1.1, 1]
(4)      assert not nx.is_digraphical(din, dout)
(4)      din = dout = [1, 1, "rer", 1]
(4)      assert not nx.is_digraphical(din, dout)
(0)  def test_multi_sequence():
(4)      seq = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1]
(4)      assert not nx.is_multigraphical(seq)
(4)      seq = [6, 5, 4, 4, 2, 1, 1, 1]
(4)      assert nx.is_multigraphical(seq)
(4)      seq = [6, 5, 4, -4, 2, 1, 1, 1]
(4)      assert not nx.is_multigraphical(seq)
(4)      seq = [1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
(4)      assert not nx.is_multigraphical(seq)
```

```
94: (4)                 seq = [1, 1, 1.1, 1]
95: (4)                 assert not nx.is_multigraphical(seq)
96: (4)                 seq = [1, 1, "rer", 1]
97: (4)                 assert not nx.is_multigraphical(seq)
98: (0)             def test_pseudo_sequence():
99: (4)                 seq = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1]
100: (4)                assert nx.is_pseudographical(seq)
101: (4)                seq = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
102: (4)                assert not nx.is_pseudographical(seq)
103: (4)                seq = [1000, 3, 3, 3, 3, 2, 2, -2, 1, 1]
104: (4)                assert not nx.is_pseudographical(seq)
105: (4)                seq = [1, 1, 1.1, 1]
106: (4)                assert not nx.is_pseudographical(seq)
107: (4)                seq = [1, 1, "rer", 1]
108: (4)                assert not nx.is_pseudographical(seq)
109: (0)            def test_numpy_degree_sequence():
110: (4)                np = pytest.importorskip("numpy")
111: (4)                ds = np.array([1, 2, 2, 2, 1], dtype=np.int64)
112: (4)                assert nx.is_graphical(ds, "eg")
113: (4)                assert nx.is_graphical(ds, "hh")
114: (4)                ds = np.array([1, 2, 2, 2, 1], dtype=np.float64)
115: (4)                assert nx.is_graphical(ds, "eg")
116: (4)                assert nx.is_graphical(ds, "hh")
117: (4)                ds = np.array([1.1, 2, 2, 2, 1], dtype=np.float64)
118: (4)                pytest.raises(nx.NetworkXException, nx.is_graphical, ds, "eg")
119: (4)                pytest.raises(nx.NetworkXException, nx.is_graphical, ds, "hh")

----------------------------------------

File 330 - test_hierarchy.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_hierarchy_exception():
4: (4)                  G = nx.cycle_graph(5)
5: (4)                  pytest.raises(nx.NetworkXError, nx.flow_hierarchy, G)
6: (0)              def test_hierarchy_cycle():
7: (4)                  G = nx.cycle_graph(5, create_using=nx.DiGraph())
8: (4)                  assert nx.flow_hierarchy(G) == 0.0
9: (0)              def test_hierarchy_tree():
10: (4)                 G = nx.full_rary_tree(2, 16, create_using=nx.DiGraph())
11: (4)                 assert nx.flow_hierarchy(G) == 1.0
12: (0)             def test_hierarchy_1():
13: (4)                 G = nx.DiGraph()
14: (4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 1), (3, 4), (0, 4)])
15: (4)                 assert nx.flow_hierarchy(G) == 0.5
16: (0)             def test_hierarchy_weight():
17: (4)                 G = nx.DiGraph()
18: (4)                 G.add_edges_from(
19: (8)                     [
20: (12)                        (0, 1, {"weight": 0.3}),
21: (12)                        (1, 2, {"weight": 0.1}),
22: (12)                        (2, 3, {"weight": 0.1}),
23: (12)                        (3, 1, {"weight": 0.1}),
24: (12)                        (3, 4, {"weight": 0.3}),
25: (12)                        (0, 4, {"weight": 0.3}),
26: (8)                     ]
27: (4)                 )
28: (4)                 assert nx.flow_hierarchy(G, weight="weight") == 0.75

----------------------------------------

File 331 - test_isolate.py:

1: (0)              """Unit tests for the :mod:`networkx.algorithms.isolates` module."""
2: (0)              import networkx as nx
3: (0)              def test_is_isolate():
4: (4)                  G = nx.Graph()
5: (4)                  G.add_edge(0, 1)
```

```
6: (4)                  G.add_node(2)
7: (4)                  assert not nx.is_isolate(G, 0)
8: (4)                  assert not nx.is_isolate(G, 1)
9: (4)                  assert nx.is_isolate(G, 2)
10: (0)             def test_isolates():
11: (4)                 G = nx.Graph()
12: (4)                 G.add_edge(0, 1)
13: (4)                 G.add_nodes_from([2, 3])
14: (4)                 assert sorted(nx.isolates(G)) == [2, 3]
15: (0)             def test_number_of_isolates():
16: (4)                 G = nx.Graph()
17: (4)                 G.add_edge(0, 1)
18: (4)                 G.add_nodes_from([2, 3])
19: (4)                 assert nx.number_of_isolates(G) == 2
```

File 332 - test_link_prediction.py:

```
1: (0)              import math
2: (0)              from functools import partial
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              def _test_func(G, ebunch, expected, predict_func, **kwargs):
6: (4)                  result = predict_func(G, ebunch, **kwargs)
7: (4)                  exp_dict = {tuple(sorted([u, v])): score for u, v, score in expected}
8: (4)                  res_dict = {tuple(sorted([u, v])): score for u, v, score in result}
9: (4)                  assert len(exp_dict) == len(res_dict)
10: (4)                 for p in exp_dict:
11: (8)                     assert exp_dict[p] == pytest.approx(res_dict[p], abs=1e-7)
12: (0)             class TestResourceAllocationIndex:
13: (4)                 @classmethod
14: (4)                 def setup_class(cls):
15: (8)                     cls.func = staticmethod(nx.resource_allocation_index)
16: (8)                     cls.test = partial(_test_func, predict_func=cls.func)
17: (4)                 def test_K5(self):
18: (8)                     G = nx.complete_graph(5)
19: (8)                     self.test(G, [(0, 1)], [(0, 1, 0.75)])
20: (4)                 def test_P3(self):
21: (8)                     G = nx.path_graph(3)
22: (8)                     self.test(G, [(0, 2)], [(0, 2, 0.5)])
23: (4)                 def test_S4(self):
24: (8)                     G = nx.star_graph(4)
25: (8)                     self.test(G, [(1, 2)], [(1, 2, 0.25)])
26: (4)                 @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph,
nx.MultiDiGraph))
27: (4)                 def test_notimplemented(self, graph_type):
28: (8)                     assert pytest.raises(
29: (12)                        nx.NetworkXNotImplemented, self.func, graph_type([(0, 1), (1,
2)]), [(0, 2)]
30: (8)                     )
31: (4)                 def test_node_not_found(self):
32: (8)                     G = nx.Graph()
33: (8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
34: (8)                     assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
35: (4)                 def test_no_common_neighbor(self):
36: (8)                     G = nx.Graph()
37: (8)                     G.add_nodes_from([0, 1])
38: (8)                     self.test(G, [(0, 1)], [(0, 1, 0)])
39: (4)                 def test_equal_nodes(self):
40: (8)                     G = nx.complete_graph(4)
41: (8)                     self.test(G, [(0, 0)], [(0, 0, 1)])
42: (4)                 def test_all_nonexistent_edges(self):
43: (8)                     G = nx.Graph()
44: (8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
45: (8)                     self.test(G, None, [(0, 3, 0.5), (1, 2, 0.5), (1, 3, 0)])
46: (0)             class TestJaccardCoefficient:
47: (4)                 @classmethod
48: (4)                 def setup_class(cls):
```

```
(8)                     cls.func = staticmethod(nx.jaccard_coefficient)
(8)                     cls.test = partial(_test_func, predict_func=cls.func)
(4)                 def test_K5(self):
(8)                     G = nx.complete_graph(5)
(8)                     self.test(G, [(0, 1)], [(0, 1, 0.6)])
(4)                 def test_P4(self):
(8)                     G = nx.path_graph(4)
(8)                     self.test(G, [(0, 2)], [(0, 2, 0.5)])
(4)                 @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph))
(4)                 def test_notimplemented(self, graph_type):
(8)                     assert pytest.raises(
(12)                        nx.NetworkXNotImplemented, self.func, graph_type([(0, 1), (1, 2)]), [(0, 2)]
(8)                     )
(4)                 def test_node_not_found(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
(4)                 def test_no_common_neighbor(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (2, 3)])
(8)                     self.test(G, [(0, 2)], [(0, 2, 0)])
(4)                 def test_isolated_nodes(self):
(8)                     G = nx.Graph()
(8)                     G.add_nodes_from([0, 1])
(8)                     self.test(G, [(0, 1)], [(0, 1, 0)])
(4)                 def test_all_nonexistent_edges(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     self.test(G, None, [(0, 3, 0.5), (1, 2, 0.5), (1, 3, 0)])
(0)             class TestAdamicAdarIndex:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.func = staticmethod(nx.adamic_adar_index)
(8)                     cls.test = partial(_test_func, predict_func=cls.func)
(4)                 def test_K5(self):
(8)                     G = nx.complete_graph(5)
(8)                     self.test(G, [(0, 1)], [(0, 1, 3 / math.log(4))])
(4)                 def test_P3(self):
(8)                     G = nx.path_graph(3)
(8)                     self.test(G, [(0, 2)], [(0, 2, 1 / math.log(2))])
(4)                 def test_S4(self):
(8)                     G = nx.star_graph(4)
(8)                     self.test(G, [(1, 2)], [(1, 2, 1 / math.log(4))])
(4)                 @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph))
(4)                 def test_notimplemented(self, graph_type):
(8)                     assert pytest.raises(
(12)                        nx.NetworkXNotImplemented, self.func, graph_type([(0, 1), (1, 2)]), [(0, 2)]
(8)                     )
(4)                 def test_node_not_found(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
(4)                 def test_no_common_neighbor(self):
(8)                     G = nx.Graph()
(8)                     G.add_nodes_from([0, 1])
(8)                     self.test(G, [(0, 1)], [(0, 1, 0)])
(4)                 def test_equal_nodes(self):
(8)                     G = nx.complete_graph(4)
(8)                     self.test(G, [(0, 0)], [(0, 0, 3 / math.log(3))])
(4)                 def test_all_nonexistent_edges(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     self.test(
(12)                        G, None, [(0, 3, 1 / math.log(2)), (1, 2, 1 / math.log(2)), (1, 3, 0)]
```

```
113: (8)                    )
114: (0)            class TestCommonNeighborCentrality:
115: (4)                @classmethod
116: (4)                def setup_class(cls):
117: (8)                    cls.func = staticmethod(nx.common_neighbor_centrality)
118: (8)                    cls.test = partial(_test_func, predict_func=cls.func)
119: (4)                def test_K5(self):
120: (8)                    G = nx.complete_graph(5)
121: (8)                    self.test(G, [(0, 1)], [(0, 1, 3.0)], alpha=1)
122: (8)                    self.test(G, [(0, 1)], [(0, 1, 5.0)], alpha=0)
123: (4)                def test_P3(self):
124: (8)                    G = nx.path_graph(3)
125: (8)                    self.test(G, [(0, 2)], [(0, 2, 1.25)], alpha=0.5)
126: (4)                def test_S4(self):
127: (8)                    G = nx.star_graph(4)
128: (8)                    self.test(G, [(1, 2)], [(1, 2, 1.75)], alpha=0.5)
129: (4)                @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph,
nx.MultiDiGraph))
130: (4)                def test_notimplemented(self, graph_type):
131: (8)                    assert pytest.raises(
132: (12)                       nx.NetworkXNotImplemented, self.func, graph_type([(0, 1), (1,
2)]), [(0, 2)]
133: (8)                    )
134: (4)                def test_node_u_not_found(self):
135: (8)                    G = nx.Graph()
136: (8)                    G.add_edges_from([(1, 3), (2, 3)])
137: (8)                    assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 1)])
138: (4)                def test_node_v_not_found(self):
139: (8)                    G = nx.Graph()
140: (8)                    G.add_edges_from([(0, 1), (0, 2), (2, 3)])
141: (8)                    assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
142: (4)                def test_no_common_neighbor(self):
143: (8)                    G = nx.Graph()
144: (8)                    G.add_nodes_from([0, 1])
145: (8)                    self.test(G, [(0, 1)], [(0, 1, 0)])
146: (4)                def test_equal_nodes(self):
147: (8)                    G = nx.complete_graph(4)
148: (8)                    assert pytest.raises(nx.NetworkXAlgorithmError, self.test, G, [(0,
0)], [])
149: (4)                def test_equal_nodes_with_alpha_one_raises_error(self):
150: (8)                    G = nx.complete_graph(4)
151: (8)                    assert pytest.raises(
152: (12)                       nx.NetworkXAlgorithmError, self.test, G, [(0, 0)], [], alpha=1.0
153: (8)                    )
154: (4)                def test_all_nonexistent_edges(self):
155: (8)                    G = nx.Graph()
156: (8)                    G.add_edges_from([(0, 1), (0, 2), (2, 3)])
157: (8)                    self.test(G, None, [(0, 3, 1.5), (1, 2, 1.5), (1, 3, 2 / 3)],
alpha=0.5)
158: (0)            class TestPreferentialAttachment:
159: (4)                @classmethod
160: (4)                def setup_class(cls):
161: (8)                    cls.func = staticmethod(nx.preferential_attachment)
162: (8)                    cls.test = partial(_test_func, predict_func=cls.func)
163: (4)                def test_K5(self):
164: (8)                    G = nx.complete_graph(5)
165: (8)                    self.test(G, [(0, 1)], [(0, 1, 16)])
166: (4)                def test_P3(self):
167: (8)                    G = nx.path_graph(3)
168: (8)                    self.test(G, [(0, 1)], [(0, 1, 2)])
169: (4)                def test_S4(self):
170: (8)                    G = nx.star_graph(4)
171: (8)                    self.test(G, [(0, 2)], [(0, 2, 4)])
172: (4)                @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph,
nx.MultiDiGraph))
173: (4)                def test_notimplemented(self, graph_type):
174: (8)                    assert pytest.raises(
175: (12)                       nx.NetworkXNotImplemented, self.func, graph_type([(0, 1), (1,
2)]), [(0, 2)]
```

```
176: (8)                    )
177: (4)                def test_node_not_found(self):
178: (8)                    G = nx.Graph()
179: (8)                    G.add_edges_from([(0, 1), (0, 2), (2, 3)])
180: (8)                    assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
181: (4)                def test_zero_degrees(self):
182: (8)                    G = nx.Graph()
183: (8)                    G.add_nodes_from([0, 1])
184: (8)                    self.test(G, [(0, 1)], [(0, 1, 0)])
185: (4)                def test_all_nonexistent_edges(self):
186: (8)                    G = nx.Graph()
187: (8)                    G.add_edges_from([(0, 1), (0, 2), (2, 3)])
188: (8)                    self.test(G, None, [(0, 3, 2), (1, 2, 2), (1, 3, 1)])
189: (0)            class TestCNSoundarajanHopcroft:
190: (4)                @classmethod
191: (4)                def setup_class(cls):
192: (8)                    cls.func = staticmethod(nx.cn_soundarajan_hopcroft)
193: (8)                    cls.test = partial(_test_func, predict_func=cls.func,
community="community")
194: (4)                def test_K5(self):
195: (8)                    G = nx.complete_graph(5)
196: (8)                    G.nodes[0]["community"] = 0
197: (8)                    G.nodes[1]["community"] = 0
198: (8)                    G.nodes[2]["community"] = 0
199: (8)                    G.nodes[3]["community"] = 0
200: (8)                    G.nodes[4]["community"] = 1
201: (8)                    self.test(G, [(0, 1)], [(0, 1, 5)])
202: (4)                def test_P3(self):
203: (8)                    G = nx.path_graph(3)
204: (8)                    G.nodes[0]["community"] = 0
205: (8)                    G.nodes[1]["community"] = 1
206: (8)                    G.nodes[2]["community"] = 0
207: (8)                    self.test(G, [(0, 2)], [(0, 2, 1)])
208: (4)                def test_S4(self):
209: (8)                    G = nx.star_graph(4)
210: (8)                    G.nodes[0]["community"] = 1
211: (8)                    G.nodes[1]["community"] = 1
212: (8)                    G.nodes[2]["community"] = 1
213: (8)                    G.nodes[3]["community"] = 0
214: (8)                    G.nodes[4]["community"] = 0
215: (8)                    self.test(G, [(1, 2)], [(1, 2, 2)])
216: (4)                @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph,
nx.MultiDiGraph))
217: (4)                def test_notimplemented(self, graph_type):
218: (8)                    G = graph_type([(0, 1), (1, 2)])
219: (8)                    G.add_nodes_from([0, 1, 2], community=0)
220: (8)                    assert pytest.raises(nx.NetworkXNotImplemented, self.func, G, [(0,
2)])
221: (4)                def test_node_not_found(self):
222: (8)                    G = nx.Graph()
223: (8)                    G.add_edges_from([(0, 1), (0, 2), (2, 3)])
224: (8)                    G.nodes[0]["community"] = 0
225: (8)                    G.nodes[1]["community"] = 1
226: (8)                    G.nodes[2]["community"] = 0
227: (8)                    G.nodes[3]["community"] = 0
228: (8)                    assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
229: (4)                def test_no_common_neighbor(self):
230: (8)                    G = nx.Graph()
231: (8)                    G.add_nodes_from([0, 1])
232: (8)                    G.nodes[0]["community"] = 0
233: (8)                    G.nodes[1]["community"] = 0
234: (8)                    self.test(G, [(0, 1)], [(0, 1, 0)])
235: (4)                def test_equal_nodes(self):
236: (8)                    G = nx.complete_graph(3)
237: (8)                    G.nodes[0]["community"] = 0
238: (8)                    G.nodes[1]["community"] = 0
239: (8)                    G.nodes[2]["community"] = 0
240: (8)                    self.test(G, [(0, 0)], [(0, 0, 4)])
241: (4)                def test_different_community(self):
```

```
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 1
(8)                     self.test(G, [(0, 3)], [(0, 3, 2)])
(4)                 def test_no_community_information(self):
(8)                     G = nx.complete_graph(5)
(8)                     assert pytest.raises(nx.NetworkXAlgorithmError, list, self.func(G,
[(0, 1)]))
(4)                 def test_insufficient_community_information(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     assert pytest.raises(nx.NetworkXAlgorithmError, list, self.func(G,
[(0, 3)]))
(4)                 def test_sufficient_community_information(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5)])
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     G.nodes[4]["community"] = 0
(8)                     self.test(G, [(1, 4)], [(1, 4, 4)])
(4)                 def test_custom_community_attribute_name(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["cmty"] = 0
(8)                     G.nodes[1]["cmty"] = 0
(8)                     G.nodes[2]["cmty"] = 0
(8)                     G.nodes[3]["cmty"] = 1
(8)                     self.test(G, [(0, 3)], [(0, 3, 2)], community="cmty")
(4)                 def test_all_nonexistent_edges(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     self.test(G, None, [(0, 3, 2), (1, 2, 1), (1, 3, 0)])
(0)             class TestRAIndexSoundarajanHopcroft:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.func = staticmethod(nx.ra_index_soundarajan_hopcroft)
(8)                     cls.test = partial(_test_func, predict_func=cls.func,
community="community")
(4)                 def test_K5(self):
(8)                     G = nx.complete_graph(5)
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     G.nodes[4]["community"] = 1
(8)                     self.test(G, [(0, 1)], [(0, 1, 0.5)])
(4)                 def test_P3(self):
(8)                     G = nx.path_graph(3)
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 0
(8)                     self.test(G, [(0, 2)], [(0, 2, 0)])
(4)                 def test_S4(self):
(8)                     G = nx.star_graph(4)
(8)                     G.nodes[0]["community"] = 1
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 1
(8)                     G.nodes[3]["community"] = 0
```

```
(8)                     G.nodes[4]["community"] = 0
(8)                     self.test(G, [(1, 2)], [(1, 2, 0.25)])
(4)                 @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph))
(4)                 def test_notimplemented(self, graph_type):
(8)                     G = graph_type([(0, 1), (1, 2)])
(8)                     G.add_nodes_from([0, 1, 2], community=0)
(8)                     assert pytest.raises(nx.NetworkXNotImplemented, self.func, G, [(0, 2)])
(4)                 def test_node_not_found(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
(4)                 def test_no_common_neighbor(self):
(8)                     G = nx.Graph()
(8)                     G.add_nodes_from([0, 1])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     self.test(G, [(0, 1)], [(0, 1, 0)])
(4)                 def test_equal_nodes(self):
(8)                     G = nx.complete_graph(3)
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     self.test(G, [(0, 0)], [(0, 0, 1)])
(4)                 def test_different_community(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 1
(8)                     self.test(G, [(0, 3)], [(0, 3, 0)])
(4)                 def test_no_community_information(self):
(8)                     G = nx.complete_graph(5)
(8)                     assert pytest.raises(nx.NetworkXAlgorithmError, list, self.func(G, [(0, 1)]))
(4)                 def test_insufficient_community_information(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     assert pytest.raises(nx.NetworkXAlgorithmError, list, self.func(G, [(0, 3)]))
(4)                 def test_sufficient_community_information(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5)])
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     G.nodes[4]["community"] = 0
(8)                     self.test(G, [(1, 4)], [(1, 4, 1)])
(4)                 def test_custom_community_attribute_name(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["cmty"] = 0
(8)                     G.nodes[1]["cmty"] = 0
(8)                     G.nodes[2]["cmty"] = 0
(8)                     G.nodes[3]["cmty"] = 1
(8)                     self.test(G, [(0, 3)], [(0, 3, 0)], community="cmty")
(4)                 def test_all_nonexistent_edges(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
```

```
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     self.test(G, None, [(0, 3, 0.5), (1, 2, 0), (1, 3, 0)])
(0)             class TestWithinInterCluster:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.delta = 0.001
(8)                     cls.func = staticmethod(nx.within_inter_cluster)
(8)                     cls.test = partial(
(12)                        _test_func, predict_func=cls.func, delta=cls.delta,
community="community"
(8)                     )
(4)                 def test_K5(self):
(8)                     G = nx.complete_graph(5)
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     G.nodes[4]["community"] = 1
(8)                     self.test(G, [(0, 1)], [(0, 1, 2 / (1 + self.delta))])
(4)                 def test_P3(self):
(8)                     G = nx.path_graph(3)
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 0
(8)                     self.test(G, [(0, 2)], [(0, 2, 0)])
(4)                 def test_S4(self):
(8)                     G = nx.star_graph(4)
(8)                     G.nodes[0]["community"] = 1
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 1
(8)                     G.nodes[3]["community"] = 0
(8)                     G.nodes[4]["community"] = 0
(8)                     self.test(G, [(1, 2)], [(1, 2, 1 / self.delta)])
(4)                 @pytest.mark.parametrize("graph_type", (nx.DiGraph, nx.MultiGraph,
nx.MultiDiGraph))
(4)                 def test_notimplemented(self, graph_type):
(8)                     G = graph_type([(0, 1), (1, 2)])
(8)                     G.add_nodes_from([0, 1, 2], community=0)
(8)                     assert pytest.raises(nx.NetworkXNotImplemented, self.func, G, [(0,
2)])
(4)                 def test_node_not_found(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 1
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 0
(8)                     assert pytest.raises(nx.NodeNotFound, self.func, G, [(0, 4)])
(4)                 def test_no_common_neighbor(self):
(8)                     G = nx.Graph()
(8)                     G.add_nodes_from([0, 1])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     self.test(G, [(0, 1)], [(0, 1, 0)])
(4)                 def test_equal_nodes(self):
(8)                     G = nx.complete_graph(3)
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     self.test(G, [(0, 0)], [(0, 0, 2 / self.delta)])
(4)                 def test_different_community(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
(8)                     G.nodes[0]["community"] = 0
(8)                     G.nodes[1]["community"] = 0
(8)                     G.nodes[2]["community"] = 0
(8)                     G.nodes[3]["community"] = 1
```

```
439: (8)                    self.test(G, [(0, 3)], [(0, 3, 0)])
440: (4)                def test_no_inter_cluster_common_neighbor(self):
441: (8)                    G = nx.complete_graph(4)
442: (8)                    G.nodes[0]["community"] = 0
443: (8)                    G.nodes[1]["community"] = 0
444: (8)                    G.nodes[2]["community"] = 0
445: (8)                    G.nodes[3]["community"] = 0
446: (8)                    self.test(G, [(0, 3)], [(0, 3, 2 / self.delta)])
447: (4)                def test_no_community_information(self):
448: (8)                    G = nx.complete_graph(5)
449: (8)                    assert pytest.raises(nx.NetworkXAlgorithmError, list, self.func(G,
[(0, 1)]))
450: (4)                def test_insufficient_community_information(self):
451: (8)                    G = nx.Graph()
452: (8)                    G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
453: (8)                    G.nodes[0]["community"] = 0
454: (8)                    G.nodes[1]["community"] = 0
455: (8)                    G.nodes[3]["community"] = 0
456: (8)                    assert pytest.raises(nx.NetworkXAlgorithmError, list, self.func(G,
[(0, 3)]))
457: (4)                def test_sufficient_community_information(self):
458: (8)                    G = nx.Graph()
459: (8)                    G.add_edges_from([(0, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5)])
460: (8)                    G.nodes[1]["community"] = 0
461: (8)                    G.nodes[2]["community"] = 0
462: (8)                    G.nodes[3]["community"] = 0
463: (8)                    G.nodes[4]["community"] = 0
464: (8)                    self.test(G, [(1, 4)], [(1, 4, 2 / self.delta)])
465: (4)                def test_invalid_delta(self):
466: (8)                    G = nx.complete_graph(3)
467: (8)                    G.add_nodes_from([0, 1, 2], community=0)
468: (8)                    assert pytest.raises(nx.NetworkXAlgorithmError, self.func, G, [(0,
1)], 0)
469: (8)                    assert pytest.raises(nx.NetworkXAlgorithmError, self.func, G, [(0,
1)], -0.5)
470: (4)                def test_custom_community_attribute_name(self):
471: (8)                    G = nx.complete_graph(4)
472: (8)                    G.nodes[0]["cmty"] = 0
473: (8)                    G.nodes[1]["cmty"] = 0
474: (8)                    G.nodes[2]["cmty"] = 0
475: (8)                    G.nodes[3]["cmty"] = 0
476: (8)                    self.test(G, [(0, 3)], [(0, 3, 2 / self.delta)], community="cmty")
477: (4)                def test_all_nonexistent_edges(self):
478: (8)                    G = nx.Graph()
479: (8)                    G.add_edges_from([(0, 1), (0, 2), (2, 3)])
480: (8)                    G.nodes[0]["community"] = 0
481: (8)                    G.nodes[1]["community"] = 1
482: (8)                    G.nodes[2]["community"] = 0
483: (8)                    G.nodes[3]["community"] = 0
484: (8)                    self.test(G, None, [(0, 3, 1 / self.delta), (1, 2, 0), (1, 3, 0)])

----------------------------------------

File 333 - test_lowest_common_ancestors.py:

1: (0)              from itertools import chain, combinations, product
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              tree_all_pairs_lca = nx.tree_all_pairs_lowest_common_ancestor
5: (0)              all_pairs_lca = nx.all_pairs_lowest_common_ancestor
6: (0)              def get_pair(dictionary, n1, n2):
7: (4)                  if (n1, n2) in dictionary:
8: (8)                      return dictionary[n1, n2]
9: (4)                  else:
10: (8)                     return dictionary[n2, n1]
11: (0)             class TestTreeLCA:
12: (4)                 @classmethod
13: (4)                 def setup_class(cls):
14: (8)                     cls.DG = nx.DiGraph()
```

```
 15: (8)                    edges = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
 16: (8)                    cls.DG.add_edges_from(edges)
 17: (8)                    cls.ans = dict(tree_all_pairs_lca(cls.DG, 0))
 18: (8)                    gold = {(n, n): n for n in cls.DG}
 19: (8)                    gold.update({(0, i): 0 for i in range(1, 7)})
 20: (8)                    gold.update(
 21: (12)                       {
 22: (16)                           (1, 2): 0,
 23: (16)                           (1, 3): 1,
 24: (16)                           (1, 4): 1,
 25: (16)                           (1, 5): 0,
 26: (16)                           (1, 6): 0,
 27: (16)                           (2, 3): 0,
 28: (16)                           (2, 4): 0,
 29: (16)                           (2, 5): 2,
 30: (16)                           (2, 6): 2,
 31: (16)                           (3, 4): 1,
 32: (16)                           (3, 5): 0,
 33: (16)                           (3, 6): 0,
 34: (16)                           (4, 5): 0,
 35: (16)                           (4, 6): 0,
 36: (16)                           (5, 6): 2,
 37: (12)                       }
 38: (8)                    )
 39: (8)                    cls.gold = gold
 40: (4)                @staticmethod
 41: (4)                def assert_has_same_pairs(d1, d2):
 42: (8)                    for a, b in ((min(pair), max(pair)) for pair in chain(d1, d2)):
 43: (12)                       assert get_pair(d1, a, b) == get_pair(d2, a, b)
 44: (4)                def test_tree_all_pairs_lca_default_root(self):
 45: (8)                    assert dict(tree_all_pairs_lca(self.DG)) == self.ans
 46: (4)                def test_tree_all_pairs_lca_return_subset(self):
 47: (8)                    test_pairs = [(0, 1), (0, 1), (1, 0)]
 48: (8)                    ans = dict(tree_all_pairs_lca(self.DG, 0, test_pairs))
 49: (8)                    assert (0, 1) in ans and (1, 0) in ans
 50: (8)                    assert len(ans) == 2
 51: (4)                def test_tree_all_pairs_lca(self):
 52: (8)                    all_pairs = chain(combinations(self.DG, 2), ((node, node) for node in
self.DG))
 53: (8)                    ans = dict(tree_all_pairs_lca(self.DG, 0, all_pairs))
 54: (8)                    self.assert_has_same_pairs(ans, self.ans)
 55: (4)                def test_tree_all_pairs_gold_example(self):
 56: (8)                    ans = dict(tree_all_pairs_lca(self.DG))
 57: (8)                    self.assert_has_same_pairs(self.gold, ans)
 58: (4)                def test_tree_all_pairs_lca_invalid_input(self):
 59: (8)                    empty_digraph = tree_all_pairs_lca(nx.DiGraph())
 60: (8)                    pytest.raises(nx.NetworkXPointlessConcept, list, empty_digraph)
 61: (8)                    bad_pairs_digraph = tree_all_pairs_lca(self.DG, pairs=[(-1, -2)])
 62: (8)                    pytest.raises(nx.NodeNotFound, list, bad_pairs_digraph)
 63: (4)                def test_tree_all_pairs_lca_subtrees(self):
 64: (8)                    ans = dict(tree_all_pairs_lca(self.DG, 1))
 65: (8)                    gold = {
 66: (12)                       pair: lca
 67: (12)                       for (pair, lca) in self.gold.items()
 68: (12)                       if all(n in (1, 3, 4) for n in pair)
 69: (8)                    }
 70: (8)                    self.assert_has_same_pairs(gold, ans)
 71: (4)                def test_tree_all_pairs_lca_disconnected_nodes(self):
 72: (8)                    G = nx.DiGraph()
 73: (8)                    G.add_node(1)
 74: (8)                    assert {(1, 1): 1} == dict(tree_all_pairs_lca(G))
 75: (8)                    G.add_node(0)
 76: (8)                    assert {(1, 1): 1} == dict(tree_all_pairs_lca(G, 1))
 77: (8)                    assert {(0, 0): 0} == dict(tree_all_pairs_lca(G, 0))
 78: (8)                    pytest.raises(nx.NetworkXError, list, tree_all_pairs_lca(G))
 79: (4)                def test_tree_all_pairs_lca_error_if_input_not_tree(self):
 80: (8)                    G = nx.DiGraph([(1, 2), (2, 1)])
 81: (8)                    pytest.raises(nx.NetworkXError, list, tree_all_pairs_lca(G))
 82: (8)                    G = nx.DiGraph([(0, 2), (1, 2)])
```

```
 83: (8)                    pytest.raises(nx.NetworkXError, list, tree_all_pairs_lca(G))
 84: (4)                def test_tree_all_pairs_lca_generator(self):
 85: (8)                    pairs = iter([(0, 1), (0, 1), (1, 0)])
 86: (8)                    some_pairs = dict(tree_all_pairs_lca(self.DG, 0, pairs))
 87: (8)                    assert (0, 1) in some_pairs and (1, 0) in some_pairs
 88: (8)                    assert len(some_pairs) == 2
 89: (4)                def test_tree_all_pairs_lca_nonexisting_pairs_exception(self):
 90: (8)                    lca = tree_all_pairs_lca(self.DG, 0, [(-1, -1)])
 91: (8)                    pytest.raises(nx.NodeNotFound, list, lca)
 92: (8)                    lca = tree_all_pairs_lca(self.DG, None, [(-1, -1)])
 93: (8)                    pytest.raises(nx.NodeNotFound, list, lca)
 94: (4)                def test_tree_all_pairs_lca_routine_bails_on_DAGs(self):
 95: (8)                    G = nx.DiGraph([(3, 4), (5, 4)])
 96: (8)                    pytest.raises(nx.NetworkXError, list, tree_all_pairs_lca(G))
 97: (4)                def test_tree_all_pairs_lca_not_implemented(self):
 98: (8)                    NNI = nx.NetworkXNotImplemented
 99: (8)                    G = nx.Graph([(0, 1)])
100: (8)                    with pytest.raises(NNI):
101: (12)                       next(tree_all_pairs_lca(G))
102: (8)                    with pytest.raises(NNI):
103: (12)                       next(all_pairs_lca(G))
104: (8)                    pytest.raises(NNI, nx.lowest_common_ancestor, G, 0, 1)
105: (8)                    G = nx.MultiGraph([(0, 1)])
106: (8)                    with pytest.raises(NNI):
107: (12)                       next(tree_all_pairs_lca(G))
108: (8)                    with pytest.raises(NNI):
109: (12)                       next(all_pairs_lca(G))
110: (8)                    pytest.raises(NNI, nx.lowest_common_ancestor, G, 0, 1)
111: (4)                def test_tree_all_pairs_lca_trees_without_LCAs(self):
112: (8)                    G = nx.DiGraph()
113: (8)                    G.add_node(3)
114: (8)                    ans = list(tree_all_pairs_lca(G))
115: (8)                    assert ans == [((3, 3), 3)]
116: (0)            class TestMultiTreeLCA(TestTreeLCA):
117: (4)                @classmethod
118: (4)                def setup_class(cls):
119: (8)                    cls.DG = nx.MultiDiGraph()
120: (8)                    edges = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
121: (8)                    cls.DG.add_edges_from(edges)
122: (8)                    cls.ans = dict(tree_all_pairs_lca(cls.DG, 0))
123: (8)                    cls.DG.add_edges_from(edges)
124: (8)                    gold = {(n, n): n for n in cls.DG}
125: (8)                    gold.update({(0, i): 0 for i in range(1, 7)})
126: (8)                    gold.update(
127: (12)                       {
128: (16)                           (1, 2): 0,
129: (16)                           (1, 3): 1,
130: (16)                           (1, 4): 1,
131: (16)                           (1, 5): 0,
132: (16)                           (1, 6): 0,
133: (16)                           (2, 3): 0,
134: (16)                           (2, 4): 0,
135: (16)                           (2, 5): 2,
136: (16)                           (2, 6): 2,
137: (16)                           (3, 4): 1,
138: (16)                           (3, 5): 0,
139: (16)                           (3, 6): 0,
140: (16)                           (4, 5): 0,
141: (16)                           (4, 6): 0,
142: (16)                           (5, 6): 2,
143: (12)                       }
144: (8)                    )
145: (8)                    cls.gold = gold
146: (0)            class TestDAGLCA:
147: (4)                @classmethod
148: (4)                def setup_class(cls):
149: (8)                    cls.DG = nx.DiGraph()
150: (8)                    nx.add_path(cls.DG, (0, 1, 2, 3))
151: (8)                    nx.add_path(cls.DG, (0, 4, 3))
```

```
(8)                     nx.add_path(cls.DG, (0, 5, 6, 8, 3))
(8)                     nx.add_path(cls.DG, (5, 7, 8))
(8)                     cls.DG.add_edge(6, 2)
(8)                     cls.DG.add_edge(7, 2)
(8)                     cls.root_distance = nx.shortest_path_length(cls.DG, source=0)
(8)                     cls.gold = {
(12)                        (1, 1): 1,
(12)                        (1, 2): 1,
(12)                        (1, 3): 1,
(12)                        (1, 4): 0,
(12)                        (1, 5): 0,
(12)                        (1, 6): 0,
(12)                        (1, 7): 0,
(12)                        (1, 8): 0,
(12)                        (2, 2): 2,
(12)                        (2, 3): 2,
(12)                        (2, 4): 0,
(12)                        (2, 5): 5,
(12)                        (2, 6): 6,
(12)                        (2, 7): 7,
(12)                        (2, 8): 7,
(12)                        (3, 3): 3,
(12)                        (3, 4): 4,
(12)                        (3, 5): 5,
(12)                        (3, 6): 6,
(12)                        (3, 7): 7,
(12)                        (3, 8): 8,
(12)                        (4, 4): 4,
(12)                        (4, 5): 0,
(12)                        (4, 6): 0,
(12)                        (4, 7): 0,
(12)                        (4, 8): 0,
(12)                        (5, 5): 5,
(12)                        (5, 6): 5,
(12)                        (5, 7): 5,
(12)                        (5, 8): 5,
(12)                        (6, 6): 6,
(12)                        (6, 7): 5,
(12)                        (6, 8): 6,
(12)                        (7, 7): 7,
(12)                        (7, 8): 7,
(12)                        (8, 8): 8,
(8)                     }
(8)                     cls.gold.update(((0, n), 0) for n in cls.DG)
(4)                 def assert_lca_dicts_same(self, d1, d2, G=None):
(8)                     """Checks if d1 and d2 contain the same pairs and
(8)                     have a node at the same distance from root for each.
(8)                     If G is None use self.DG."""
(8)                     if G is None:
(12)                        G = self.DG
(12)                        root_distance = self.root_distance
(8)                     else:
(12)                        roots = [n for n, deg in G.in_degree if deg == 0]
(12)                        assert len(roots) == 1
(12)                        root_distance = nx.shortest_path_length(G, source=roots[0])
(8)                     for a, b in ((min(pair), max(pair)) for pair in chain(d1, d2)):
(12)                        assert (
(16)                            root_distance[get_pair(d1, a, b)] ==
root_distance[get_pair(d2, a, b)]
(12)                        )
(4)                 def test_all_pairs_lca_gold_example(self):
(8)                     self.assert_lca_dicts_same(dict(all_pairs_lca(self.DG)), self.gold)
(4)                 def test_all_pairs_lca_all_pairs_given(self):
(8)                     all_pairs = list(product(self.DG.nodes(), self.DG.nodes()))
(8)                     ans = all_pairs_lca(self.DG, pairs=all_pairs)
(8)                     self.assert_lca_dicts_same(dict(ans), self.gold)
(4)                 def test_all_pairs_lca_generator(self):
(8)                     all_pairs = product(self.DG.nodes(), self.DG.nodes())
(8)                     ans = all_pairs_lca(self.DG, pairs=all_pairs)
```

```
(8)                 self.assert_lca_dicts_same(dict(ans), self.gold)
(4)             def test_all_pairs_lca_input_graph_with_two_roots(self):
(8)                 G = self.DG.copy()
(8)                 G.add_edge(9, 10)
(8)                 G.add_edge(9, 4)
(8)                 gold = self.gold.copy()
(8)                 gold[9, 9] = 9
(8)                 gold[9, 10] = 9
(8)                 gold[9, 4] = 9
(8)                 gold[9, 3] = 9
(8)                 gold[10, 4] = 9
(8)                 gold[10, 3] = 9
(8)                 gold[10, 10] = 10
(8)                 testing = dict(all_pairs_lca(G))
(8)                 G.add_edge(-1, 9)
(8)                 G.add_edge(-1, 0)
(8)                 self.assert_lca_dicts_same(testing, gold, G)
(4)             def test_all_pairs_lca_nonexisting_pairs_exception(self):
(8)                 pytest.raises(nx.NodeNotFound, all_pairs_lca, self.DG, [(-1, -1)])
(4)             def test_all_pairs_lca_pairs_without_lca(self):
(8)                 G = self.DG.copy()
(8)                 G.add_node(-1)
(8)                 gen = all_pairs_lca(G, [(-1, -1), (-1, 0)])
(8)                 assert dict(gen) == {(-1, -1): -1}
(4)             def test_all_pairs_lca_null_graph(self):
(8)                 pytest.raises(nx.NetworkXPointlessConcept, all_pairs_lca,
nx.DiGraph())
(4)             def test_all_pairs_lca_non_dags(self):
(8)                 pytest.raises(nx.NetworkXError, all_pairs_lca, nx.DiGraph([(3, 4), (4,
3)]))
(4)             def test_all_pairs_lca_nonempty_graph_without_lca(self):
(8)                 G = nx.DiGraph()
(8)                 G.add_node(3)
(8)                 ans = list(all_pairs_lca(G))
(8)                 assert ans == [((3, 3), 3)]
(4)             def test_all_pairs_lca_bug_gh4942(self):
(8)                 G = nx.DiGraph([(0, 2), (1, 2), (2, 3)])
(8)                 ans = list(all_pairs_lca(G))
(8)                 assert len(ans) == 9
(4)             def test_all_pairs_lca_default_kwarg(self):
(8)                 G = nx.DiGraph([(0, 1), (2, 1)])
(8)                 sentinel = object()
(8)                 assert nx.lowest_common_ancestor(G, 0, 2, default=sentinel) is
sentinel
(4)             def test_all_pairs_lca_identity(self):
(8)                 G = nx.DiGraph()
(8)                 G.add_node(3)
(8)                 assert nx.lowest_common_ancestor(G, 3, 3) == 3
(4)             def test_all_pairs_lca_issue_4574(self):
(8)                 G = nx.DiGraph()
(8)                 G.add_nodes_from(range(17))
(8)                 G.add_edges_from(
(12)                    [
(16)                        (2, 0),
(16)                        (1, 2),
(16)                        (3, 2),
(16)                        (5, 2),
(16)                        (8, 2),
(16)                        (11, 2),
(16)                        (4, 5),
(16)                        (6, 5),
(16)                        (7, 8),
(16)                        (10, 8),
(16)                        (13, 11),
(16)                        (14, 11),
(16)                        (15, 11),
(16)                        (9, 10),
(16)                        (12, 13),
(16)                        (16, 15),
```

```
(12)                        ]
(8)                     )
(8)                     assert nx.lowest_common_ancestor(G, 7, 9) == None
(4)                 def test_all_pairs_lca_one_pair_gh4942(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_edge(0, 1)
(8)                     G.add_edge(2, 0)
(8)                     G.add_edge(2, 3)
(8)                     G.add_edge(4, 0)
(8)                     G.add_edge(5, 2)
(8)                     assert nx.lowest_common_ancestor(G, 1, 3) == 2
(0)             class TestMultiDiGraph_DAGLCA(TestDAGLCA):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.DG = nx.MultiDiGraph()
(8)                     nx.add_path(cls.DG, (0, 1, 2, 3))
(8)                     nx.add_path(cls.DG, (0, 1, 2, 3))
(8)                     nx.add_path(cls.DG, (0, 4, 3))
(8)                     nx.add_path(cls.DG, (0, 5, 6, 8, 3))
(8)                     nx.add_path(cls.DG, (5, 7, 8))
(8)                     cls.DG.add_edge(6, 2)
(8)                     cls.DG.add_edge(7, 2)
(8)                     cls.root_distance = nx.shortest_path_length(cls.DG, source=0)
(8)                     cls.gold = {
(12)                        (1, 1): 1,
(12)                        (1, 2): 1,
(12)                        (1, 3): 1,
(12)                        (1, 4): 0,
(12)                        (1, 5): 0,
(12)                        (1, 6): 0,
(12)                        (1, 7): 0,
(12)                        (1, 8): 0,
(12)                        (2, 2): 2,
(12)                        (2, 3): 2,
(12)                        (2, 4): 0,
(12)                        (2, 5): 5,
(12)                        (2, 6): 6,
(12)                        (2, 7): 7,
(12)                        (2, 8): 7,
(12)                        (3, 3): 3,
(12)                        (3, 4): 4,
(12)                        (3, 5): 5,
(12)                        (3, 6): 6,
(12)                        (3, 7): 7,
(12)                        (3, 8): 8,
(12)                        (4, 4): 4,
(12)                        (4, 5): 0,
(12)                        (4, 6): 0,
(12)                        (4, 7): 0,
(12)                        (4, 8): 0,
(12)                        (5, 5): 5,
(12)                        (5, 6): 5,
(12)                        (5, 7): 5,
(12)                        (5, 8): 5,
(12)                        (6, 6): 6,
(12)                        (6, 7): 5,
(12)                        (6, 8): 6,
(12)                        (7, 7): 7,
(12)                        (7, 8): 7,
(12)                        (8, 8): 8,
(8)                     }
(8)                     cls.gold.update(((0, n), 0) for n in cls.DG)
(0)             def test_all_pairs_lca_self_ancestors():
(4)                 """Self-ancestors should always be the node itself, i.e. lca of (0, 0) is
0.
(4)                 See gh-4458."""
(4)                 G = nx.DiGraph()
(4)                 G.add_nodes_from(range(5))
(4)                 G.add_edges_from([(1, 0), (2, 0), (3, 2), (4, 1), (4, 3)])
```

```
354: (4)                ap_lca = nx.all_pairs_lowest_common_ancestor
355: (4)                assert all(u == v == a for (u, v), a in ap_lca(G) if u == v)
356: (4)                MG = nx.MultiDiGraph(G)
357: (4)                assert all(u == v == a for (u, v), a in ap_lca(MG) if u == v)
358: (4)                MG.add_edges_from([(1, 0), (2, 0)])
359: (4)                assert all(u == v == a for (u, v), a in ap_lca(MG) if u == v)
```

File 334 - test_matching.py:

```
1: (0)              import math
2: (0)              from itertools import permutations
3: (0)              from pytest import raises
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.matching import matching_dict_to_set
6: (0)              from networkx.utils import edges_equal
7: (0)              class TestMaxWeightMatching:
8: (4)                  """Unit tests for the
9: (4)                  :func:`~networkx.algorithms.matching.max_weight_matching` function.
10: (4)                 """
11: (4)                 def test_trivial1(self):
12: (8)                     """Empty graph"""
13: (8)                     G = nx.Graph()
14: (8)                     assert nx.max_weight_matching(G) == set()
15: (8)                     assert nx.min_weight_matching(G) == set()
16: (4)                 def test_selfloop(self):
17: (8)                     G = nx.Graph()
18: (8)                     G.add_edge(0, 0, weight=100)
19: (8)                     assert nx.max_weight_matching(G) == set()
20: (8)                     assert nx.min_weight_matching(G) == set()
21: (4)                 def test_single_edge(self):
22: (8)                     G = nx.Graph()
23: (8)                     G.add_edge(0, 1)
24: (8)                     assert edges_equal(
25: (12)                        nx.max_weight_matching(G), matching_dict_to_set({0: 1, 1: 0})
26: (8)                     )
27: (8)                     assert edges_equal(
28: (12)                        nx.min_weight_matching(G), matching_dict_to_set({0: 1, 1: 0})
29: (8)                     )
30: (4)                 def test_two_path(self):
31: (8)                     G = nx.Graph()
32: (8)                     G.add_edge("one", "two", weight=10)
33: (8)                     G.add_edge("two", "three", weight=11)
34: (8)                     assert edges_equal(
35: (12)                        nx.max_weight_matching(G),
36: (12)                        matching_dict_to_set({"three": "two", "two": "three"}),
37: (8)                     )
38: (8)                     assert edges_equal(
39: (12)                        nx.min_weight_matching(G),
40: (12)                        matching_dict_to_set({"one": "two", "two": "one"}),
41: (8)                     )
42: (4)                 def test_path(self):
43: (8)                     G = nx.Graph()
44: (8)                     G.add_edge(1, 2, weight=5)
45: (8)                     G.add_edge(2, 3, weight=11)
46: (8)                     G.add_edge(3, 4, weight=5)
47: (8)                     assert edges_equal(
48: (12)                        nx.max_weight_matching(G), matching_dict_to_set({2: 3, 3: 2})
49: (8)                     )
50: (8)                     assert edges_equal(
51: (12)                        nx.max_weight_matching(G, 1), matching_dict_to_set({1: 2, 2: 1, 3:
4, 4: 3})
52: (8)                     )
53: (8)                     assert edges_equal(
54: (12)                        nx.min_weight_matching(G), matching_dict_to_set({1: 2, 3: 4})
55: (8)                     )
56: (8)                     assert edges_equal(
57: (12)                        nx.min_weight_matching(G, 1), matching_dict_to_set({1: 2, 3: 4})
```

```
58: (8)                     )
59: (4)                 def test_square(self):
60: (8)                     G = nx.Graph()
61: (8)                     G.add_edge(1, 4, weight=2)
62: (8)                     G.add_edge(2, 3, weight=2)
63: (8)                     G.add_edge(1, 2, weight=1)
64: (8)                     G.add_edge(3, 4, weight=4)
65: (8)                     assert edges_equal(
66: (12)                        nx.max_weight_matching(G), matching_dict_to_set({1: 2, 3: 4})
67: (8)                     )
68: (8)                     assert edges_equal(
69: (12)                        nx.min_weight_matching(G), matching_dict_to_set({1: 4, 2: 3})
70: (8)                     )
71: (4)                 def test_edge_attribute_name(self):
72: (8)                     G = nx.Graph()
73: (8)                     G.add_edge("one", "two", weight=10, abcd=11)
74: (8)                     G.add_edge("two", "three", weight=11, abcd=10)
75: (8)                     assert edges_equal(
76: (12)                        nx.max_weight_matching(G, weight="abcd"),
77: (12)                        matching_dict_to_set({"one": "two", "two": "one"}),
78: (8)                     )
79: (8)                     assert edges_equal(
80: (12)                        nx.min_weight_matching(G, weight="abcd"),
81: (12)                        matching_dict_to_set({"three": "two"}),
82: (8)                     )
83: (4)                 def test_floating_point_weights(self):
84: (8)                     G = nx.Graph()
85: (8)                     G.add_edge(1, 2, weight=math.pi)
86: (8)                     G.add_edge(2, 3, weight=math.exp(1))
87: (8)                     G.add_edge(1, 3, weight=3.0)
88: (8)                     G.add_edge(1, 4, weight=math.sqrt(2.0))
89: (8)                     assert edges_equal(
90: (12)                        nx.max_weight_matching(G), matching_dict_to_set({1: 4, 2: 3, 3: 2,
4: 1})
91: (8)                     )
92: (8)                     assert edges_equal(
93: (12)                        nx.min_weight_matching(G), matching_dict_to_set({1: 4, 2: 3, 3: 2,
4: 1})
94: (8)                     )
95: (4)                 def test_negative_weights(self):
96: (8)                     G = nx.Graph()
97: (8)                     G.add_edge(1, 2, weight=2)
98: (8)                     G.add_edge(1, 3, weight=-2)
99: (8)                     G.add_edge(2, 3, weight=1)
100: (8)                    G.add_edge(2, 4, weight=-1)
101: (8)                    G.add_edge(3, 4, weight=-6)
102: (8)                    assert edges_equal(
103: (12)                       nx.max_weight_matching(G), matching_dict_to_set({1: 2, 2: 1})
104: (8)                    )
105: (8)                    assert edges_equal(
106: (12)                       nx.max_weight_matching(G, maxcardinality=True),
107: (12)                       matching_dict_to_set({1: 3, 2: 4, 3: 1, 4: 2}),
108: (8)                    )
109: (8)                    assert edges_equal(
110: (12)                       nx.min_weight_matching(G), matching_dict_to_set({1: 2, 3: 4})
111: (8)                    )
112: (4)                def test_s_blossom(self):
113: (8)                    """Create S-blossom and use it for augmentation:"""
114: (8)                    G = nx.Graph()
115: (8)                    G.add_weighted_edges_from([(1, 2, 8), (1, 3, 9), (2, 3, 10), (3, 4,
7)])
116: (8)                    answer = matching_dict_to_set({1: 2, 2: 1, 3: 4, 4: 3})
117: (8)                    assert edges_equal(nx.max_weight_matching(G), answer)
118: (8)                    assert edges_equal(nx.min_weight_matching(G), answer)
119: (8)                    G.add_weighted_edges_from([(1, 6, 5), (4, 5, 6)])
120: (8)                    answer = matching_dict_to_set({1: 6, 2: 3, 3: 2, 4: 5, 5: 4, 6: 1})
121: (8)                    assert edges_equal(nx.max_weight_matching(G), answer)
122: (8)                    assert edges_equal(nx.min_weight_matching(G), answer)
123: (4)                def test_s_t_blossom(self):
```

```
124: (8)                    """Create S-blossom, relabel as T-blossom, use for augmentation:"""
125: (8)                    G = nx.Graph()
126: (8)                    G.add_weighted_edges_from(
127: (12)                       [(1, 2, 9), (1, 3, 8), (2, 3, 10), (1, 4, 5), (4, 5, 4), (1, 6,
3)]
128: (8)                    )
129: (8)                    answer = matching_dict_to_set({1: 6, 2: 3, 3: 2, 4: 5, 5: 4, 6: 1})
130: (8)                    assert edges_equal(nx.max_weight_matching(G), answer)
131: (8)                    assert edges_equal(nx.min_weight_matching(G), answer)
132: (8)                    G.add_edge(4, 5, weight=3)
133: (8)                    G.add_edge(1, 6, weight=4)
134: (8)                    assert edges_equal(nx.max_weight_matching(G), answer)
135: (8)                    assert edges_equal(nx.min_weight_matching(G), answer)
136: (8)                    G.remove_edge(1, 6)
137: (8)                    G.add_edge(3, 6, weight=4)
138: (8)                    answer = matching_dict_to_set({1: 2, 2: 1, 3: 6, 4: 5, 5: 4, 6: 3})
139: (8)                    assert edges_equal(nx.max_weight_matching(G), answer)
140: (8)                    assert edges_equal(nx.min_weight_matching(G), answer)
141: (4)                def test_nested_s_blossom(self):
142: (8)                    """Create nested S-blossom, use for augmentation:"""
143: (8)                    G = nx.Graph()
144: (8)                    G.add_weighted_edges_from(
145: (12)                       [
146: (16)                           (1, 2, 9),
147: (16)                           (1, 3, 9),
148: (16)                           (2, 3, 10),
149: (16)                           (2, 4, 8),
150: (16)                           (3, 5, 8),
151: (16)                           (4, 5, 10),
152: (16)                           (5, 6, 6),
153: (12)                       ]
154: (8)                    )
155: (8)                    dict_format = {1: 3, 2: 4, 3: 1, 4: 2, 5: 6, 6: 5}
156: (8)                    expected = {frozenset(e) for e in matching_dict_to_set(dict_format)}
157: (8)                    answer = {frozenset(e) for e in nx.max_weight_matching(G)}
158: (8)                    assert answer == expected
159: (8)                    answer = {frozenset(e) for e in nx.min_weight_matching(G)}
160: (8)                    assert answer == expected
161: (4)                def test_nested_s_blossom_relabel(self):
162: (8)                    """Create S-blossom, relabel as S, include in nested S-blossom:"""
163: (8)                    G = nx.Graph()
164: (8)                    G.add_weighted_edges_from(
165: (12)                       [
166: (16)                           (1, 2, 10),
167: (16)                           (1, 7, 10),
168: (16)                           (2, 3, 12),
169: (16)                           (3, 4, 20),
170: (16)                           (3, 5, 20),
171: (16)                           (4, 5, 25),
172: (16)                           (5, 6, 10),
173: (16)                           (6, 7, 10),
174: (16)                           (7, 8, 8),
175: (12)                       ]
176: (8)                    )
177: (8)                    answer = matching_dict_to_set({1: 2, 2: 1, 3: 4, 4: 3, 5: 6, 6: 5, 7:
8, 8: 7})
178: (8)                    assert edges_equal(nx.max_weight_matching(G), answer)
179: (8)                    assert edges_equal(nx.min_weight_matching(G), answer)
180: (4)                def test_nested_s_blossom_expand(self):
181: (8)                    """Create nested S-blossom, augment, expand recursively:"""
182: (8)                    G = nx.Graph()
183: (8)                    G.add_weighted_edges_from(
184: (12)                       [
185: (16)                           (1, 2, 8),
186: (16)                           (1, 3, 8),
187: (16)                           (2, 3, 10),
188: (16)                           (2, 4, 12),
189: (16)                           (3, 5, 12),
190: (16)                           (4, 5, 14),
```

```
(16)                            (4, 6, 12),
(16)                            (5, 7, 12),
(16)                            (6, 7, 14),
(16)                            (7, 8, 12),
(12)                        ]
(8)                     )
(8)                     answer = matching_dict_to_set({1: 2, 2: 1, 3: 5, 4: 6, 5: 3, 6: 4, 7:
8, 8: 7})
(8)                     assert edges_equal(nx.max_weight_matching(G), answer)
(8)                     assert edges_equal(nx.min_weight_matching(G), answer)
(4)                 def test_s_blossom_relabel_expand(self):
(8)                     """Create S-blossom, relabel as T, expand:"""
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            (1, 2, 23),
(16)                            (1, 5, 22),
(16)                            (1, 6, 15),
(16)                            (2, 3, 25),
(16)                            (3, 4, 22),
(16)                            (4, 5, 25),
(16)                            (4, 8, 14),
(16)                            (5, 7, 13),
(12)                        ]
(8)                     )
(8)                     answer = matching_dict_to_set({1: 6, 2: 3, 3: 2, 4: 8, 5: 7, 6: 1, 7:
5, 8: 4})
(8)                     assert edges_equal(nx.max_weight_matching(G), answer)
(8)                     assert edges_equal(nx.min_weight_matching(G), answer)
(4)                 def test_nested_s_blossom_relabel_expand(self):
(8)                     """Create nested S-blossom, relabel as T, expand:"""
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            (1, 2, 19),
(16)                            (1, 3, 20),
(16)                            (1, 8, 8),
(16)                            (2, 3, 25),
(16)                            (2, 4, 18),
(16)                            (3, 5, 18),
(16)                            (4, 5, 13),
(16)                            (4, 7, 7),
(16)                            (5, 6, 7),
(12)                        ]
(8)                     )
(8)                     answer = matching_dict_to_set({1: 8, 2: 3, 3: 2, 4: 7, 5: 6, 6: 5, 7:
4, 8: 1})
(8)                     assert edges_equal(nx.max_weight_matching(G), answer)
(8)                     assert edges_equal(nx.min_weight_matching(G), answer)
(4)                 def test_nasty_blossom1(self):
(8)                     """Create blossom, relabel as T in more than one way, expand,
(8)                     augment:
(8)                     """
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            (1, 2, 45),
(16)                            (1, 5, 45),
(16)                            (2, 3, 50),
(16)                            (3, 4, 45),
(16)                            (4, 5, 50),
(16)                            (1, 6, 30),
(16)                            (3, 9, 35),
(16)                            (4, 8, 35),
(16)                            (5, 7, 26),
(16)                            (9, 10, 5),
(12)                        ]
(8)                     )
(8)                     ansdict = {1: 6, 2: 3, 3: 2, 4: 8, 5: 7, 6: 1, 7: 5, 8: 4, 9: 10, 10:
```

```
9}
(8)                 answer = matching_dict_to_set(ansdict)
(8)                 assert edges_equal(nx.max_weight_matching(G), answer)
(8)                 assert edges_equal(nx.min_weight_matching(G), answer)
(4)             def test_nasty_blossom2(self):
(8)                 """Again but slightly different:"""
(8)                 G = nx.Graph()
(8)                 G.add_weighted_edges_from(
(12)                    [
(16)                        (1, 2, 45),
(16)                        (1, 5, 45),
(16)                        (2, 3, 50),
(16)                        (3, 4, 45),
(16)                        (4, 5, 50),
(16)                        (1, 6, 30),
(16)                        (3, 9, 35),
(16)                        (4, 8, 26),
(16)                        (5, 7, 40),
(16)                        (9, 10, 5),
(12)                    ]
(8)                 )
(8)                 ans = {1: 6, 2: 3, 3: 2, 4: 8, 5: 7, 6: 1, 7: 5, 8: 4, 9: 10, 10: 9}
(8)                 answer = matching_dict_to_set(ans)
(8)                 assert edges_equal(nx.max_weight_matching(G), answer)
(8)                 assert edges_equal(nx.min_weight_matching(G), answer)
(4)             def test_nasty_blossom_least_slack(self):
(8)                 """Create blossom, relabel as T, expand such that a new
(8)                 least-slack S-to-free dge is produced, augment:
(8)                 """
(8)                 G = nx.Graph()
(8)                 G.add_weighted_edges_from(
(12)                    [
(16)                        (1, 2, 45),
(16)                        (1, 5, 45),
(16)                        (2, 3, 50),
(16)                        (3, 4, 45),
(16)                        (4, 5, 50),
(16)                        (1, 6, 30),
(16)                        (3, 9, 35),
(16)                        (4, 8, 28),
(16)                        (5, 7, 26),
(16)                        (9, 10, 5),
(12)                    ]
(8)                 )
(8)                 ans = {1: 6, 2: 3, 3: 2, 4: 8, 5: 7, 6: 1, 7: 5, 8: 4, 9: 10, 10: 9}
(8)                 answer = matching_dict_to_set(ans)
(8)                 assert edges_equal(nx.max_weight_matching(G), answer)
(8)                 assert edges_equal(nx.min_weight_matching(G), answer)
(4)             def test_nasty_blossom_augmenting(self):
(8)                 """Create nested blossom, relabel as T in more than one way"""
(8)                 G = nx.Graph()
(8)                 G.add_weighted_edges_from(
(12)                    [
(16)                        (1, 2, 45),
(16)                        (1, 7, 45),
(16)                        (2, 3, 50),
(16)                        (3, 4, 45),
(16)                        (4, 5, 95),
(16)                        (4, 6, 94),
(16)                        (5, 6, 94),
(16)                        (6, 7, 50),
(16)                        (1, 8, 30),
(16)                        (3, 11, 35),
(16)                        (5, 9, 36),
(16)                        (7, 10, 26),
(16)                        (11, 12, 5),
(12)                    ]
(8)                 )
(8)                 ans = {
```

```
(12)                        1: 8,
(12)                        2: 3,
(12)                        3: 2,
(12)                        4: 6,
(12)                        5: 9,
(12)                        6: 4,
(12)                        7: 10,
(12)                        8: 1,
(12)                        9: 5,
(12)                        10: 7,
(12)                        11: 12,
(12)                        12: 11,
(8)                     }
(8)                     answer = matching_dict_to_set(ans)
(8)                     assert edges_equal(nx.max_weight_matching(G), answer)
(8)                     assert edges_equal(nx.min_weight_matching(G), answer)
(4)                 def test_nasty_blossom_expand_recursively(self):
(8)                     """Create nested S-blossom, relabel as S, expand recursively:"""
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            (1, 2, 40),
(16)                            (1, 3, 40),
(16)                            (2, 3, 60),
(16)                            (2, 4, 55),
(16)                            (3, 5, 55),
(16)                            (4, 5, 50),
(16)                            (1, 8, 15),
(16)                            (5, 7, 30),
(16)                            (7, 6, 10),
(16)                            (8, 10, 10),
(16)                            (4, 9, 30),
(12)                        ]
(8)                     )
(8)                     ans = {1: 2, 2: 1, 3: 5, 4: 9, 5: 3, 6: 7, 7: 6, 8: 10, 9: 4, 10: 8}
(8)                     answer = matching_dict_to_set(ans)
(8)                     assert edges_equal(nx.max_weight_matching(G), answer)
(8)                     assert edges_equal(nx.min_weight_matching(G), answer)
(4)                 def test_wrong_graph_type(self):
(8)                     error = nx.NetworkXNotImplemented
(8)                     raises(error, nx.max_weight_matching, nx.MultiGraph())
(8)                     raises(error, nx.max_weight_matching, nx.MultiDiGraph())
(8)                     raises(error, nx.max_weight_matching, nx.DiGraph())
(8)                     raises(error, nx.min_weight_matching, nx.DiGraph())
(0)             class TestIsMatching:
(4)                 """Unit tests for the
(4)                 :func:`~networkx.algorithms.matching.is_matching` function.
(4)                 """
(4)                 def test_dict(self):
(8)                     G = nx.path_graph(4)
(8)                     assert nx.is_matching(G, {0: 1, 1: 0, 2: 3, 3: 2})
(4)                 def test_empty_matching(self):
(8)                     G = nx.path_graph(4)
(8)                     assert nx.is_matching(G, set())
(4)                 def test_single_edge(self):
(8)                     G = nx.path_graph(4)
(8)                     assert nx.is_matching(G, {(1, 2)})
(4)                 def test_edge_order(self):
(8)                     G = nx.path_graph(4)
(8)                     assert nx.is_matching(G, {(0, 1), (2, 3)})
(8)                     assert nx.is_matching(G, {(1, 0), (2, 3)})
(8)                     assert nx.is_matching(G, {(0, 1), (3, 2)})
(8)                     assert nx.is_matching(G, {(1, 0), (3, 2)})
(4)                 def test_valid_matching(self):
(8)                     G = nx.path_graph(4)
(8)                     assert nx.is_matching(G, {(0, 1), (2, 3)})
(4)                 def test_invalid_input(self):
(8)                     error = nx.NetworkXError
(8)                     G = nx.path_graph(4)
```

```
(8)                 raises(error, nx.is_matching, G, {(0, 5), (2, 3)})
(8)                 raises(error, nx.is_matching, G, {(0, 1, 2), (2, 3)})
(8)                 raises(error, nx.is_matching, G, {(0,), (2, 3)})
(4)             def test_selfloops(self):
(8)                 error = nx.NetworkXError
(8)                 G = nx.path_graph(4)
(8)                 raises(error, nx.is_matching, G, {(5, 5), (2, 3)})
(8)                 assert not nx.is_matching(G, {(0, 0), (1, 2), (2, 3)})
(8)                 G.add_edge(0, 0)
(8)                 assert not nx.is_matching(G, {(0, 0), (1, 2)})
(4)             def test_invalid_matching(self):
(8)                 G = nx.path_graph(4)
(8)                 assert not nx.is_matching(G, {(0, 1), (1, 2), (2, 3)})
(4)             def test_invalid_edge(self):
(8)                 G = nx.path_graph(4)
(8)                 assert not nx.is_matching(G, {(0, 3), (1, 2)})
(8)                 raises(nx.NetworkXError, nx.is_matching, G, {(0, 55)})
(8)                 G = nx.DiGraph(G.edges)
(8)                 assert nx.is_matching(G, {(0, 1)})
(8)                 assert not nx.is_matching(G, {(1, 0)})
(0)         class TestIsMaximalMatching:
(4)             """Unit tests for the
(4)             :func:`~networkx.algorithms.matching.is_maximal_matching` function.
(4)             """
(4)             def test_dict(self):
(8)                 G = nx.path_graph(4)
(8)                 assert nx.is_maximal_matching(G, {0: 1, 1: 0, 2: 3, 3: 2})
(4)             def test_invalid_input(self):
(8)                 error = nx.NetworkXError
(8)                 G = nx.path_graph(4)
(8)                 raises(error, nx.is_maximal_matching, G, {(0, 5)})
(8)                 raises(error, nx.is_maximal_matching, G, {(5, 0)})
(8)                 raises(error, nx.is_maximal_matching, G, {(0, 1, 2), (2, 3)})
(8)                 raises(error, nx.is_maximal_matching, G, {(0,), (2, 3)})
(4)             def test_valid(self):
(8)                 G = nx.path_graph(4)
(8)                 assert nx.is_maximal_matching(G, {(0, 1), (2, 3)})
(4)             def test_not_matching(self):
(8)                 G = nx.path_graph(4)
(8)                 assert not nx.is_maximal_matching(G, {(0, 1), (1, 2), (2, 3)})
(8)                 assert not nx.is_maximal_matching(G, {(0, 3)})
(8)                 G.add_edge(0, 0)
(8)                 assert not nx.is_maximal_matching(G, {(0, 0)})
(4)             def test_not_maximal(self):
(8)                 G = nx.path_graph(4)
(8)                 assert not nx.is_maximal_matching(G, {(0, 1)})
(0)         class TestIsPerfectMatching:
(4)             """Unit tests for the
(4)             :func:`~networkx.algorithms.matching.is_perfect_matching` function.
(4)             """
(4)             def test_dict(self):
(8)                 G = nx.path_graph(4)
(8)                 assert nx.is_perfect_matching(G, {0: 1, 1: 0, 2: 3, 3: 2})
(4)             def test_valid(self):
(8)                 G = nx.path_graph(4)
(8)                 assert nx.is_perfect_matching(G, {(0, 1), (2, 3)})
(4)             def test_valid_not_path(self):
(8)                 G = nx.cycle_graph(4)
(8)                 G.add_edge(0, 4)
(8)                 G.add_edge(1, 4)
(8)                 G.add_edge(5, 2)
(8)                 assert nx.is_perfect_matching(G, {(1, 4), (0, 3), (5, 2)})
(4)             def test_invalid_input(self):
(8)                 error = nx.NetworkXError
(8)                 G = nx.path_graph(4)
(8)                 raises(error, nx.is_perfect_matching, G, {(0, 5)})
(8)                 raises(error, nx.is_perfect_matching, G, {(5, 0)})
(8)                 raises(error, nx.is_perfect_matching, G, {(0, 1, 2), (2, 3)})
(8)                 raises(error, nx.is_perfect_matching, G, {(0,), (2, 3)})
```

```
463: (4)                def test_selfloops(self):
464: (8)                    error = nx.NetworkXError
465: (8)                    G = nx.path_graph(4)
466: (8)                    raises(error, nx.is_perfect_matching, G, {(5, 5), (2, 3)})
467: (8)                    assert not nx.is_perfect_matching(G, {(0, 0), (1, 2), (2, 3)})
468: (8)                    G.add_edge(0, 0)
469: (8)                    assert not nx.is_perfect_matching(G, {(0, 0), (1, 2)})
470: (4)                def test_not_matching(self):
471: (8)                    G = nx.path_graph(4)
472: (8)                    assert not nx.is_perfect_matching(G, {(0, 3)})
473: (8)                    assert not nx.is_perfect_matching(G, {(0, 1), (1, 2), (2, 3)})
474: (4)                def test_maximal_but_not_perfect(self):
475: (8)                    G = nx.cycle_graph(4)
476: (8)                    G.add_edge(0, 4)
477: (8)                    G.add_edge(1, 4)
478: (8)                    assert not nx.is_perfect_matching(G, {(1, 4), (0, 3)})
479: (0)            class TestMaximalMatching:
480: (4)                """Unit tests for the
481: (4)                :func:`~networkx.algorithms.matching.maximal_matching`.
482: (4)                """
483: (4)                def test_valid_matching(self):
484: (8)                    edges = [(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (3, 6), (5, 6)]
485: (8)                    G = nx.Graph(edges)
486: (8)                    matching = nx.maximal_matching(G)
487: (8)                    assert nx.is_maximal_matching(G, matching)
488: (4)                def test_single_edge_matching(self):
489: (8)                    G = nx.star_graph(5)
490: (8)                    matching = nx.maximal_matching(G)
491: (8)                    assert 1 == len(matching)
492: (8)                    assert nx.is_maximal_matching(G, matching)
493: (4)                def test_self_loops(self):
494: (8)                    G = nx.path_graph(3)
495: (8)                    G.add_edges_from([(0, 0), (1, 1)])
496: (8)                    matching = nx.maximal_matching(G)
497: (8)                    assert len(matching) == 1
498: (8)                    assert not any(u == v for u, v in matching)
499: (8)                    assert nx.is_maximal_matching(G, matching)
500: (4)                def test_ordering(self):
501: (8)                    """Tests that a maximal matching is computed correctly
502: (8)                    regardless of the order in which nodes are added to the graph.
503: (8)                    """
504: (8)                    for nodes in permutations(range(3)):
505: (12)                       G = nx.Graph()
506: (12)                       G.add_nodes_from(nodes)
507: (12)                       G.add_edges_from([(0, 1), (0, 2)])
508: (12)                       matching = nx.maximal_matching(G)
509: (12)                       assert len(matching) == 1
510: (12)                       assert nx.is_maximal_matching(G, matching)
511: (4)                def test_wrong_graph_type(self):
512: (8)                    error = nx.NetworkXNotImplemented
513: (8)                    raises(error, nx.maximal_matching, nx.MultiGraph())
514: (8)                    raises(error, nx.maximal_matching, nx.MultiDiGraph())
515: (8)                    raises(error, nx.maximal_matching, nx.DiGraph())
```

File 335 - test_mis.py:

```
1: (0)              """
2: (0)              Tests for maximal (not maximum) independent sets.
3: (0)              """
4: (0)              import random
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              def test_random_seed():
8: (4)                  G = nx.empty_graph(5)
9: (4)                  assert nx.maximal_independent_set(G, seed=1) == [1, 0, 3, 2, 4]
10: (0)             @pytest.mark.parametrize("graph", [nx.complete_graph(5),
nx.complete_graph(55)])
```

```
11: (0)             def test_K5(graph):
12: (4)                 """Maximal independent set for complete graphs"""
13: (4)                 assert all(nx.maximal_independent_set(graph, [n]) == [n] for n in graph)
14: (0)             def test_exceptions():
15: (4)                 """Bad input should raise exception."""
16: (4)                 G = nx.florentine_families_graph()
17: (4)                 pytest.raises(nx.NetworkXUnfeasible, nx.maximal_independent_set, G,
["Smith"])
18: (4)                 pytest.raises(
19: (8)                     nx.NetworkXUnfeasible, nx.maximal_independent_set, G, ["Salviati",
"Pazzi"]
20: (4)                 )
21: (4)                 pytest.raises(nx.NetworkXNotImplemented, nx.maximal_independent_set,
nx.DiGraph(G))
22: (0)             def test_florentine_family():
23: (4)                 G = nx.florentine_families_graph()
24: (4)                 indep = nx.maximal_independent_set(G, ["Medici", "Bischeri"])
25: (4)                 assert set(indep) == {
26: (8)                     "Medici",
27: (8)                     "Bischeri",
28: (8)                     "Castellani",
29: (8)                     "Pazzi",
30: (8)                     "Ginori",
31: (8)                     "Lamberteschi",
32: (4)                 }
33: (0)             def test_bipartite():
34: (4)                 G = nx.complete_bipartite_graph(12, 34)
35: (4)                 indep = nx.maximal_independent_set(G, [4, 5, 9, 10])
36: (4)                 assert sorted(indep) == list(range(12))
37: (0)             def test_random_graphs():
38: (4)                 """Generate 5 random graphs of different types and sizes and
39: (4)                 make sure that all sets are independent and maximal."""
40: (4)                 for i in range(0, 50, 10):
41: (8)                     G = nx.erdos_renyi_graph(i * 10 + 1, random.random())
42: (8)                     IS = nx.maximal_independent_set(G)
43: (8)                     assert G.subgraph(IS).number_of_edges() == 0
44: (8)                     nbrs_of_MIS = set.union(*(set(G.neighbors(v)) for v in IS))
45: (8)                     assert all(v in nbrs_of_MIS for v in set(G.nodes()).difference(IS))

----------------------------------------

File 336 - test_max_weight_clique.py:

1: (0)              """Maximum weight clique test suite.
2: (0)              """
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              class TestMaximumWeightClique:
6: (4)                  def test_basic_cases(self):
7: (8)                      def check_basic_case(graph_func, expected_weight, weight_accessor):
8: (12)                         graph = graph_func()
9: (12)                         clique, weight = nx.algorithms.max_weight_clique(graph,
weight_accessor)
10: (12)                        assert verify_clique(
11: (16)                            graph, clique, weight, expected_weight, weight_accessor
12: (12)                        )
13: (8)                     for graph_func, (expected_weight, expected_size) in
TEST_CASES.items():
14: (12)                        check_basic_case(graph_func, expected_weight, "weight")
15: (12)                        check_basic_case(graph_func, expected_size, None)
16: (4)                 def test_key_error(self):
17: (8)                     graph = two_node_graph()
18: (8)                     with pytest.raises(KeyError):
19: (12)                        nx.algorithms.max_weight_clique(graph, "nonexistent-key")
20: (4)                 def test_error_on_non_integer_weight(self):
21: (8)                     graph = two_node_graph()
22: (8)                     graph.nodes[2]["weight"] = 1.5
23: (8)                     with pytest.raises(ValueError):
24: (12)                        nx.algorithms.max_weight_clique(graph)
```

```
 25: (4)                 def test_unaffected_by_self_loops(self):
 26: (8)                     graph = two_node_graph()
 27: (8)                     graph.add_edge(1, 1)
 28: (8)                     graph.add_edge(2, 2)
 29: (8)                     clique, weight = nx.algorithms.max_weight_clique(graph, "weight")
 30: (8)                     assert verify_clique(graph, clique, weight, 30, "weight")
 31: (8)                     graph = three_node_independent_set()
 32: (8)                     graph.add_edge(1, 1)
 33: (8)                     clique, weight = nx.algorithms.max_weight_clique(graph, "weight")
 34: (8)                     assert verify_clique(graph, clique, weight, 20, "weight")
 35: (4)                 def test_30_node_prob(self):
 36: (8)                     G = nx.Graph()
 37: (8)                     G.add_nodes_from(range(1, 31))
 38: (8)                     for i in range(1, 31):
 39: (12)                        G.nodes[i]["weight"] = i + 1
 40: (8)                     G.add_edges_from(
 41: (12)                        [
 42: (16)                            (1, 12), (1, 13), (1, 15), (1, 16), (1, 18), (1, 19), (1, 20),
 43: (16)                            (1, 23), (1, 26), (1, 28), (1, 29), (1, 30), (2, 3), (2, 4),
 44: (16)                            (2, 5), (2, 8), (2, 9), (2, 10), (2, 14), (2, 17), (2, 18),
 45: (16)                            (2, 21), (2, 22), (2, 23), (2, 27), (3, 9), (3, 15), (3, 21),
 46: (16)                            (3, 22), (3, 23), (3, 24), (3, 27), (3, 28), (3, 29), (4, 5),
 47: (16)                            (4, 6), (4, 8), (4, 21), (4, 22), (4, 23), (4, 26), (4, 28),
 48: (16)                            (4, 30), (5, 6), (5, 8), (5, 9), (5, 13), (5, 14), (5, 15),
 49: (16)                            (5, 16), (5, 20), (5, 21), (5, 22), (5, 25), (5, 28), (5, 29),
 50: (16)                            (6, 7), (6, 8), (6, 13), (6, 17), (6, 18), (6, 19), (6, 24),
 51: (16)                            (6, 26), (6, 27), (6, 28), (6, 29), (7, 12), (7, 14), (7, 15),
 52: (16)                            (7, 16), (7, 17), (7, 20), (7, 25), (7, 27), (7, 29), (7, 30),
 53: (16)                            (8, 10), (8, 15), (8, 16), (8, 18), (8, 20), (8, 22), (8, 24),
 54: (16)                            (8, 26), (8, 27), (8, 28), (8, 30), (9, 11), (9, 12), (9, 13),
 55: (16)                            (9, 14), (9, 15), (9, 16), (9, 19), (9, 20), (9, 21), (9, 24),
 56: (16)                            (9, 30), (10, 12), (10, 15), (10, 18), (10, 19), (10, 20),
 57: (16)                            (10, 22), (10, 23), (10, 24), (10, 26), (10, 27), (10, 29),
 58: (16)                            (10, 30), (11, 13), (11, 15), (11, 16), (11, 17), (11, 18),
 59: (16)                            (11, 19), (11, 20), (11, 22), (11, 29), (11, 30), (12, 14),
 60: (16)                            (12, 17), (12, 18), (12, 19), (12, 20), (12, 21), (12, 23),
 61: (16)                            (12, 25), (12, 26), (12, 30), (13, 20), (13, 22), (13, 23),
 62: (16)                            (13, 24), (13, 30), (14, 16), (14, 20), (14, 21), (14, 22),
 63: (16)                            (14, 23), (14, 25), (14, 26), (14, 27), (14, 29), (14, 30),
 64: (16)                            (15, 17), (15, 18), (15, 20), (15, 21), (15, 26), (15, 27),
 65: (16)                            (15, 28), (16, 17), (16, 18), (16, 19), (16, 20), (16, 21),
 66: (16)                            (16, 29), (16, 30), (17, 18), (17, 21), (17, 22), (17, 25),
 67: (16)                            (17, 27), (17, 28), (17, 30), (18, 19), (18, 20), (18, 21),
 68: (16)                            (18, 22), (18, 23), (18, 24), (19, 20), (19, 22), (19, 23),
 69: (16)                            (19, 24), (19, 25), (19, 27), (19, 30), (20, 21), (20, 23),
 70: (16)                            (20, 24), (20, 26), (20, 28), (20, 29), (21, 23), (21, 26),
 71: (16)                            (21, 27), (21, 29), (22, 24), (22, 25), (22, 26), (22, 29),
 72: (16)                            (23, 25), (23, 30), (24, 25), (24, 26), (25, 27), (25, 29),
 73: (16)                            (26, 27), (26, 28), (26, 30), (28, 29), (29, 30),
 74: (12)                        ]
 75: (8)                     )
 76: (8)                     clique, weight = nx.algorithms.max_weight_clique(G)
 77: (8)                     assert verify_clique(G, clique, weight, 111, "weight")
 78: (0)             def verify_clique(
 79: (4)                 graph, clique, reported_clique_weight, expected_clique_weight,
weight_accessor
 80: (0)             ):
 81: (4)                 for node1 in clique:
 82: (8)                     for node2 in clique:
 83: (12)                        if node1 == node2:
 84: (16)                            continue
 85: (12)                        if not graph.has_edge(node1, node2):
 86: (16)                            return False
 87: (4)                 if weight_accessor is None:
 88: (8)                     clique_weight = len(clique)
 89: (4)                 else:
 90: (8)                     clique_weight = sum(graph.nodes[v]["weight"] for v in clique)
 91: (4)                 if clique_weight != expected_clique_weight:
 92: (8)                     return False
```

```
93: (4)                 if clique_weight != reported_clique_weight:
94: (8)                     return False
95: (4)                 return True
96: (0)             def empty_graph():
97: (4)                 return nx.Graph()
98: (0)             def one_node_graph():
99: (4)                 graph = nx.Graph()
100: (4)                graph.add_nodes_from([1])
101: (4)                graph.nodes[1]["weight"] = 10
102: (4)                return graph
103: (0)            def two_node_graph():
104: (4)                graph = nx.Graph()
105: (4)                graph.add_nodes_from([1, 2])
106: (4)                graph.add_edges_from([(1, 2)])
107: (4)                graph.nodes[1]["weight"] = 10
108: (4)                graph.nodes[2]["weight"] = 20
109: (4)                return graph
110: (0)            def three_node_clique():
111: (4)                graph = nx.Graph()
112: (4)                graph.add_nodes_from([1, 2, 3])
113: (4)                graph.add_edges_from([(1, 2), (1, 3), (2, 3)])
114: (4)                graph.nodes[1]["weight"] = 10
115: (4)                graph.nodes[2]["weight"] = 20
116: (4)                graph.nodes[3]["weight"] = 5
117: (4)                return graph
118: (0)            def three_node_independent_set():
119: (4)                graph = nx.Graph()
120: (4)                graph.add_nodes_from([1, 2, 3])
121: (4)                graph.nodes[1]["weight"] = 10
122: (4)                graph.nodes[2]["weight"] = 20
123: (4)                graph.nodes[3]["weight"] = 5
124: (4)                return graph
125: (0)            def disconnected():
126: (4)                graph = nx.Graph()
127: (4)                graph.add_edges_from([(1, 2), (2, 3), (4, 5), (5, 6)])
128: (4)                graph.nodes[1]["weight"] = 10
129: (4)                graph.nodes[2]["weight"] = 20
130: (4)                graph.nodes[3]["weight"] = 5
131: (4)                graph.nodes[4]["weight"] = 100
132: (4)                graph.nodes[5]["weight"] = 200
133: (4)                graph.nodes[6]["weight"] = 50
134: (4)                return graph
135: (0)            TEST_CASES = {
136: (4)                empty_graph: (0, 0),
137: (4)                one_node_graph: (10, 1),
138: (4)                two_node_graph: (30, 2),
139: (4)                three_node_clique: (35, 3),
140: (4)                three_node_independent_set: (20, 1),
141: (4)                disconnected: (300, 2),
142: (0)            }

----------------------------------------

File 337 - test_moral.py:

1: (0)              import networkx as nx
2: (0)              from networkx.algorithms.moral import moral_graph
3: (0)              def test_get_moral_graph():
4: (4)                  graph = nx.DiGraph()
5: (4)                  graph.add_nodes_from([1, 2, 3, 4, 5, 6, 7])
6: (4)                  graph.add_edges_from([(1, 2), (3, 2), (4, 1), (4, 5), (6, 5), (7, 5)])
7: (4)                  H = moral_graph(graph)
8: (4)                  assert not H.is_directed()
9: (4)                  assert H.has_edge(1, 3)
10: (4)                 assert H.has_edge(4, 6)
11: (4)                 assert H.has_edge(6, 7)
12: (4)                 assert H.has_edge(4, 7)
13: (4)                 assert not H.has_edge(1, 5)
```

```
----------------------------------------

File 338 - test_node_classification.py:

1: (0)              import pytest
2: (0)              pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms import node_classification
6: (0)              class TestHarmonicFunction:
7: (4)                  def test_path_graph(self):
8: (8)                      G = nx.path_graph(4)
9: (8)                      label_name = "label"
10: (8)                     G.nodes[0][label_name] = "A"
11: (8)                     G.nodes[3][label_name] = "B"
12: (8)                     predicted = node_classification.harmonic_function(G,
label_name=label_name)
13: (8)                     assert predicted[0] == "A"
14: (8)                     assert predicted[1] == "A"
15: (8)                     assert predicted[2] == "B"
16: (8)                     assert predicted[3] == "B"
17: (4)                 def test_no_labels(self):
18: (8)                     with pytest.raises(nx.NetworkXError):
19: (12)                        G = nx.path_graph(4)
20: (12)                        node_classification.harmonic_function(G)
21: (4)                 def test_no_nodes(self):
22: (8)                     with pytest.raises(nx.NetworkXError):
23: (12)                        G = nx.Graph()
24: (12)                        node_classification.harmonic_function(G)
25: (4)                 def test_no_edges(self):
26: (8)                     with pytest.raises(nx.NetworkXError):
27: (12)                        G = nx.Graph()
28: (12)                        G.add_node(1)
29: (12)                        G.add_node(2)
30: (12)                        node_classification.harmonic_function(G)
31: (4)                 def test_digraph(self):
32: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
33: (12)                        G = nx.DiGraph()
34: (12)                        G.add_edge(0, 1)
35: (12)                        G.add_edge(1, 2)
36: (12)                        G.add_edge(2, 3)
37: (12)                        label_name = "label"
38: (12)                        G.nodes[0][label_name] = "A"
39: (12)                        G.nodes[3][label_name] = "B"
40: (12)                        node_classification.harmonic_function(G)
41: (4)                 def test_one_labeled_node(self):
42: (8)                     G = nx.path_graph(4)
43: (8)                     label_name = "label"
44: (8)                     G.nodes[0][label_name] = "A"
45: (8)                     predicted = node_classification.harmonic_function(G,
label_name=label_name)
46: (8)                     assert predicted[0] == "A"
47: (8)                     assert predicted[1] == "A"
48: (8)                     assert predicted[2] == "A"
49: (8)                     assert predicted[3] == "A"
50: (4)                 def test_nodes_all_labeled(self):
51: (8)                     G = nx.karate_club_graph()
52: (8)                     label_name = "club"
53: (8)                     predicted = node_classification.harmonic_function(G,
label_name=label_name)
54: (8)                     for i in range(len(G)):
55: (12)                        assert predicted[i] == G.nodes[i][label_name]
56: (4)                 def test_labeled_nodes_are_not_changed(self):
57: (8)                     G = nx.karate_club_graph()
58: (8)                     label_name = "club"
59: (8)                     label_removed = {0, 1, 2, 3, 4, 5, 6, 7}
60: (8)                     for i in label_removed:
61: (12)                        del G.nodes[i][label_name]
62: (8)                     predicted = node_classification.harmonic_function(G,
```

```
label_name=label_name)
63: (8)                     label_not_removed = set(range(len(G))) - label_removed
64: (8)                     for i in label_not_removed:
65: (12)                        assert predicted[i] == G.nodes[i][label_name]
66: (0)             class TestLocalAndGlobalConsistency:
67: (4)                 def test_path_graph(self):
68: (8)                     G = nx.path_graph(4)
69: (8)                     label_name = "label"
70: (8)                     G.nodes[0][label_name] = "A"
71: (8)                     G.nodes[3][label_name] = "B"
72: (8)                     predicted = node_classification.local_and_global_consistency(
73: (12)                        G, label_name=label_name
74: (8)                     )
75: (8)                     assert predicted[0] == "A"
76: (8)                     assert predicted[1] == "A"
77: (8)                     assert predicted[2] == "B"
78: (8)                     assert predicted[3] == "B"
79: (4)                 def test_no_labels(self):
80: (8)                     with pytest.raises(nx.NetworkXError):
81: (12)                        G = nx.path_graph(4)
82: (12)                        node_classification.local_and_global_consistency(G)
83: (4)                 def test_no_nodes(self):
84: (8)                     with pytest.raises(nx.NetworkXError):
85: (12)                        G = nx.Graph()
86: (12)                        node_classification.local_and_global_consistency(G)
87: (4)                 def test_no_edges(self):
88: (8)                     with pytest.raises(nx.NetworkXError):
89: (12)                        G = nx.Graph()
90: (12)                        G.add_node(1)
91: (12)                        G.add_node(2)
92: (12)                        node_classification.local_and_global_consistency(G)
93: (4)                 def test_digraph(self):
94: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
95: (12)                        G = nx.DiGraph()
96: (12)                        G.add_edge(0, 1)
97: (12)                        G.add_edge(1, 2)
98: (12)                        G.add_edge(2, 3)
99: (12)                        label_name = "label"
100: (12)                       G.nodes[0][label_name] = "A"
101: (12)                       G.nodes[3][label_name] = "B"
102: (12)                       node_classification.harmonic_function(G)
103: (4)                def test_one_labeled_node(self):
104: (8)                    G = nx.path_graph(4)
105: (8)                    label_name = "label"
106: (8)                    G.nodes[0][label_name] = "A"
107: (8)                    predicted = node_classification.local_and_global_consistency(
108: (12)                       G, label_name=label_name
109: (8)                    )
110: (8)                    assert predicted[0] == "A"
111: (8)                    assert predicted[1] == "A"
112: (8)                    assert predicted[2] == "A"
113: (8)                    assert predicted[3] == "A"
114: (4)                def test_nodes_all_labeled(self):
115: (8)                    G = nx.karate_club_graph()
116: (8)                    label_name = "club"
117: (8)                    predicted = node_classification.local_and_global_consistency(
118: (12)                       G, alpha=0, label_name=label_name
119: (8)                    )
120: (8)                    for i in range(len(G)):
121: (12)                       assert predicted[i] == G.nodes[i][label_name]

----------------------------------------

File 339 - test_non_randomness.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              np = pytest.importorskip("numpy")
4: (0)              @pytest.mark.parametrize(
```

```
5: (4)                  "k, weight, expected",
6: (4)                  [
7: (8)                      (None, None, 7.21),  # infers 3 communities
8: (8)                      (2, None, 11.7),
9: (8)                      (None, "weight", 25.45),
10: (8)                     (2, "weight", 38.8),
11: (4)                 ],
12: (0)             )
13: (0)             def test_non_randomness(k, weight, expected):
14: (4)                 G = nx.karate_club_graph()
15: (4)                 np.testing.assert_almost_equal(
16: (8)                     nx.non_randomness(G, k, weight)[0], expected, decimal=2
17: (4)                 )
18: (0)             def test_non_connected():
19: (4)                 G = nx.Graph()
20: (4)                 G.add_edge(1, 2)
21: (4)                 G.add_node(3)
22: (4)                 with pytest.raises(nx.NetworkXException):
23: (8)                     nx.non_randomness(G)
24: (0)             def test_self_loops():
25: (4)                 G = nx.Graph()
26: (4)                 G.add_edge(1, 2)
27: (4)                 G.add_edge(1, 1)
28: (4)                 with pytest.raises(nx.NetworkXError):
29: (8)                     nx.non_randomness(G)
```

File 340 - test_planar_drawing.py:

```
1: (0)              import math
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms.planar_drawing import triangulate_embedding
5: (0)              def test_graph1():
6: (4)                  embedding_data = {0: [1, 2, 3], 1: [2, 0], 2: [3, 0, 1], 3: [2, 0]}
7: (4)                  check_embedding_data(embedding_data)
8: (0)              def test_graph2():
9: (4)                  embedding_data = {
10: (8)                     0: [8, 6],
11: (8)                     1: [2, 6, 9],
12: (8)                     2: [8, 1, 7, 9, 6, 4],
13: (8)                     3: [9],
14: (8)                     4: [2],
15: (8)                     5: [6, 8],
16: (8)                     6: [9, 1, 0, 5, 2],
17: (8)                     7: [9, 2],
18: (8)                     8: [0, 2, 5],
19: (8)                     9: [1, 6, 2, 7, 3],
20: (4)                 }
21: (4)                 check_embedding_data(embedding_data)
22: (0)             def test_circle_graph():
23: (4)                 embedding_data = {
24: (8)                     0: [1, 9],
25: (8)                     1: [0, 2],
26: (8)                     2: [1, 3],
27: (8)                     3: [2, 4],
28: (8)                     4: [3, 5],
29: (8)                     5: [4, 6],
30: (8)                     6: [5, 7],
31: (8)                     7: [6, 8],
32: (8)                     8: [7, 9],
33: (8)                     9: [8, 0],
34: (4)                 }
35: (4)                 check_embedding_data(embedding_data)
36: (0)             def test_grid_graph():
37: (4)                 embedding_data = {
38: (8)                     (0, 1): [(0, 0), (1, 1), (0, 2)],
39: (8)                     (1, 2): [(1, 1), (2, 2), (0, 2)],
```

```
40: (8)                     (0, 0): [(0, 1), (1, 0)],
41: (8)                     (2, 1): [(2, 0), (2, 2), (1, 1)],
42: (8)                     (1, 1): [(2, 1), (1, 2), (0, 1), (1, 0)],
43: (8)                     (2, 0): [(1, 0), (2, 1)],
44: (8)                     (2, 2): [(1, 2), (2, 1)],
45: (8)                     (1, 0): [(0, 0), (2, 0), (1, 1)],
46: (8)                     (0, 2): [(1, 2), (0, 1)],
47: (4)                 }
48: (4)                 check_embedding_data(embedding_data)
49: (0)             def test_one_node_graph():
50: (4)                 embedding_data = {0: []}
51: (4)                 check_embedding_data(embedding_data)
52: (0)             def test_two_node_graph():
53: (4)                 embedding_data = {0: [1], 1: [0]}
54: (4)                 check_embedding_data(embedding_data)
55: (0)             def test_three_node_graph():
56: (4)                 embedding_data = {0: [1, 2], 1: [0, 2], 2: [0, 1]}
57: (4)                 check_embedding_data(embedding_data)
58: (0)             def test_multiple_component_graph1():
59: (4)                 embedding_data = {0: [], 1: []}
60: (4)                 check_embedding_data(embedding_data)
61: (0)             def test_multiple_component_graph2():
62: (4)                 embedding_data = {0: [1, 2], 1: [0, 2], 2: [0, 1], 3: [4, 5], 4: [3, 5],
5: [3, 4]}
63: (4)                 check_embedding_data(embedding_data)
64: (0)             def test_invalid_half_edge():
65: (4)                 with pytest.raises(nx.NetworkXException):
66: (8)                     embedding_data = {1: [2, 3, 4], 2: [1, 3, 4], 3: [1, 2, 4], 4: [1, 2,
3]}
67: (8)                     embedding = nx.PlanarEmbedding()
68: (8)                     embedding.set_data(embedding_data)
69: (8)                     nx.combinatorial_embedding_to_pos(embedding)
70: (0)             def test_triangulate_embedding1():
71: (4)                 embedding = nx.PlanarEmbedding()
72: (4)                 embedding.add_node(1)
73: (4)                 expected_embedding = {1: []}
74: (4)                 check_triangulation(embedding, expected_embedding)
75: (0)             def test_triangulate_embedding2():
76: (4)                 embedding = nx.PlanarEmbedding()
77: (4)                 embedding.connect_components(1, 2)
78: (4)                 expected_embedding = {1: [2], 2: [1]}
79: (4)                 check_triangulation(embedding, expected_embedding)
80: (0)             def check_triangulation(embedding, expected_embedding):
81: (4)                 res_embedding, _ = triangulate_embedding(embedding, True)
82: (4)                 assert (
83: (8)                     res_embedding.get_data() == expected_embedding
84: (4)                 ), "Expected embedding incorrect"
85: (4)                 res_embedding, _ = triangulate_embedding(embedding, False)
86: (4)                 assert (
87: (8)                     res_embedding.get_data() == expected_embedding
88: (4)                 ), "Expected embedding incorrect"
89: (0)             def check_embedding_data(embedding_data):
90: (4)                 """Checks that the planar embedding of the input is correct"""
91: (4)                 embedding = nx.PlanarEmbedding()
92: (4)                 embedding.set_data(embedding_data)
93: (4)                 pos_fully = nx.combinatorial_embedding_to_pos(embedding, False)
94: (4)                 msg = "Planar drawing does not conform to the embedding (fully
triangulation)"
95: (4)                 assert planar_drawing_conforms_to_embedding(embedding, pos_fully), msg
96: (4)                 check_edge_intersections(embedding, pos_fully)
97: (4)                 pos_internally = nx.combinatorial_embedding_to_pos(embedding, True)
98: (4)                 msg = "Planar drawing does not conform to the embedding (internal
triangulation)"
99: (4)                 assert planar_drawing_conforms_to_embedding(embedding, pos_internally),
msg
100: (4)                check_edge_intersections(embedding, pos_internally)
101: (0)            def is_close(a, b, rel_tol=1e-09, abs_tol=0.0):
102: (4)                return abs(a - b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
103: (0)            def point_in_between(a, b, p):
```

```
104: (4)                x1, y1 = a
105: (4)                x2, y2 = b
106: (4)                px, py = p
107: (4)                dist_1_2 = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
108: (4)                dist_1_p = math.sqrt((x1 - px) ** 2 + (y1 - py) ** 2)
109: (4)                dist_2_p = math.sqrt((x2 - px) ** 2 + (y2 - py) ** 2)
110: (4)                return is_close(dist_1_p + dist_2_p, dist_1_2)
111: (0)            def check_edge_intersections(G, pos):
112: (4)                """Check all edges in G for intersections.
113: (4)                Raises an exception if an intersection is found.
114: (4)                Parameters
115: (4)                ----------
116: (4)                G : NetworkX graph
117: (4)                pos : dict
118: (8)                    Maps every node to a tuple (x, y) representing its position
119: (4)                """
120: (4)                for a, b in G.edges():
121: (8)                    for c, d in G.edges():
122: (12)                       if a != c and b != d and b != c and a != d:
123: (16)                           x1, y1 = pos[a]
124: (16)                           x2, y2 = pos[b]
125: (16)                           x3, y3 = pos[c]
126: (16)                           x4, y4 = pos[d]
127: (16)                           determinant = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
128: (16)                           if determinant != 0:  # the lines are not parallel
129: (20)                               px = (x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (
130: (24)                                   x3 * y4 - y3 * x4
131: (20)                               ) / determinant
132: (20)                               py = (x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (
133: (24)                                   x3 * y4 - y3 * x4
134: (20)                               ) / determinant
135: (20)                               if point_in_between(pos[a], pos[b], (px, py)) and
point_in_between(
136: (24)                                   pos[c], pos[d], (px, py)
137: (20)                               ):
138: (24)                                   msg = f"There is an intersection at {px},{py}"
139: (24)                                   raise nx.NetworkXException(msg)
140: (16)                           msg = "A node lies on a edge connecting two other nodes"
141: (16)                           if (
142: (20)                               point_in_between(pos[a], pos[b], pos[c])
143: (20)                               or point_in_between(pos[a], pos[b], pos[d])
144: (20)                               or point_in_between(pos[c], pos[d], pos[a])
145: (20)                               or point_in_between(pos[c], pos[d], pos[b])
146: (16)                           ):
147: (20)                               raise nx.NetworkXException(msg)
148: (0)            class Vector:
149: (4)                """Compare vectors by their angle without loss of precision
150: (4)                All vectors in direction [0, 1] are the smallest.
151: (4)                The vectors grow in clockwise direction.
152: (4)                """
153: (4)                __slots__ = ["x", "y", "node", "quadrant"]
154: (4)                def __init__(self, x, y, node):
155: (8)                    self.x = x
156: (8)                    self.y = y
157: (8)                    self.node = node
158: (8)                    if self.x >= 0 and self.y > 0:
159: (12)                       self.quadrant = 1
160: (8)                    elif self.x > 0 and self.y <= 0:
161: (12)                       self.quadrant = 2
162: (8)                    elif self.x <= 0 and self.y < 0:
163: (12)                       self.quadrant = 3
164: (8)                    else:
165: (12)                       self.quadrant = 4
166: (4)                def __eq__(self, other):
167: (8)                    return self.quadrant == other.quadrant and self.x * other.y == self.y
* other.x
168: (4)                def __lt__(self, other):
169: (8)                    if self.quadrant < other.quadrant:
170: (12)                       return True
```

```
171: (8)                    elif self.quadrant > other.quadrant:
172: (12)                       return False
173: (8)                    else:
174: (12)                       return self.x * other.y < self.y * other.x
175: (4)                def __ne__(self, other):
176: (8)                    return self != other
177: (4)                def __le__(self, other):
178: (8)                    return not other < self
179: (4)                def __gt__(self, other):
180: (8)                    return other < self
181: (4)                def __ge__(self, other):
182: (8)                    return not self < other
183: (0)            def planar_drawing_conforms_to_embedding(embedding, pos):
184: (4)                """Checks if pos conforms to the planar embedding
185: (4)                Returns true iff the neighbors are actually oriented in the orientation
186: (4)                specified of the embedding
187: (4)                """
188: (4)                for v in embedding:
189: (8)                    nbr_vectors = []
190: (8)                    v_pos = pos[v]
191: (8)                    for nbr in embedding[v]:
192: (12)                       new_vector = Vector(pos[nbr][0] - v_pos[0], pos[nbr][1] -
v_pos[1], nbr)
193: (12)                       nbr_vectors.append(new_vector)
194: (8)                    nbr_vectors.sort()
195: (8)                    for idx, nbr_vector in enumerate(nbr_vectors):
196: (12)                       cw_vector = nbr_vectors[(idx + 1) % len(nbr_vectors)]
197: (12)                       ccw_vector = nbr_vectors[idx - 1]
198: (12)                       if (
199: (16)                           embedding[v][nbr_vector.node]["cw"] != cw_vector.node
200: (16)                           or embedding[v][nbr_vector.node]["ccw"] != ccw_vector.node
201: (12)                       ):
202: (16)                           return False
203: (12)                       if cw_vector.node != nbr_vector.node and cw_vector == nbr_vector:
204: (16)                           return False
205: (12)                       if ccw_vector.node != nbr_vector.node and ccw_vector ==
nbr_vector:
206: (16)                           return False
207: (4)                return True
```

File 341 - test_planarity.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.planarity import (
4: (4)                  check_planarity_recursive,
5: (4)                  get_counterexample,
6: (4)                  get_counterexample_recursive,
7: (0)              )
8: (0)              class TestLRPlanarity:
9: (4)                  """Nose Unit tests for the :mod:`networkx.algorithms.planarity` module.
10: (4)                 Tests three things:
11: (4)                 1. Check that the result is correct
12: (8)                     (returns planar if and only if the graph is actually planar)
13: (4)                 2. In case a counter example is returned: Check if it is correct
14: (4)                 3. In case an embedding is returned: Check if its actually an embedding
15: (4)                 """
16: (4)                 @staticmethod
17: (4)                 def check_graph(G, is_planar=None):
18: (8)                     """Raises an exception if the lr_planarity check returns a wrong
result
19: (8)                     Parameters
20: (8)                     ----------
21: (8)                     G : NetworkX graph
22: (8)                     is_planar : bool
23: (12)                        The expected result of the planarity check.
24: (12)                        If set to None only counter example or embedding are verified.
```

```
(8)         """
(8)         is_planar_lr, result = nx.check_planarity(G, True)
(8)         is_planar_lr_rec, result_rec = check_planarity_recursive(G, True)
(8)         if is_planar is not None:
(12)            if is_planar:
(16)                msg = "Wrong planarity check result. Should be planar."
(12)            else:
(16)                msg = "Wrong planarity check result. Should be non-planar."
(12)            assert is_planar == is_planar_lr, msg
(12)            assert is_planar == is_planar_lr_rec, msg
(8)         if is_planar_lr:
(12)            check_embedding(G, result)
(12)            check_embedding(G, result_rec)
(8)         else:
(12)            check_counterexample(G, result)
(12)            check_counterexample(G, result_rec)
(4)     def test_simple_planar_graph(self):
(8)         e = [
(12)            (1, 2),
(12)            (2, 3),
(12)            (3, 4),
(12)            (4, 6),
(12)            (6, 7),
(12)            (7, 1),
(12)            (1, 5),
(12)            (5, 2),
(12)            (2, 4),
(12)            (4, 5),
(12)            (5, 7),
(8)         ]
(8)         self.check_graph(nx.Graph(e), is_planar=True)
(4)     def test_planar_with_selfloop(self):
(8)         e = [
(12)            (1, 1),
(12)            (2, 2),
(12)            (3, 3),
(12)            (4, 4),
(12)            (5, 5),
(12)            (1, 2),
(12)            (1, 3),
(12)            (1, 5),
(12)            (2, 5),
(12)            (2, 4),
(12)            (3, 4),
(12)            (3, 5),
(12)            (4, 5),
(8)         ]
(8)         self.check_graph(nx.Graph(e), is_planar=True)
(4)     def test_k3_3(self):
(8)         self.check_graph(nx.complete_bipartite_graph(3, 3), is_planar=False)
(4)     def test_k5(self):
(8)         self.check_graph(nx.complete_graph(5), is_planar=False)
(4)     def test_multiple_components_planar(self):
(8)         e = [(1, 2), (2, 3), (3, 1), (4, 5), (5, 6), (6, 4)]
(8)         self.check_graph(nx.Graph(e), is_planar=True)
(4)     def test_multiple_components_non_planar(self):
(8)         G = nx.complete_graph(5)
(8)         G.add_edges_from([(6, 7), (7, 8), (8, 6)])
(8)         self.check_graph(G, is_planar=False)
(4)     def test_non_planar_with_selfloop(self):
(8)         G = nx.complete_graph(5)
(8)         for i in range(5):
(12)            G.add_edge(i, i)
(8)         self.check_graph(G, is_planar=False)
(4)     def test_non_planar1(self):
(8)         e = [
(12)            (1, 5),
(12)            (1, 6),
(12)            (1, 7),
```

```
(12)                        (2, 6),
(12)                        (2, 3),
(12)                        (3, 5),
(12)                        (3, 7),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (4, 7),
(8)                     ]
(8)                     self.check_graph(nx.Graph(e), is_planar=False)
(4)                 def test_loop(self):
(8)                     e = [(1, 2), (2, 2)]
(8)                     G = nx.Graph(e)
(8)                     self.check_graph(G, is_planar=True)
(4)                 def test_comp(self):
(8)                     e = [(1, 2), (3, 4)]
(8)                     G = nx.Graph(e)
(8)                     G.remove_edge(1, 2)
(8)                     self.check_graph(G, is_planar=True)
(4)                 def test_goldner_harary(self):
(8)                     e = [
(12)                        (1, 2),
(12)                        (1, 3),
(12)                        (1, 4),
(12)                        (1, 5),
(12)                        (1, 7),
(12)                        (1, 8),
(12)                        (1, 10),
(12)                        (1, 11),
(12)                        (2, 3),
(12)                        (2, 4),
(12)                        (2, 6),
(12)                        (2, 7),
(12)                        (2, 9),
(12)                        (2, 10),
(12)                        (2, 11),
(12)                        (3, 4),
(12)                        (4, 5),
(12)                        (4, 6),
(12)                        (4, 7),
(12)                        (5, 7),
(12)                        (6, 7),
(12)                        (7, 8),
(12)                        (7, 9),
(12)                        (7, 10),
(12)                        (8, 10),
(12)                        (9, 10),
(12)                        (10, 11),
(8)                     ]
(8)                     G = nx.Graph(e)
(8)                     self.check_graph(G, is_planar=True)
(4)                 def test_planar_multigraph(self):
(8)                     G = nx.MultiGraph([(1, 2), (1, 2), (1, 2), (1, 2), (2, 3), (3, 1)])
(8)                     self.check_graph(G, is_planar=True)
(4)                 def test_non_planar_multigraph(self):
(8)                     G = nx.MultiGraph(nx.complete_graph(5))
(8)                     G.add_edges_from([(1, 2)] * 5)
(8)                     self.check_graph(G, is_planar=False)
(4)                 def test_planar_digraph(self):
(8)                     G = nx.DiGraph([(1, 2), (2, 3), (2, 4), (4, 1), (4, 2), (1, 4), (3,
2)])
(8)                     self.check_graph(G, is_planar=True)
(4)                 def test_non_planar_digraph(self):
(8)                     G = nx.DiGraph(nx.complete_graph(5))
(8)                     G.remove_edge(1, 2)
(8)                     G.remove_edge(4, 1)
(8)                     self.check_graph(G, is_planar=False)
(4)                 def test_single_component(self):
(8)                     G = nx.Graph()
(8)                     G.add_node(1)
```

```
(8)                 self.check_graph(G, is_planar=True)
(4)             def test_graph1(self):
(8)                 G = nx.Graph(
(12)                    [
(16)                        (3, 10),
(16)                        (2, 13),
(16)                        (1, 13),
(16)                        (7, 11),
(16)                        (0, 8),
(16)                        (8, 13),
(16)                        (0, 2),
(16)                        (0, 7),
(16)                        (0, 10),
(16)                        (1, 7),
(12)                    ]
(8)                 )
(8)                 self.check_graph(G, is_planar=True)
(4)             def test_graph2(self):
(8)                 G = nx.Graph(
(12)                    [
(16)                        (1, 2),
(16)                        (4, 13),
(16)                        (0, 13),
(16)                        (4, 5),
(16)                        (7, 10),
(16)                        (1, 7),
(16)                        (0, 3),
(16)                        (2, 6),
(16)                        (5, 6),
(16)                        (7, 13),
(16)                        (4, 8),
(16)                        (0, 8),
(16)                        (0, 9),
(16)                        (2, 13),
(16)                        (6, 7),
(16)                        (3, 6),
(16)                        (2, 8),
(12)                    ]
(8)                 )
(8)                 self.check_graph(G, is_planar=False)
(4)             def test_graph3(self):
(8)                 G = nx.Graph(
(12)                    [
(16)                        (0, 7),
(16)                        (3, 11),
(16)                        (3, 4),
(16)                        (8, 9),
(16)                        (4, 11),
(16)                        (1, 7),
(16)                        (1, 13),
(16)                        (1, 11),
(16)                        (3, 5),
(16)                        (5, 7),
(16)                        (1, 3),
(16)                        (0, 4),
(16)                        (5, 11),
(16)                        (5, 13),
(12)                    ]
(8)                 )
(8)                 self.check_graph(G, is_planar=False)
(4)             def test_counterexample_planar(self):
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    G = nx.Graph()
(12)                    G.add_node(1)
(12)                    get_counterexample(G)
(4)             def test_counterexample_planar_recursive(self):
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    G = nx.Graph()
(12)                    G.add_node(1)
```

```
(12)                        get_counterexample_recursive(G)
(4)                 def test_edge_removal_from_planar_embedding(self):
(8)                     edges = ((0, 1), (1, 2), (2, 3), (3, 4), (4, 0), (0, 2), (0, 3))
(8)                     G = nx.Graph(edges)
(8)                     cert, P = nx.check_planarity(G)
(8)                     assert cert is True
(8)                     P.remove_edge(0, 2)
(8)                     self.check_graph(P, is_planar=True)
(8)                     P.add_half_edge_ccw(1, 3, 2)
(8)                     P.add_half_edge_cw(3, 1, 2)
(8)                     self.check_graph(P, is_planar=True)
(8)                     P.remove_edges_from(((0, 3), (1, 3)))
(8)                     self.check_graph(P, is_planar=True)
(0)             def check_embedding(G, embedding):
(4)                 """Raises an exception if the combinatorial embedding is not correct
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 embedding : a dict mapping nodes to a list of edges
(8)                     This specifies the ordering of the outgoing edges from a node for
(8)                     a combinatorial embedding
(4)                 Notes
(4)                 -----
(4)                 Checks the following things:
(8)                     - The type of the embedding is correct
(8)                     - The nodes and edges match the original graph
(8)                     - Every half edge has its matching opposite half edge
(8)                     - No intersections of edges (checked by Euler's formula)
(4)                 """
(4)                 if not isinstance(embedding, nx.PlanarEmbedding):
(8)                     raise nx.NetworkXException("Bad embedding. Not of type
nx.PlanarEmbedding")
(4)                 embedding.check_structure()
(4)                 assert set(G.nodes) == set(
(8)                     embedding.nodes
(4)                 ), "Bad embedding. Nodes don't match the original graph."
(4)                 g_edges = set()
(4)                 for edge in G.edges:
(8)                     if edge[0] != edge[1]:
(12)                        g_edges.add((edge[0], edge[1]))
(12)                        g_edges.add((edge[1], edge[0]))
(4)                 assert g_edges == set(
(8)                     embedding.edges
(4)                 ), "Bad embedding. Edges don't match the original graph."
(0)             def check_counterexample(G, sub_graph):
(4)                 """Raises an exception if the counterexample is wrong.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 subdivision_nodes : set
(8)                     A set of nodes inducing a subgraph as a counterexample
(4)                 """
(4)                 sub_graph = nx.Graph(sub_graph)
(4)                 for u in sub_graph:
(8)                     if sub_graph.has_edge(u, u):
(12)                        sub_graph.remove_edge(u, u)
(4)                 contract = list(sub_graph)
(4)                 while len(contract) > 0:
(8)                     contract_node = contract.pop()
(8)                     if contract_node not in sub_graph:
(12)                        continue
(8)                     degree = sub_graph.degree[contract_node]
(8)                     if degree == 2:
(12)                        neighbors = iter(sub_graph[contract_node])
(12)                        u = next(neighbors)
(12)                        v = next(neighbors)
(12)                        contract.append(u)
(12)                        contract.append(v)
(12)                        sub_graph.remove_node(contract_node)
```

```
299: (12)                        sub_graph.add_edge(u, v)
300: (4)                 if len(sub_graph) == 5:
301: (8)                     if not nx.is_isomorphic(nx.complete_graph(5), sub_graph):
302: (12)                        raise nx.NetworkXException("Bad counter example.")
303: (4)                 elif len(sub_graph) == 6:
304: (8)                     if not nx.is_isomorphic(nx.complete_bipartite_graph(3, 3), sub_graph):
305: (12)                        raise nx.NetworkXException("Bad counter example.")
306: (4)                 else:
307: (8)                     raise nx.NetworkXException("Bad counter example.")
308: (0)             class TestPlanarEmbeddingClass:
309: (4)                 def test_add_half_edge(self):
310: (8)                     embedding = nx.PlanarEmbedding()
311: (8)                     embedding.add_half_edge(0, 1)
312: (8)                     with pytest.raises(
313: (12)                        nx.NetworkXException, match="Invalid clockwise reference node."
314: (8)                     ):
315: (12)                        embedding.add_half_edge(0, 2, cw=3)
316: (8)                     with pytest.raises(
317: (12)                        nx.NetworkXException, match="Invalid counterclockwise reference
node."
318: (8)                     ):
319: (12)                        embedding.add_half_edge(0, 2, ccw=3)
320: (8)                     with pytest.raises(
321: (12)                        nx.NetworkXException, match="Only one of cw/ccw can be specified."
322: (8)                     ):
323: (12)                        embedding.add_half_edge(0, 2, cw=1, ccw=1)
324: (8)                     with pytest.raises(
325: (12)                        nx.NetworkXException,
326: (12)                        match=(
327: (16)                            r"Node already has out-half-edge\(s\), either"
328: (16)                            " cw or ccw reference node required."
329: (12)                        ),
330: (8)                     ):
331: (12)                        embedding.add_half_edge(0, 2)
332: (8)                     embedding.add_half_edge(0, 2, cw=1)
333: (8)                     embedding.add_half_edge(0, 3, ccw=1)
334: (8)                     assert sorted(embedding.edges(data=True)) == [
335: (12)                        (0, 1, {"ccw": 2, "cw": 3}),
336: (12)                        (0, 2, {"cw": 1, "ccw": 3}),
337: (12)                        (0, 3, {"cw": 2, "ccw": 1}),
338: (8)                     ]
339: (4)                 def test_get_data(self):
340: (8)                     embedding = self.get_star_embedding(4)
341: (8)                     data = embedding.get_data()
342: (8)                     data_cmp = {0: [3, 2, 1], 1: [0], 2: [0], 3: [0]}
343: (8)                     assert data == data_cmp
344: (4)                 def test_edge_removal(self):
345: (8)                     embedding = nx.PlanarEmbedding()
346: (8)                     embedding.set_data(
347: (12)                        {
348: (16)                            1: [2, 5, 7],
349: (16)                            2: [1, 3, 4, 5],
350: (16)                            3: [2, 4],
351: (16)                            4: [3, 6, 5, 2],
352: (16)                            5: [7, 1, 2, 4],
353: (16)                            6: [4, 7],
354: (16)                            7: [6, 1, 5],
355: (12)                        }
356: (8)                     )
357: (8)                     embedding.remove_edges_from(((5, 4), (1, 5)))
358: (8)                     embedding.check_structure()
359: (8)                     embedding_expected = nx.PlanarEmbedding()
360: (8)                     embedding_expected.set_data(
361: (12)                        {
362: (16)                            1: [2, 7],
363: (16)                            2: [1, 3, 4, 5],
364: (16)                            3: [2, 4],
365: (16)                            4: [3, 6, 2],
366: (16)                            5: [7, 2],
```

```
(16)                        6: [4, 7],
(16)                        7: [6, 1, 5],
(12)                    }
(8)                 )
(8)                 assert nx.utils.graphs_equal(embedding, embedding_expected)
(4)             def test_missing_edge_orientation(self):
(8)                 embedding = nx.PlanarEmbedding({1: {2: {}}, 2: {1: {}}})
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    embedding.check_structure()
(4)             def test_invalid_edge_orientation(self):
(8)                 embedding = nx.PlanarEmbedding(
(12)                    {
(16)                        1: {2: {"cw": 2, "ccw": 2}},
(16)                        2: {1: {"cw": 1, "ccw": 1}},
(16)                        1: {3: {}},
(16)                        3: {1: {}},
(12)                    }
(8)                 )
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    embedding.check_structure()
(4)             def test_missing_half_edge(self):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 embedding.add_half_edge(1, 2)
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    embedding.check_structure()
(4)             def test_not_fulfilling_euler_formula(self):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 for i in range(5):
(12)                    ref = None
(12)                    for j in range(5):
(16)                        if i != j:
(20)                            embedding.add_half_edge(i, j, cw=ref)
(20)                            ref = j
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    embedding.check_structure()
(4)             def test_missing_reference(self):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 with pytest.raises(nx.NetworkXException, match="Invalid reference
node."):
(12)                    embedding.add_half_edge(1, 2, ccw=3)
(4)             def test_connect_components(self):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 embedding.connect_components(1, 2)
(4)             def test_successful_face_traversal(self):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 embedding.add_half_edge(1, 2)
(8)                 embedding.add_half_edge(2, 1)
(8)                 face = embedding.traverse_face(1, 2)
(8)                 assert face == [1, 2]
(4)             def test_unsuccessful_face_traversal(self):
(8)                 embedding = nx.PlanarEmbedding(
(12)                    {1: {2: {"cw": 3, "ccw": 2}}, 2: {1: {"cw": 3, "ccw": 1}}}
(8)                 )
(8)                 with pytest.raises(nx.NetworkXException):
(12)                    embedding.traverse_face(1, 2)
(4)             def test_forbidden_methods(self):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 embedding.add_node(42)  # no exception
(8)                 embedding.add_nodes_from([(23, 24)])  # no exception
(8)                 with pytest.raises(NotImplementedError):
(12)                    embedding.add_edge(1, 3)
(8)                 with pytest.raises(NotImplementedError):
(12)                    embedding.add_edges_from([(0, 2), (1, 4)])
(8)                 with pytest.raises(NotImplementedError):
(12)                    embedding.add_weighted_edges_from([(0, 2, 350), (1, 4, 125)])
(4)             @staticmethod
(4)             def get_star_embedding(n):
(8)                 embedding = nx.PlanarEmbedding()
(8)                 ref = None
```

```
435: (8)                    for i in range(1, n):
436: (12)                       embedding.add_half_edge(0, i, cw=ref)
437: (12)                       ref = i
438: (12)                       embedding.add_half_edge(i, 0)
439: (8)                    return embedding
```

File 342 - test_polynomials.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.polynomials` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              sympy = pytest.importorskip("sympy")
5: (0)              _test_tutte_graphs = {
6: (4)                  nx.complete_graph(1): "1",
7: (4)                  nx.complete_graph(4): "x**3 + 3*x**2 + 4*x*y + 2*x + y**3 + 3*y**2 + 2*y",
8: (4)                  nx.cycle_graph(5): "x**4 + x**3 + x**2 + x + y",
9: (4)                  nx.diamond_graph(): "x**3 + 2*x**2 + 2*x*y + x + y**2 + y",
10: (0)             }
11: (0)             _test_chromatic_graphs = {
12: (4)                 nx.complete_graph(1): "x",
13: (4)                 nx.complete_graph(4): "x**4 - 6*x**3 + 11*x**2 - 6*x",
14: (4)                 nx.cycle_graph(5): "x**5 - 5*x**4 + 10*x**3 - 10*x**2 + 4*x",
15: (4)                 nx.diamond_graph(): "x**4 - 5*x**3 + 8*x**2 - 4*x",
16: (4)                 nx.path_graph(5): "x**5 - 4*x**4 + 6*x**3 - 4*x**2 + x",
17: (0)             }
18: (0)             @pytest.mark.parametrize(("G", "expected"), _test_tutte_graphs.items())
19: (0)             def test_tutte_polynomial(G, expected):
20: (4)                 assert nx.tutte_polynomial(G).equals(expected)
21: (0)             @pytest.mark.parametrize("G", _test_tutte_graphs.keys())
22: (0)             def test_tutte_polynomial_disjoint(G):
23: (4)                 """Tutte polynomial factors into the Tutte polynomials of its components.
24: (4)                 Verify this property with the disjoint union of two copies of the input
graph.
25: (4)                 """
26: (4)                 t_g = nx.tutte_polynomial(G)
27: (4)                 H = nx.disjoint_union(G, G)
28: (4)                 t_h = nx.tutte_polynomial(H)
29: (4)                 assert sympy.simplify(t_g * t_g).equals(t_h)
30: (0)             @pytest.mark.parametrize(("G", "expected"), _test_chromatic_graphs.items())
31: (0)             def test_chromatic_polynomial(G, expected):
32: (4)                 assert nx.chromatic_polynomial(G).equals(expected)
33: (0)             @pytest.mark.parametrize("G", _test_chromatic_graphs.keys())
34: (0)             def test_chromatic_polynomial_disjoint(G):
35: (4)                 """Chromatic polynomial factors into the Chromatic polynomials of its
36: (4)                 components. Verify this property with the disjoint union of two copies of
37: (4)                 the input graph.
38: (4)                 """
39: (4)                 x_g = nx.chromatic_polynomial(G)
40: (4)                 H = nx.disjoint_union(G, G)
41: (4)                 x_h = nx.chromatic_polynomial(H)
42: (4)                 assert sympy.simplify(x_g * x_g).equals(x_h)
```

File 343 - test_reciprocity.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestReciprocity:
4: (4)                  def test_reciprocity_digraph(self):
5: (8)                      DG = nx.DiGraph([(1, 2), (2, 1)])
6: (8)                      reciprocity = nx.reciprocity(DG)
7: (8)                      assert reciprocity == 1.0
8: (4)                  def test_overall_reciprocity_empty_graph(self):
9: (8)                      with pytest.raises(nx.NetworkXError):
10: (12)                        DG = nx.DiGraph()
11: (12)                        nx.overall_reciprocity(DG)
```

```
12: (4)                 def test_reciprocity_graph_nodes(self):
13: (8)                     DG = nx.DiGraph([(1, 2), (2, 3), (3, 2)])
14: (8)                     reciprocity = nx.reciprocity(DG, [1, 2])
15: (8)                     expected_reciprocity = {1: 0.0, 2: 0.6666666666666666}
16: (8)                     assert reciprocity == expected_reciprocity
17: (4)                 def test_reciprocity_graph_node(self):
18: (8)                     DG = nx.DiGraph([(1, 2), (2, 3), (3, 2)])
19: (8)                     reciprocity = nx.reciprocity(DG, 2)
20: (8)                     assert reciprocity == 0.6666666666666666
21: (4)                 def test_reciprocity_graph_isolated_nodes(self):
22: (8)                     with pytest.raises(nx.NetworkXError):
23: (12)                        DG = nx.DiGraph([(1, 2)])
24: (12)                        DG.add_node(4)
25: (12)                        nx.reciprocity(DG, 4)
```

File 344 - test_regular.py:

```
1: (0)              import pytest
2: (0)              import networkx
3: (0)              import networkx as nx
4: (0)              import networkx.algorithms.regular as reg
5: (0)              import networkx.generators as gen
6: (0)              class TestKFactor:
7: (4)                  def test_k_factor_trivial(self):
8: (8)                      g = gen.cycle_graph(4)
9: (8)                      f = reg.k_factor(g, 2)
10: (8)                     assert g.edges == f.edges
11: (4)                 def test_k_factor1(self):
12: (8)                     g = gen.grid_2d_graph(4, 4)
13: (8)                     g_kf = reg.k_factor(g, 2)
14: (8)                     for edge in g_kf.edges():
15: (12)                        assert g.has_edge(edge[0], edge[1])
16: (8)                     for _, degree in g_kf.degree():
17: (12)                        assert degree == 2
18: (4)                 def test_k_factor2(self):
19: (8)                     g = gen.complete_graph(6)
20: (8)                     g_kf = reg.k_factor(g, 3)
21: (8)                     for edge in g_kf.edges():
22: (12)                        assert g.has_edge(edge[0], edge[1])
23: (8)                     for _, degree in g_kf.degree():
24: (12)                        assert degree == 3
25: (4)                 def test_k_factor3(self):
26: (8)                     g = gen.grid_2d_graph(4, 4)
27: (8)                     with pytest.raises(nx.NetworkXUnfeasible):
28: (12)                        reg.k_factor(g, 3)
29: (4)                 def test_k_factor4(self):
30: (8)                     g = gen.lattice.hexagonal_lattice_graph(4, 4)
31: (8)                     with pytest.raises(nx.NetworkXUnfeasible):
32: (12)                        reg.k_factor(g, 2)
33: (4)                 def test_k_factor5(self):
34: (8)                     g = gen.complete_graph(6)
35: (8)                     g_kf = reg.k_factor(g, 2)
36: (8)                     for edge in g_kf.edges():
37: (12)                        assert g.has_edge(edge[0], edge[1])
38: (8)                     for _, degree in g_kf.degree():
39: (12)                        assert degree == 2
40: (0)             class TestIsRegular:
41: (4)                 def test_is_regular1(self):
42: (8)                     g = gen.cycle_graph(4)
43: (8)                     assert reg.is_regular(g)
44: (4)                 def test_is_regular2(self):
45: (8)                     g = gen.complete_graph(5)
46: (8)                     assert reg.is_regular(g)
47: (4)                 def test_is_regular3(self):
48: (8)                     g = gen.lollipop_graph(5, 5)
49: (8)                     assert not reg.is_regular(g)
50: (4)                 def test_is_regular4(self):
```

```
51: (8)                     g = nx.DiGraph()
52: (8)                     g.add_edges_from([(0, 1), (1, 2), (2, 0)])
53: (8)                     assert reg.is_regular(g)
54: (0)             def test_is_regular_empty_graph_raises():
55: (4)                 G = nx.Graph()
56: (4)                 with pytest.raises(nx.NetworkXPointlessConcept, match="Graph has no
nodes"):
57: (8)                     nx.is_regular(G)
58: (0)             class TestIsKRegular:
59: (4)                 def test_is_k_regular1(self):
60: (8)                     g = gen.cycle_graph(4)
61: (8)                     assert reg.is_k_regular(g, 2)
62: (8)                     assert not reg.is_k_regular(g, 3)
63: (4)                 def test_is_k_regular2(self):
64: (8)                     g = gen.complete_graph(5)
65: (8)                     assert reg.is_k_regular(g, 4)
66: (8)                     assert not reg.is_k_regular(g, 3)
67: (8)                     assert not reg.is_k_regular(g, 6)
68: (4)                 def test_is_k_regular3(self):
69: (8)                     g = gen.lollipop_graph(5, 5)
70: (8)                     assert not reg.is_k_regular(g, 5)
71: (8)                     assert not reg.is_k_regular(g, 6)

----------------------------------------

File 345 - test_richclub.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_richclub():
4: (4)                  G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (4, 5)])
5: (4)                  rc = nx.richclub.rich_club_coefficient(G, normalized=False)
6: (4)                  assert rc == {0: 12.0 / 30, 1: 8.0 / 12}
7: (4)                  rc0 = nx.richclub.rich_club_coefficient(G, normalized=False)[0]
8: (4)                  assert rc0 == 12.0 / 30.0
9: (0)              def test_richclub_seed():
10: (4)                 G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (4, 5)])
11: (4)                 rcNorm = nx.richclub.rich_club_coefficient(G, Q=2, seed=1)
12: (4)                 assert rcNorm == {0: 1.0, 1: 1.0}
13: (0)             def test_richclub_normalized():
14: (4)                 G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (4, 5)])
15: (4)                 rcNorm = nx.richclub.rich_club_coefficient(G, Q=2, seed=42)
16: (4)                 assert rcNorm == {0: 1.0, 1: 1.0}
17: (0)             def test_richclub2():
18: (4)                 T = nx.balanced_tree(2, 10)
19: (4)                 rc = nx.richclub.rich_club_coefficient(T, normalized=False)
20: (4)                 assert rc == {
21: (8)                     0: 4092 / (2047 * 2046.0),
22: (8)                     1: (2044.0 / (1023 * 1022)),
23: (8)                     2: (2040.0 / (1022 * 1021)),
24: (4)                 }
25: (0)             def test_richclub3():
26: (4)                 G = nx.karate_club_graph()
27: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
28: (4)                 assert rc == {
29: (8)                     0: 156.0 / 1122,
30: (8)                     1: 154.0 / 1056,
31: (8)                     2: 110.0 / 462,
32: (8)                     3: 78.0 / 240,
33: (8)                     4: 44.0 / 90,
34: (8)                     5: 22.0 / 42,
35: (8)                     6: 10.0 / 20,
36: (8)                     7: 10.0 / 20,
37: (8)                     8: 10.0 / 20,
38: (8)                     9: 6.0 / 12,
39: (8)                     10: 2.0 / 6,
40: (8)                     11: 2.0 / 6,
41: (8)                     12: 0.0,
42: (8)                     13: 0.0,
```

```
43: (8)                     14: 0.0,
44: (8)                     15: 0.0,
45: (4)                 }
46: (0)             def test_richclub4():
47: (4)                 G = nx.Graph()
48: (4)                 G.add_edges_from(
49: (8)                     [(0, 1), (0, 2), (0, 3), (0, 4), (4, 5), (5, 9), (6, 9), (7, 9), (8,
9)]
50: (4)                 )
51: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
52: (4)                 assert rc == {0: 18 / 90.0, 1: 6 / 12.0, 2: 0.0, 3: 0.0}
53: (0)             def test_richclub_exception():
54: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
55: (8)                     G = nx.DiGraph()
56: (8)                     nx.rich_club_coefficient(G)
57: (0)             def test_rich_club_exception2():
58: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
59: (8)                     G = nx.MultiGraph()
60: (8)                     nx.rich_club_coefficient(G)
61: (0)             def test_rich_club_selfloop():
62: (4)                 G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
63: (4)                 G.add_edge(1, 1)  # self loop
64: (4)                 G.add_edge(1, 2)
65: (4)                 with pytest.raises(
66: (8)                     Exception,
67: (8)                     match="rich_club_coefficient is not implemented for " "graphs with
self loops.",
68: (4)                 ):
69: (8)                     nx.rich_club_coefficient(G)
70: (0)             def test_rich_club_leq_3_nodes_unnormalized():
71: (4)                 G = nx.Graph()
72: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
73: (4)                 assert rc == {}
74: (4)                 for i in range(3):
75: (8)                     G.add_node(i)
76: (8)                     rc = nx.rich_club_coefficient(G, normalized=False)
77: (8)                     assert rc == {}
78: (4)                 G = nx.Graph()
79: (4)                 G.add_edge(0, 1)
80: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
81: (4)                 assert rc == {0: 1}
82: (4)                 G = nx.Graph()
83: (4)                 G.add_nodes_from([0, 1, 2])
84: (4)                 G.add_edge(0, 1)
85: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
86: (4)                 assert rc == {0: 1}
87: (4)                 G.add_edge(1, 2)
88: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
89: (4)                 assert rc == {0: 2 / 3}
90: (4)                 G.add_edge(0, 2)
91: (4)                 rc = nx.rich_club_coefficient(G, normalized=False)
92: (4)                 assert rc == {0: 1, 1: 1}
93: (0)             def test_rich_club_leq_3_nodes_normalized():
94: (4)                 G = nx.Graph()
95: (4)                 with pytest.raises(
96: (8)                     nx.exception.NetworkXError,
97: (8)                     match="Graph has fewer than four nodes",
98: (4)                 ):
99: (8)                     rc = nx.rich_club_coefficient(G, normalized=True)
100: (4)                for i in range(3):
101: (8)                    G.add_node(i)
102: (8)                    with pytest.raises(
103: (12)                       nx.exception.NetworkXError,
104: (12)                       match="Graph has fewer than four nodes",
105: (8)                    ):
106: (12)                       rc = nx.rich_club_coefficient(G, normalized=True)

----------------------------------------
```

File 346 - test_similarity.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.similarity import (
4: (4)                  graph_edit_distance,
5: (4)                  optimal_edit_paths,
6: (4)                  optimize_graph_edit_distance,
7: (0)              )
8: (0)              from networkx.generators.classic import (
9: (4)                  circular_ladder_graph,
10: (4)                 cycle_graph,
11: (4)                 path_graph,
12: (4)                 wheel_graph,
13: (0)             )
14: (0)             def nmatch(n1, n2):
15: (4)                 return n1 == n2
16: (0)             def ematch(e1, e2):
17: (4)                 return e1 == e2
18: (0)             def getCanonical():
19: (4)                 G = nx.Graph()
20: (4)                 G.add_node("A", label="A")
21: (4)                 G.add_node("B", label="B")
22: (4)                 G.add_node("C", label="C")
23: (4)                 G.add_node("D", label="D")
24: (4)                 G.add_edge("A", "B", label="a-b")
25: (4)                 G.add_edge("B", "C", label="b-c")
26: (4)                 G.add_edge("B", "D", label="b-d")
27: (4)                 return G
28: (0)             class TestSimilarity:
29: (4)                 @classmethod
30: (4)                 def setup_class(cls):
31: (8)                     global np
32: (8)                     np = pytest.importorskip("numpy")
33: (8)                     pytest.importorskip("scipy")
34: (4)                 def test_graph_edit_distance_roots_and_timeout(self):
35: (8)                     G0 = nx.star_graph(5)
36: (8)                     G1 = G0.copy()
37: (8)                     pytest.raises(ValueError, graph_edit_distance, G0, G1, roots=[2])
38: (8)                     pytest.raises(ValueError, graph_edit_distance, G0, G1, roots=[2, 3,
4])
39: (8)                     pytest.raises(nx.NodeNotFound, graph_edit_distance, G0, G1, roots=(9,
3))
40: (8)                     pytest.raises(nx.NodeNotFound, graph_edit_distance, G0, G1, roots=(3,
9))
41: (8)                     pytest.raises(nx.NodeNotFound, graph_edit_distance, G0, G1, roots=(9,
9))
42: (8)                     assert graph_edit_distance(G0, G1, roots=(1, 2)) == 0
43: (8)                     assert graph_edit_distance(G0, G1, roots=(0, 1)) == 8
44: (8)                     assert graph_edit_distance(G0, G1, roots=(1, 2), timeout=5) == 0
45: (8)                     assert graph_edit_distance(G0, G1, roots=(0, 1), timeout=5) == 8
46: (8)                     assert graph_edit_distance(G0, G1, roots=(0, 1), timeout=0.0001) is
None
47: (8)                     pytest.raises(nx.NetworkXError, graph_edit_distance, G0, G1,
timeout=0)
48: (4)                 def test_graph_edit_distance(self):
49: (8)                     G0 = nx.Graph()
50: (8)                     G1 = path_graph(6)
51: (8)                     G2 = cycle_graph(6)
52: (8)                     G3 = wheel_graph(7)
53: (8)                     assert graph_edit_distance(G0, G0) == 0
54: (8)                     assert graph_edit_distance(G0, G1) == 11
55: (8)                     assert graph_edit_distance(G1, G0) == 11
56: (8)                     assert graph_edit_distance(G0, G2) == 12
57: (8)                     assert graph_edit_distance(G2, G0) == 12
58: (8)                     assert graph_edit_distance(G0, G3) == 19
59: (8)                     assert graph_edit_distance(G3, G0) == 19
60: (8)                     assert graph_edit_distance(G1, G1) == 0
61: (8)                     assert graph_edit_distance(G1, G2) == 1
```

```
(8)                 assert graph_edit_distance(G2, G1) == 1
(8)                 assert graph_edit_distance(G1, G3) == 8
(8)                 assert graph_edit_distance(G3, G1) == 8
(8)                 assert graph_edit_distance(G2, G2) == 0
(8)                 assert graph_edit_distance(G2, G3) == 7
(8)                 assert graph_edit_distance(G3, G2) == 7
(8)                 assert graph_edit_distance(G3, G3) == 0
(4)             def test_graph_edit_distance_node_match(self):
(8)                 G1 = cycle_graph(5)
(8)                 G2 = cycle_graph(5)
(8)                 for n, attr in G1.nodes.items():
(12)                    attr["color"] = "red" if n % 2 == 0 else "blue"
(8)                 for n, attr in G2.nodes.items():
(12)                    attr["color"] = "red" if n % 2 == 1 else "blue"
(8)                 assert graph_edit_distance(G1, G2) == 0
(8)                 assert (
(12)                    graph_edit_distance(
(16)                        G1, G2, node_match=lambda n1, n2: n1["color"] == n2["color"]
(12)                    )
(12)                    == 1
(8)                 )
(4)             def test_graph_edit_distance_edge_match(self):
(8)                 G1 = path_graph(6)
(8)                 G2 = path_graph(6)
(8)                 for e, attr in G1.edges.items():
(12)                    attr["color"] = "red" if min(e) % 2 == 0 else "blue"
(8)                 for e, attr in G2.edges.items():
(12)                    attr["color"] = "red" if min(e) // 3 == 0 else "blue"
(8)                 assert graph_edit_distance(G1, G2) == 0
(8)                 assert (
(12)                    graph_edit_distance(
(16)                        G1, G2, edge_match=lambda e1, e2: e1["color"] == e2["color"]
(12)                    )
(12)                    == 2
(8)                 )
(4)             def test_graph_edit_distance_node_cost(self):
(8)                 G1 = path_graph(6)
(8)                 G2 = path_graph(6)
(8)                 for n, attr in G1.nodes.items():
(12)                    attr["color"] = "red" if n % 2 == 0 else "blue"
(8)                 for n, attr in G2.nodes.items():
(12)                    attr["color"] = "red" if n % 2 == 1 else "blue"
(8)                 def node_subst_cost(uattr, vattr):
(12)                    if uattr["color"] == vattr["color"]:
(16)                        return 1
(12)                    else:
(16)                        return 10
(8)                 def node_del_cost(attr):
(12)                    if attr["color"] == "blue":
(16)                        return 20
(12)                    else:
(16)                        return 50
(8)                 def node_ins_cost(attr):
(12)                    if attr["color"] == "blue":
(16)                        return 40
(12)                    else:
(16)                        return 100
(8)                 assert (
(12)                    graph_edit_distance(
(16)                        G1,
(16)                        G2,
(16)                        node_subst_cost=node_subst_cost,
(16)                        node_del_cost=node_del_cost,
(16)                        node_ins_cost=node_ins_cost,
(12)                    )
(12)                    == 6
(8)                 )
(4)             def test_graph_edit_distance_edge_cost(self):
(8)                 G1 = path_graph(6)
```

```
(8)                     G2 = path_graph(6)
(8)                     for e, attr in G1.edges.items():
(12)                        attr["color"] = "red" if min(e) % 2 == 0 else "blue"
(8)                     for e, attr in G2.edges.items():
(12)                        attr["color"] = "red" if min(e) // 3 == 0 else "blue"
(8)                     def edge_subst_cost(gattr, hattr):
(12)                        if gattr["color"] == hattr["color"]:
(16)                            return 0.01
(12)                        else:
(16)                            return 0.1
(8)                     def edge_del_cost(attr):
(12)                        if attr["color"] == "blue":
(16)                            return 0.2
(12)                        else:
(16)                            return 0.5
(8)                     def edge_ins_cost(attr):
(12)                        if attr["color"] == "blue":
(16)                            return 0.4
(12)                        else:
(16)                            return 1.0
(8)                     assert (
(12)                        graph_edit_distance(
(16)                            G1,
(16)                            G2,
(16)                            edge_subst_cost=edge_subst_cost,
(16)                            edge_del_cost=edge_del_cost,
(16)                            edge_ins_cost=edge_ins_cost,
(12)                        )
(12)                        == 0.23
(8)                     )
(4)                 def test_graph_edit_distance_upper_bound(self):
(8)                     G1 = circular_ladder_graph(2)
(8)                     G2 = circular_ladder_graph(6)
(8)                     assert graph_edit_distance(G1, G2, upper_bound=5) is None
(8)                     assert graph_edit_distance(G1, G2, upper_bound=24) == 22
(8)                     assert graph_edit_distance(G1, G2) == 22
(4)                 def test_optimal_edit_paths(self):
(8)                     G1 = path_graph(3)
(8)                     G2 = cycle_graph(3)
(8)                     paths, cost = optimal_edit_paths(G1, G2)
(8)                     assert cost == 1
(8)                     assert len(paths) == 6
(8)                     def canonical(vertex_path, edge_path):
(12)                        return (
(16)                            tuple(sorted(vertex_path)),
(16)                            tuple(sorted(edge_path, key=lambda x: (None in x, x))),
(12)                        )
(8)                     expected_paths = [
(12)                        (
(16)                            [(0, 0), (1, 1), (2, 2)],
(16)                            [((0, 1), (0, 1)), ((1, 2), (1, 2)), (None, (0, 2))],
(12)                        ),
(12)                        (
(16)                            [(0, 0), (1, 2), (2, 1)],
(16)                            [((0, 1), (0, 2)), ((1, 2), (1, 2)), (None, (0, 1))],
(12)                        ),
(12)                        (
(16)                            [(0, 1), (1, 0), (2, 2)],
(16)                            [((0, 1), (0, 1)), ((1, 2), (0, 2)), (None, (1, 2))],
(12)                        ),
(12)                        (
(16)                            [(0, 1), (1, 2), (2, 0)],
(16)                            [((0, 1), (1, 2)), ((1, 2), (0, 2)), (None, (0, 1))],
(12)                        ),
(12)                        (
(16)                            [(0, 2), (1, 0), (2, 1)],
(16)                            [((0, 1), (0, 2)), ((1, 2), (0, 1)), (None, (1, 2))],
(12)                        ),
(12)                        (
```

```
(16)                            [(0, 2), (1, 1), (2, 0)],
(16)                            [((0, 1), (1, 2)), ((1, 2), (0, 1)), (None, (0, 2))],
(12)                        ),
(8)                     ]
(8)                     assert {canonical(*p) for p in paths} == {canonical(*p) for p in expected_paths}
(4)                 def test_optimize_graph_edit_distance(self):
(8)                     G1 = circular_ladder_graph(2)
(8)                     G2 = circular_ladder_graph(6)
(8)                     bestcost = 1000
(8)                     for cost in optimize_graph_edit_distance(G1, G2):
(12)                        assert cost < bestcost
(12)                        bestcost = cost
(8)                     assert bestcost == 22
(4)                 def test_selfloops(self):
(8)                     G0 = nx.Graph()
(8)                     G1 = nx.Graph()
(8)                     G1.add_edges_from(((("A", "A"), ("A", "B")))
(8)                     G2 = nx.Graph()
(8)                     G2.add_edges_from(((("A", "B"), ("B", "B")))
(8)                     G3 = nx.Graph()
(8)                     G3.add_edges_from(((("A", "A"), ("A", "B"), ("B", "B")))
(8)                     assert graph_edit_distance(G0, G0) == 0
(8)                     assert graph_edit_distance(G0, G1) == 4
(8)                     assert graph_edit_distance(G1, G0) == 4
(8)                     assert graph_edit_distance(G0, G2) == 4
(8)                     assert graph_edit_distance(G2, G0) == 4
(8)                     assert graph_edit_distance(G0, G3) == 5
(8)                     assert graph_edit_distance(G3, G0) == 5
(8)                     assert graph_edit_distance(G1, G1) == 0
(8)                     assert graph_edit_distance(G1, G2) == 0
(8)                     assert graph_edit_distance(G2, G1) == 0
(8)                     assert graph_edit_distance(G1, G3) == 1
(8)                     assert graph_edit_distance(G3, G1) == 1
(8)                     assert graph_edit_distance(G2, G2) == 0
(8)                     assert graph_edit_distance(G2, G3) == 1
(8)                     assert graph_edit_distance(G3, G2) == 1
(8)                     assert graph_edit_distance(G3, G3) == 0
(4)                 def test_digraph(self):
(8)                     G0 = nx.DiGraph()
(8)                     G1 = nx.DiGraph()
(8)                     G1.add_edges_from(((("A", "B"), ("B", "C"), ("C", "D"), ("D", "A")))
(8)                     G2 = nx.DiGraph()
(8)                     G2.add_edges_from(((("A", "B"), ("B", "C"), ("C", "D"), ("A", "D")))
(8)                     G3 = nx.DiGraph()
(8)                     G3.add_edges_from(((("A", "B"), ("A", "C"), ("B", "D"), ("C", "D")))
(8)                     assert graph_edit_distance(G0, G0) == 0
(8)                     assert graph_edit_distance(G0, G1) == 8
(8)                     assert graph_edit_distance(G1, G0) == 8
(8)                     assert graph_edit_distance(G0, G2) == 8
(8)                     assert graph_edit_distance(G2, G0) == 8
(8)                     assert graph_edit_distance(G0, G3) == 8
(8)                     assert graph_edit_distance(G3, G0) == 8
(8)                     assert graph_edit_distance(G1, G1) == 0
(8)                     assert graph_edit_distance(G1, G2) == 2
(8)                     assert graph_edit_distance(G2, G1) == 2
(8)                     assert graph_edit_distance(G1, G3) == 4
(8)                     assert graph_edit_distance(G3, G1) == 4
(8)                     assert graph_edit_distance(G2, G2) == 0
(8)                     assert graph_edit_distance(G2, G3) == 2
(8)                     assert graph_edit_distance(G3, G2) == 2
(8)                     assert graph_edit_distance(G3, G3) == 0
(4)                 def test_multigraph(self):
(8)                     G0 = nx.MultiGraph()
(8)                     G1 = nx.MultiGraph()
(8)                     G1.add_edges_from(((("A", "B"), ("B", "C"), ("A", "C")))
(8)                     G2 = nx.MultiGraph()
(8)                     G2.add_edges_from(((("A", "B"), ("B", "C"), ("B", "C"), ("A", "C")))
(8)                     G3 = nx.MultiGraph()
```

```
268: (8)                    G3.add_edges_from((("A", "B"), ("B", "C"), ("A", "C"), ("A", "C"),
("A", "C")))
269: (8)                    assert graph_edit_distance(G0, G0) == 0
270: (8)                    assert graph_edit_distance(G0, G1) == 6
271: (8)                    assert graph_edit_distance(G1, G0) == 6
272: (8)                    assert graph_edit_distance(G0, G2) == 7
273: (8)                    assert graph_edit_distance(G2, G0) == 7
274: (8)                    assert graph_edit_distance(G0, G3) == 8
275: (8)                    assert graph_edit_distance(G3, G0) == 8
276: (8)                    assert graph_edit_distance(G1, G1) == 0
277: (8)                    assert graph_edit_distance(G1, G2) == 1
278: (8)                    assert graph_edit_distance(G2, G1) == 1
279: (8)                    assert graph_edit_distance(G1, G3) == 2
280: (8)                    assert graph_edit_distance(G3, G1) == 2
281: (8)                    assert graph_edit_distance(G2, G2) == 0
282: (8)                    assert graph_edit_distance(G2, G3) == 1
283: (8)                    assert graph_edit_distance(G3, G2) == 1
284: (8)                    assert graph_edit_distance(G3, G3) == 0
285: (4)                def test_multidigraph(self):
286: (8)                    G1 = nx.MultiDiGraph()
287: (8)                    G1.add_edges_from(
288: (12)                       (
289: (16)                           ("hardware", "kernel"),
290: (16)                           ("kernel", "hardware"),
291: (16)                           ("kernel", "userspace"),
292: (16)                           ("userspace", "kernel"),
293: (12)                       )
294: (8)                    )
295: (8)                    G2 = nx.MultiDiGraph()
296: (8)                    G2.add_edges_from(
297: (12)                       (
298: (16)                           ("winter", "spring"),
299: (16)                           ("spring", "summer"),
300: (16)                           ("summer", "autumn"),
301: (16)                           ("autumn", "winter"),
302: (12)                       )
303: (8)                    )
304: (8)                    assert graph_edit_distance(G1, G2) == 5
305: (8)                    assert graph_edit_distance(G2, G1) == 5
306: (4)                def testCopy(self):
307: (8)                    G = nx.Graph()
308: (8)                    G.add_node("A", label="A")
309: (8)                    G.add_node("B", label="B")
310: (8)                    G.add_edge("A", "B", label="a-b")
311: (8)                    assert (
312: (12)                       graph_edit_distance(G, G.copy(), node_match=nmatch,
edge_match=ematch) == 0
313: (8)                    )
314: (4)                def testSame(self):
315: (8)                    G1 = nx.Graph()
316: (8)                    G1.add_node("A", label="A")
317: (8)                    G1.add_node("B", label="B")
318: (8)                    G1.add_edge("A", "B", label="a-b")
319: (8)                    G2 = nx.Graph()
320: (8)                    G2.add_node("A", label="A")
321: (8)                    G2.add_node("B", label="B")
322: (8)                    G2.add_edge("A", "B", label="a-b")
323: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 0
324: (4)                def testOneEdgeLabelDiff(self):
325: (8)                    G1 = nx.Graph()
326: (8)                    G1.add_node("A", label="A")
327: (8)                    G1.add_node("B", label="B")
328: (8)                    G1.add_edge("A", "B", label="a-b")
329: (8)                    G2 = nx.Graph()
330: (8)                    G2.add_node("A", label="A")
331: (8)                    G2.add_node("B", label="B")
332: (8)                    G2.add_edge("A", "B", label="bad")
333: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
```

```
edge_match=ematch) == 1
(4)                 def testOneNodeLabelDiff(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_node("A", label="A")
(8)                     G1.add_node("B", label="B")
(8)                     G1.add_edge("A", "B", label="a-b")
(8)                     G2 = nx.Graph()
(8)                     G2.add_node("A", label="Z")
(8)                     G2.add_node("B", label="B")
(8)                     G2.add_edge("A", "B", label="a-b")
(8)                     assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 1
(4)                 def testOneExtraNode(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_node("A", label="A")
(8)                     G1.add_node("B", label="B")
(8)                     G1.add_edge("A", "B", label="a-b")
(8)                     G2 = nx.Graph()
(8)                     G2.add_node("A", label="A")
(8)                     G2.add_node("B", label="B")
(8)                     G2.add_edge("A", "B", label="a-b")
(8)                     G2.add_node("C", label="C")
(8)                     assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 1
(4)                 def testOneExtraEdge(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_node("A", label="A")
(8)                     G1.add_node("B", label="B")
(8)                     G1.add_node("C", label="C")
(8)                     G1.add_node("C", label="C")
(8)                     G1.add_edge("A", "B", label="a-b")
(8)                     G2 = nx.Graph()
(8)                     G2.add_node("A", label="A")
(8)                     G2.add_node("B", label="B")
(8)                     G2.add_node("C", label="C")
(8)                     G2.add_edge("A", "B", label="a-b")
(8)                     G2.add_edge("A", "C", label="a-c")
(8)                     assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 1
(4)                 def testOneExtraNodeAndEdge(self):
(8)                     G1 = nx.Graph()
(8)                     G1.add_node("A", label="A")
(8)                     G1.add_node("B", label="B")
(8)                     G1.add_edge("A", "B", label="a-b")
(8)                     G2 = nx.Graph()
(8)                     G2.add_node("A", label="A")
(8)                     G2.add_node("B", label="B")
(8)                     G2.add_node("C", label="C")
(8)                     G2.add_edge("A", "B", label="a-b")
(8)                     G2.add_edge("A", "C", label="a-c")
(8)                     assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 2
(4)                 def testGraph1(self):
(8)                     G1 = getCanonical()
(8)                     G2 = nx.Graph()
(8)                     G2.add_node("A", label="A")
(8)                     G2.add_node("B", label="B")
(8)                     G2.add_node("D", label="D")
(8)                     G2.add_node("E", label="E")
(8)                     G2.add_edge("A", "B", label="a-b")
(8)                     G2.add_edge("B", "D", label="b-d")
(8)                     G2.add_edge("D", "E", label="d-e")
(8)                     assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 3
(4)                 def testGraph2(self):
(8)                     G1 = getCanonical()
(8)                     G2 = nx.Graph()
(8)                     G2.add_node("A", label="A")
(8)                     G2.add_node("B", label="B")
```

```
397: (8)                    G2.add_node("C", label="C")
398: (8)                    G2.add_node("D", label="D")
399: (8)                    G2.add_node("E", label="E")
400: (8)                    G2.add_edge("A", "B", label="a-b")
401: (8)                    G2.add_edge("B", "C", label="b-c")
402: (8)                    G2.add_edge("C", "D", label="c-d")
403: (8)                    G2.add_edge("C", "E", label="c-e")
404: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 4
405: (4)                def testGraph3(self):
406: (8)                    G1 = getCanonical()
407: (8)                    G2 = nx.Graph()
408: (8)                    G2.add_node("A", label="A")
409: (8)                    G2.add_node("B", label="B")
410: (8)                    G2.add_node("C", label="C")
411: (8)                    G2.add_node("D", label="D")
412: (8)                    G2.add_node("E", label="E")
413: (8)                    G2.add_node("F", label="F")
414: (8)                    G2.add_node("G", label="G")
415: (8)                    G2.add_edge("A", "C", label="a-c")
416: (8)                    G2.add_edge("A", "D", label="a-d")
417: (8)                    G2.add_edge("D", "E", label="d-e")
418: (8)                    G2.add_edge("D", "F", label="d-f")
419: (8)                    G2.add_edge("D", "G", label="d-g")
420: (8)                    G2.add_edge("E", "B", label="e-b")
421: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 12
422: (4)                def testGraph4(self):
423: (8)                    G1 = getCanonical()
424: (8)                    G2 = nx.Graph()
425: (8)                    G2.add_node("A", label="A")
426: (8)                    G2.add_node("B", label="B")
427: (8)                    G2.add_node("C", label="C")
428: (8)                    G2.add_node("D", label="D")
429: (8)                    G2.add_edge("A", "B", label="a-b")
430: (8)                    G2.add_edge("B", "C", label="b-c")
431: (8)                    G2.add_edge("C", "D", label="c-d")
432: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 2
433: (4)                def testGraph4_a(self):
434: (8)                    G1 = getCanonical()
435: (8)                    G2 = nx.Graph()
436: (8)                    G2.add_node("A", label="A")
437: (8)                    G2.add_node("B", label="B")
438: (8)                    G2.add_node("C", label="C")
439: (8)                    G2.add_node("D", label="D")
440: (8)                    G2.add_edge("A", "B", label="a-b")
441: (8)                    G2.add_edge("B", "C", label="b-c")
442: (8)                    G2.add_edge("A", "D", label="a-d")
443: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 2
444: (4)                def testGraph4_b(self):
445: (8)                    G1 = getCanonical()
446: (8)                    G2 = nx.Graph()
447: (8)                    G2.add_node("A", label="A")
448: (8)                    G2.add_node("B", label="B")
449: (8)                    G2.add_node("C", label="C")
450: (8)                    G2.add_node("D", label="D")
451: (8)                    G2.add_edge("A", "B", label="a-b")
452: (8)                    G2.add_edge("B", "C", label="b-c")
453: (8)                    G2.add_edge("B", "D", label="bad")
454: (8)                    assert graph_edit_distance(G1, G2, node_match=nmatch,
edge_match=ematch) == 1
455: (4)                simrank_algs = [
456: (8)                    nx.simrank_similarity,
457: (8)                    nx.algorithms.similarity._simrank_similarity_python,
458: (4)                ]
459: (4)                @pytest.mark.parametrize("simrank_similarity", simrank_algs)
460: (4)                def test_simrank_no_source_no_target(self, simrank_similarity):
```

```
(8)                     G = nx.cycle_graph(5)
(8)                     expected = {
(12)                        0: {
(16)                            0: 1,
(16)                            1: 0.3951219505902448,
(16)                            2: 0.5707317069281646,
(16)                            3: 0.5707317069281646,
(16)                            4: 0.3951219505902449,
(12)                        },
(12)                        1: {
(16)                            0: 0.3951219505902448,
(16)                            1: 1,
(16)                            2: 0.3951219505902449,
(16)                            3: 0.5707317069281646,
(16)                            4: 0.5707317069281646,
(12)                        },
(12)                        2: {
(16)                            0: 0.5707317069281646,
(16)                            1: 0.3951219505902449,
(16)                            2: 1,
(16)                            3: 0.3951219505902449,
(16)                            4: 0.5707317069281646,
(12)                        },
(12)                        3: {
(16)                            0: 0.5707317069281646,
(16)                            1: 0.5707317069281646,
(16)                            2: 0.3951219505902449,
(16)                            3: 1,
(16)                            4: 0.3951219505902449,
(12)                        },
(12)                        4: {
(16)                            0: 0.3951219505902449,
(16)                            1: 0.5707317069281646,
(16)                            2: 0.5707317069281646,
(16)                            3: 0.3951219505902449,
(16)                            4: 1,
(12)                        },
(8)                     }
(8)                     actual = simrank_similarity(G)
(8)                     for k, v in expected.items():
(12)                        assert v == pytest.approx(actual[k], abs=1e-2)
(8)                     G = nx.DiGraph()
(8)                     G.add_node(0, label="Univ")
(8)                     G.add_node(1, label="ProfA")
(8)                     G.add_node(2, label="ProfB")
(8)                     G.add_node(3, label="StudentA")
(8)                     G.add_node(4, label="StudentB")
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 4), (4, 2), (3, 0)])
(8)                     expected = {
(12)                        0: {0: 1, 1: 0.0, 2: 0.1323363991265798, 3: 0.0, 4:
0.03387811817640443},
(12)                        1: {0: 0.0, 1: 1, 2: 0.4135512472705618, 3: 0.0, 4:
0.10586911930126384},
(12)                        2: {
(16)                            0: 0.1323363991265798,
(16)                            1: 0.4135512472705618,
(16)                            2: 1,
(16)                            3: 0.04234764772050554,
(16)                            4: 0.08822426608438655,
(12)                        },
(12)                        3: {0: 0.0, 1: 0.0, 2: 0.04234764772050554, 3: 1, 4:
0.3308409978164495},
(12)                        4: {
(16)                            0: 0.03387811817640443,
(16)                            1: 0.10586911930126384,
(16)                            2: 0.08822426608438655,
(16)                            3: 0.3308409978164495,
(16)                            4: 1,
(12)                        },
```

```
527: (8)                    }
528: (8)                    actual = simrank_similarity(G, importance_factor=0.8)
529: (8)                    for k, v in expected.items():
530: (12)                       assert v == pytest.approx(actual[k], abs=1e-2)
531: (4)                @pytest.mark.parametrize("simrank_similarity", simrank_algs)
532: (4)                def test_simrank_source_no_target(self, simrank_similarity):
533: (8)                    G = nx.cycle_graph(5)
534: (8)                    expected = {
535: (12)                       0: 1,
536: (12)                       1: 0.3951219505902448,
537: (12)                       2: 0.5707317069281646,
538: (12)                       3: 0.5707317069281646,
539: (12)                       4: 0.3951219505902449,
540: (8)                    }
541: (8)                    actual = simrank_similarity(G, source=0)
542: (8)                    assert expected == pytest.approx(actual, abs=1e-2)
543: (8)                    G = nx.DiGraph()
544: (8)                    G.add_node(0, label="Univ")
545: (8)                    G.add_node(1, label="ProfA")
546: (8)                    G.add_node(2, label="ProfB")
547: (8)                    G.add_node(3, label="StudentA")
548: (8)                    G.add_node(4, label="StudentB")
549: (8)                    G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 4), (4, 2), (3, 0)])
550: (8)                    expected = {0: 1, 1: 0.0, 2: 0.1323363991265798, 3: 0.0, 4:
0.03387811817640443}
551: (8)                    actual = simrank_similarity(G, importance_factor=0.8, source=0)
552: (8)                    assert expected == pytest.approx(actual, abs=1e-2)
553: (4)                @pytest.mark.parametrize("simrank_similarity", simrank_algs)
554: (4)                def test_simrank_noninteger_nodes(self, simrank_similarity):
555: (8)                    G = nx.cycle_graph(5)
556: (8)                    G = nx.relabel_nodes(G, dict(enumerate("abcde")))
557: (8)                    expected = {
558: (12)                       "a": 1,
559: (12)                       "b": 0.3951219505902448,
560: (12)                       "c": 0.5707317069281646,
561: (12)                       "d": 0.5707317069281646,
562: (12)                       "e": 0.3951219505902449,
563: (8)                    }
564: (8)                    actual = simrank_similarity(G, source="a")
565: (8)                    assert expected == pytest.approx(actual, abs=1e-2)
566: (8)                    G = nx.DiGraph()
567: (8)                    G.add_node(0, label="Univ")
568: (8)                    G.add_node(1, label="ProfA")
569: (8)                    G.add_node(2, label="ProfB")
570: (8)                    G.add_node(3, label="StudentA")
571: (8)                    G.add_node(4, label="StudentB")
572: (8)                    G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 4), (4, 2), (3, 0)])
573: (8)                    node_labels = dict(enumerate(nx.get_node_attributes(G,
"label").values()))
574: (8)                    G = nx.relabel_nodes(G, node_labels)
575: (8)                    expected = {
576: (12)                       "Univ": 1,
577: (12)                       "ProfA": 0.0,
578: (12)                       "ProfB": 0.1323363991265798,
579: (12)                       "StudentA": 0.0,
580: (12)                       "StudentB": 0.03387811817640443,
581: (8)                    }
582: (8)                    actual = simrank_similarity(G, importance_factor=0.8, source="Univ")
583: (8)                    assert expected == pytest.approx(actual, abs=1e-2)
584: (4)                @pytest.mark.parametrize("simrank_similarity", simrank_algs)
585: (4)                def test_simrank_source_and_target(self, simrank_similarity):
586: (8)                    G = nx.cycle_graph(5)
587: (8)                    expected = 1
588: (8)                    actual = simrank_similarity(G, source=0, target=0)
589: (8)                    assert expected == pytest.approx(actual, abs=1e-2)
590: (8)                    G = nx.DiGraph()
591: (8)                    G.add_node(0, label="Univ")
592: (8)                    G.add_node(1, label="ProfA")
593: (8)                    G.add_node(2, label="ProfB")
```

```
(8)                     G.add_node(3, label="StudentA")
(8)                     G.add_node(4, label="StudentB")
(8)                     G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 4), (4, 2), (3, 0)])
(8)                     expected = 0.1323363991265798
(8)                     actual = simrank_similarity(G, importance_factor=0.8, source=0,
target=2)
(8)                     assert expected == pytest.approx(actual, abs=1e-5)
(4)                 @pytest.mark.parametrize("alg", simrank_algs)
(4)                 def test_simrank_max_iterations(self, alg):
(8)                     G = nx.cycle_graph(5)
(8)                     pytest.raises(nx.ExceededMaxIterations, alg, G, max_iterations=10)
(4)                 def test_simrank_source_not_found(self):
(8)                     G = nx.cycle_graph(5)
(8)                     with pytest.raises(nx.NodeNotFound, match="Source node 10 not in G"):
(12)                        nx.simrank_similarity(G, source=10)
(4)                 def test_simrank_target_not_found(self):
(8)                     G = nx.cycle_graph(5)
(8)                     with pytest.raises(nx.NodeNotFound, match="Target node 10 not in G"):
(12)                        nx.simrank_similarity(G, target=10)
(4)                 def test_simrank_between_versions(self):
(8)                     G = nx.cycle_graph(5)
(8)                     expected_python_tol4 = {
(12)                        0: 1,
(12)                        1: 0.394512499239852,
(12)                        2: 0.5703550452791322,
(12)                        3: 0.5703550452791323,
(12)                        4: 0.394512499239852,
(8)                     }
(8)                     expected_numpy_tol4 = {
(12)                        0: 1.0,
(12)                        1: 0.3947180735764555,
(12)                        2: 0.570482097206368,
(12)                        3: 0.570482097206368,
(12)                        4: 0.3947180735764555,
(8)                     }
(8)                     actual = nx.simrank_similarity(G, source=0)
(8)                     assert expected_numpy_tol4 == pytest.approx(actual, abs=1e-7)
(8)                     assert expected_python_tol4 != pytest.approx(actual, abs=1e-4)
(8)                     assert expected_python_tol4 == pytest.approx(actual, abs=1e-3)
(8)                     actual = nx.similarity._simrank_similarity_python(G, source=0)
(8)                     assert expected_python_tol4 == pytest.approx(actual, abs=1e-7)
(8)                     assert expected_numpy_tol4 != pytest.approx(actual, abs=1e-4)
(8)                     assert expected_numpy_tol4 == pytest.approx(actual, abs=1e-3)
(4)                 def test_simrank_numpy_no_source_no_target(self):
(8)                     G = nx.cycle_graph(5)
(8)                     expected = np.array(
(12)                        [
(16)                            [
(20)                                1.0,
(20)                                0.3947180735764555,
(20)                                0.570482097206368,
(20)                                0.570482097206368,
(20)                                0.3947180735764555,
(16)                            ],
(16)                            [
(20)                                0.3947180735764555,
(20)                                1.0,
(20)                                0.3947180735764555,
(20)                                0.570482097206368,
(20)                                0.570482097206368,
(16)                            ],
(16)                            [
(20)                                0.570482097206368,
(20)                                0.3947180735764555,
(20)                                1.0,
(20)                                0.3947180735764555,
(20)                                0.570482097206368,
(16)                            ],
(16)                            [
```

```
(20)                                0.570482097206368,
(20)                                0.570482097206368,
(20)                                0.3947180735764555,
(20)                                1.0,
(20)                                0.3947180735764555,
(16)                            ],
(16)                            [
(20)                                0.3947180735764555,
(20)                                0.570482097206368,
(20)                                0.570482097206368,
(20)                                0.3947180735764555,
(20)                                1.0,
(16)                            ],
(12)                        ]
(8)                     )
(8)                     actual = nx.similarity._simrank_similarity_numpy(G)
(8)                     np.testing.assert_allclose(expected, actual, atol=1e-7)
(4)                 def test_simrank_numpy_source_no_target(self):
(8)                     G = nx.cycle_graph(5)
(8)                     expected = np.array(
(12)                        [
(16)                            1.0,
(16)                            0.3947180735764555,
(16)                            0.570482097206368,
(16)                            0.570482097206368,
(16)                            0.3947180735764555,
(12)                        ]
(8)                     )
(8)                     actual = nx.similarity._simrank_similarity_numpy(G, source=0)
(8)                     np.testing.assert_allclose(expected, actual, atol=1e-7)
(4)                 def test_simrank_numpy_source_and_target(self):
(8)                     G = nx.cycle_graph(5)
(8)                     expected = 1.0
(8)                     actual = nx.similarity._simrank_similarity_numpy(G, source=0,
target=0)
(8)                     np.testing.assert_allclose(expected, actual, atol=1e-7)
(4)                 def test_panther_similarity_unweighted(self):
(8)                     np.random.seed(42)
(8)                     G = nx.Graph()
(8)                     G.add_edge(0, 1)
(8)                     G.add_edge(0, 2)
(8)                     G.add_edge(0, 3)
(8)                     G.add_edge(1, 2)
(8)                     G.add_edge(2, 4)
(8)                     expected = {3: 0.5, 2: 0.5, 1: 0.5, 4: 0.125}
(8)                     sim = nx.panther_similarity(G, 0, path_length=2)
(8)                     assert sim == expected
(4)                 def test_panther_similarity_weighted(self):
(8)                     np.random.seed(42)
(8)                     G = nx.Graph()
(8)                     G.add_edge("v1", "v2", w=5)
(8)                     G.add_edge("v1", "v3", w=1)
(8)                     G.add_edge("v1", "v4", w=2)
(8)                     G.add_edge("v2", "v3", w=0.1)
(8)                     G.add_edge("v3", "v5", w=1)
(8)                     expected = {"v3": 0.75, "v4": 0.5, "v2": 0.5, "v5": 0.25}
(8)                     sim = nx.panther_similarity(G, "v1", path_length=2, weight="w")
(8)                     assert sim == expected
(4)                 def test_panther_similarity_source_not_found(self):
(8)                     G = nx.Graph()
(8)                     G.add_edges_from([(0, 1), (0, 2), (0, 3), (1, 2), (2, 4)])
(8)                     with pytest.raises(nx.NodeNotFound, match="Source node 10 not in G"):
(12)                        nx.panther_similarity(G, source=10)
(4)                 def test_panther_similarity_isolated(self):
(8)                     G = nx.Graph()
(8)                     G.add_nodes_from(range(5))
(8)                     with pytest.raises(
(12)                        nx.NetworkXUnfeasible,
(12)                        match="Panther similarity is not defined for the isolated source
```

```
node 1.",
730: (8)                    ):
731: (12)                       nx.panther_similarity(G, source=1)
732: (4)                def test_generate_random_paths_unweighted(self):
733: (8)                    index_map = {}
734: (8)                    num_paths = 10
735: (8)                    path_length = 2
736: (8)                    G = nx.Graph()
737: (8)                    G.add_edge(0, 1)
738: (8)                    G.add_edge(0, 2)
739: (8)                    G.add_edge(0, 3)
740: (8)                    G.add_edge(1, 2)
741: (8)                    G.add_edge(2, 4)
742: (8)                    paths = nx.generate_random_paths(
743: (12)                       G, num_paths, path_length=path_length, index_map=index_map,
seed=42
744: (8)                    )
745: (8)                    expected_paths = [
746: (12)                       [3, 0, 3],
747: (12)                       [4, 2, 1],
748: (12)                       [2, 1, 0],
749: (12)                       [2, 0, 3],
750: (12)                       [3, 0, 1],
751: (12)                       [3, 0, 1],
752: (12)                       [4, 2, 0],
753: (12)                       [2, 1, 0],
754: (12)                       [3, 0, 2],
755: (12)                       [2, 1, 2],
756: (8)                    ]
757: (8)                    expected_map = {
758: (12)                       0: {0, 2, 3, 4, 5, 6, 7, 8},
759: (12)                       1: {1, 2, 4, 5, 7, 9},
760: (12)                       2: {1, 2, 3, 6, 7, 8, 9},
761: (12)                       3: {0, 3, 4, 5, 8},
762: (12)                       4: {1, 6},
763: (8)                    }
764: (8)                    assert expected_paths == list(paths)
765: (8)                    assert expected_map == index_map
766: (4)                def test_generate_random_paths_weighted(self):
767: (8)                    np.random.seed(42)
768: (8)                    index_map = {}
769: (8)                    num_paths = 10
770: (8)                    path_length = 6
771: (8)                    G = nx.Graph()
772: (8)                    G.add_edge("a", "b", weight=0.6)
773: (8)                    G.add_edge("a", "c", weight=0.2)
774: (8)                    G.add_edge("c", "d", weight=0.1)
775: (8)                    G.add_edge("c", "e", weight=0.7)
776: (8)                    G.add_edge("c", "f", weight=0.9)
777: (8)                    G.add_edge("a", "d", weight=0.3)
778: (8)                    paths = nx.generate_random_paths(
779: (12)                       G, num_paths, path_length=path_length, index_map=index_map
780: (8)                    )
781: (8)                    expected_paths = [
782: (12)                       ["d", "c", "f", "c", "d", "a", "b"],
783: (12)                       ["e", "c", "f", "c", "f", "c", "e"],
784: (12)                       ["d", "a", "b", "a", "b", "a", "c"],
785: (12)                       ["b", "a", "d", "a", "b", "a", "b"],
786: (12)                       ["d", "a", "b", "a", "b", "a", "d"],
787: (12)                       ["d", "a", "b", "a", "b", "a", "c"],
788: (12)                       ["d", "a", "b", "a", "b", "a", "b"],
789: (12)                       ["f", "c", "f", "c", "f", "c", "e"],
790: (12)                       ["d", "a", "d", "a", "b", "a", "b"],
791: (12)                       ["e", "c", "f", "c", "e", "c", "d"],
792: (8)                    ]
793: (8)                    expected_map = {
794: (12)                       "d": {0, 2, 3, 4, 5, 6, 8, 9},
795: (12)                       "c": {0, 1, 2, 5, 7, 9},
796: (12)                       "f": {0, 1, 9, 7},
```

```
797: (12)                        "a": {0, 2, 3, 4, 5, 6, 8},
798: (12)                        "b": {0, 2, 3, 4, 5, 6, 8},
799: (12)                        "e": {1, 9, 7},
800: (8)                     }
801: (8)                     assert expected_paths == list(paths)
802: (8)                     assert expected_map == index_map
803: (4)                 def test_symmetry_with_custom_matching(self):
804: (8)                     print("G2 is edge (a,b) and G3 is edge (a,a)")
805: (8)                     print("but node order for G2 is (a,b) while for G3 it is (b,a)")
806: (8)                     a, b = "A", "B"
807: (8)                     G2 = nx.Graph()
808: (8)                     G2.add_nodes_from((a, b))
809: (8)                     G2.add_edges_from([(a, b)])
810: (8)                     G3 = nx.Graph()
811: (8)                     G3.add_nodes_from((b, a))
812: (8)                     G3.add_edges_from([(a, a)])
813: (8)                     for G in (G2, G3):
814: (12)                        for n in G:
815: (16)                            G.nodes[n]["attr"] = n
816: (12)                        for e in G.edges:
817: (16)                            G.edges[e]["attr"] = e
818: (8)                     match = lambda x, y: x == y
819: (8)                     print("Starting G2 to G3 GED calculation")
820: (8)                     assert nx.graph_edit_distance(G2, G3, node_match=match,
edge_match=match) == 1
821: (8)                     print("Starting G3 to G2 GED calculation")
822: (8)                     assert nx.graph_edit_distance(G3, G2, node_match=match,
edge_match=match) == 1
```

File 347 - test_simple_paths.py:

```
1: (0)              import random
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx import convert_node_labels_to_integers as cnlti
5: (0)              from networkx.algorithms.simple_paths import (
6: (4)                  _bidirectional_dijkstra,
7: (4)                  _bidirectional_shortest_path,
8: (0)              )
9: (0)              from networkx.utils import arbitrary_element, pairwise
10: (0)             class TestIsSimplePath:
11: (4)                 """Unit tests for the
12: (4)                 :func:`networkx.algorithms.simple_paths.is_simple_path` function.
13: (4)                 """
14: (4)                 def test_empty_list(self):
15: (8)                     """Tests that the empty list is not a valid path, since there
16: (8)                     should be a one-to-one correspondence between paths as lists of
17: (8)                     nodes and paths as lists of edges.
18: (8)                     """
19: (8)                     G = nx.trivial_graph()
20: (8)                     assert not nx.is_simple_path(G, [])
21: (4)                 def test_trivial_path(self):
22: (8)                     """Tests that the trivial path, a path of length one, is
23: (8)                     considered a simple path in a graph.
24: (8)                     """
25: (8)                     G = nx.trivial_graph()
26: (8)                     assert nx.is_simple_path(G, [0])
27: (4)                 def test_trivial_nonpath(self):
28: (8)                     """Tests that a list whose sole element is an object not in the
29: (8)                     graph is not considered a simple path.
30: (8)                     """
31: (8)                     G = nx.trivial_graph()
32: (8)                     assert not nx.is_simple_path(G, ["not a node"])
33: (4)                 def test_simple_path(self):
34: (8)                     G = nx.path_graph(2)
35: (8)                     assert nx.is_simple_path(G, [0, 1])
36: (4)                 def test_non_simple_path(self):
```

```
(8)                 G = nx.path_graph(2)
(8)                 assert not nx.is_simple_path(G, [0, 1, 0])
(4)             def test_cycle(self):
(8)                 G = nx.cycle_graph(3)
(8)                 assert not nx.is_simple_path(G, [0, 1, 2, 0])
(4)             def test_missing_node(self):
(8)                 G = nx.path_graph(2)
(8)                 assert not nx.is_simple_path(G, [0, 2])
(4)             def test_missing_starting_node(self):
(8)                 G = nx.path_graph(2)
(8)                 assert not nx.is_simple_path(G, [2, 0])
(4)             def test_directed_path(self):
(8)                 G = nx.DiGraph([(0, 1), (1, 2)])
(8)                 assert nx.is_simple_path(G, [0, 1, 2])
(4)             def test_directed_non_path(self):
(8)                 G = nx.DiGraph([(0, 1), (1, 2)])
(8)                 assert not nx.is_simple_path(G, [2, 1, 0])
(4)             def test_directed_cycle(self):
(8)                 G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
(8)                 assert not nx.is_simple_path(G, [0, 1, 2, 0])
(4)             def test_multigraph(self):
(8)                 G = nx.MultiGraph([(0, 1), (0, 1)])
(8)                 assert nx.is_simple_path(G, [0, 1])
(4)             def test_multidigraph(self):
(8)                 G = nx.MultiDiGraph([(0, 1), (0, 1), (1, 0), (1, 0)])
(8)                 assert nx.is_simple_path(G, [0, 1])
(0)         def test_all_simple_paths():
(4)             G = nx.path_graph(4)
(4)             paths = nx.all_simple_paths(G, 0, 3)
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2, 3)}
(0)         def test_all_simple_paths_with_two_targets_emits_two_paths():
(4)             G = nx.path_graph(4)
(4)             G.add_edge(2, 4)
(4)             paths = nx.all_simple_paths(G, 0, [3, 4])
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2, 3), (0, 1, 2, 4)}
(0)         def test_digraph_all_simple_paths_with_two_targets_emits_two_paths():
(4)             G = nx.path_graph(4, create_using=nx.DiGraph())
(4)             G.add_edge(2, 4)
(4)             paths = nx.all_simple_paths(G, 0, [3, 4])
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2, 3), (0, 1, 2, 4)}
(0)         def test_all_simple_paths_with_two_targets_cutoff():
(4)             G = nx.path_graph(4)
(4)             G.add_edge(2, 4)
(4)             paths = nx.all_simple_paths(G, 0, [3, 4], cutoff=3)
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2, 3), (0, 1, 2, 4)}
(0)         def test_digraph_all_simple_paths_with_two_targets_cutoff():
(4)             G = nx.path_graph(4, create_using=nx.DiGraph())
(4)             G.add_edge(2, 4)
(4)             paths = nx.all_simple_paths(G, 0, [3, 4], cutoff=3)
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2, 3), (0, 1, 2, 4)}
(0)         def test_all_simple_paths_with_two_targets_in_line_emits_two_paths():
(4)             G = nx.path_graph(4)
(4)             paths = nx.all_simple_paths(G, 0, [2, 3])
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2), (0, 1, 2, 3)}
(0)         def test_all_simple_paths_ignores_cycle():
(4)             G = nx.cycle_graph(3, create_using=nx.DiGraph())
(4)             G.add_edge(1, 3)
(4)             paths = nx.all_simple_paths(G, 0, 3)
(4)             assert {tuple(p) for p in paths} == {(0, 1, 3)}
(0)         def test_all_simple_paths_with_two_targets_inside_cycle_emits_two_paths():
(4)             G = nx.cycle_graph(3, create_using=nx.DiGraph())
(4)             G.add_edge(1, 3)
(4)             paths = nx.all_simple_paths(G, 0, [2, 3])
(4)             assert {tuple(p) for p in paths} == {(0, 1, 2), (0, 1, 3)}
(0)         def test_all_simple_paths_source_target():
(4)             G = nx.path_graph(4)
(4)             assert list(nx.all_simple_paths(G, 1, 1)) == [[1]]
(0)         def test_all_simple_paths_cutoff():
(4)             G = nx.complete_graph(4)
```

```
106: (4)                paths = nx.all_simple_paths(G, 0, 1, cutoff=1)
107: (4)                assert {tuple(p) for p in paths} == {(0, 1)}
108: (4)                paths = nx.all_simple_paths(G, 0, 1, cutoff=2)
109: (4)                assert {tuple(p) for p in paths} == {(0, 1), (0, 2, 1), (0, 3, 1)}
110: (0)            def test_all_simple_paths_on_non_trivial_graph():
111: (4)                """you may need to draw this graph to make sure it is reasonable"""
112: (4)                G = nx.path_graph(5, create_using=nx.DiGraph())
113: (4)                G.add_edges_from([(0, 5), (1, 5), (1, 3), (5, 4), (4, 2), (4, 3)])
114: (4)                paths = nx.all_simple_paths(G, 1, [2, 3])
115: (4)                assert {tuple(p) for p in paths} == {
116: (8)                    (1, 2),
117: (8)                    (1, 3, 4, 2),
118: (8)                    (1, 5, 4, 2),
119: (8)                    (1, 3),
120: (8)                    (1, 2, 3),
121: (8)                    (1, 5, 4, 3),
122: (8)                    (1, 5, 4, 2, 3),
123: (4)                }
124: (4)                paths = nx.all_simple_paths(G, 1, [2, 3], cutoff=3)
125: (4)                assert {tuple(p) for p in paths} == {
126: (8)                    (1, 2),
127: (8)                    (1, 3, 4, 2),
128: (8)                    (1, 5, 4, 2),
129: (8)                    (1, 3),
130: (8)                    (1, 2, 3),
131: (8)                    (1, 5, 4, 3),
132: (4)                }
133: (4)                paths = nx.all_simple_paths(G, 1, [2, 3], cutoff=2)
134: (4)                assert {tuple(p) for p in paths} == {(1, 2), (1, 3), (1, 2, 3)}
135: (0)            def test_all_simple_paths_multigraph():
136: (4)                G = nx.MultiGraph([(1, 2), (1, 2)])
137: (4)                assert list(nx.all_simple_paths(G, 1, 1)) == [[1]]
138: (4)                nx.add_path(G, [3, 1, 10, 2])
139: (4)                paths = list(nx.all_simple_paths(G, 1, 2))
140: (4)                assert len(paths) == 3
141: (4)                assert {tuple(p) for p in paths} == {(1, 2), (1, 2), (1, 10, 2)}
142: (0)            def test_all_simple_paths_multigraph_with_cutoff():
143: (4)                G = nx.MultiGraph([(1, 2), (1, 2), (1, 10), (10, 2)])
144: (4)                paths = list(nx.all_simple_paths(G, 1, 2, cutoff=1))
145: (4)                assert len(paths) == 2
146: (4)                assert {tuple(p) for p in paths} == {(1, 2), (1, 2)}
147: (4)                G = nx.MultiGraph([(0, 1), (0, 2)])
148: (4)                assert list(nx.all_simple_paths(G, 0, {1, 2}, cutoff=1)) == [[0, 1], [0,
2]]
149: (0)            def test_all_simple_paths_directed():
150: (4)                G = nx.DiGraph()
151: (4)                nx.add_path(G, [1, 2, 3])
152: (4)                nx.add_path(G, [3, 2, 1])
153: (4)                paths = nx.all_simple_paths(G, 1, 3)
154: (4)                assert {tuple(p) for p in paths} == {(1, 2, 3)}
155: (0)            def test_all_simple_paths_empty():
156: (4)                G = nx.path_graph(4)
157: (4)                paths = nx.all_simple_paths(G, 0, 3, cutoff=2)
158: (4)                assert list(paths) == []
159: (0)            def test_all_simple_paths_corner_cases():
160: (4)                assert list(nx.all_simple_paths(nx.empty_graph(2), 0, 0)) == [[0]]
161: (4)                assert list(nx.all_simple_paths(nx.empty_graph(2), 0, 1)) == []
162: (4)                assert list(nx.all_simple_paths(nx.path_graph(9), 0, 8, 0)) == []
163: (0)            def test_all_simple_paths_source_in_targets():
164: (4)                G = nx.path_graph(3)
165: (4)                assert list(nx.all_simple_paths(G, 0, {0, 1, 2})) == [[0], [0, 1], [0, 1,
2]]
166: (0)            def hamiltonian_path(G, source):
167: (4)                source = arbitrary_element(G)
168: (4)                neighbors = set(G[source]) - {source}
169: (4)                n = len(G)
170: (4)                for target in neighbors:
171: (8)                    for path in nx.all_simple_paths(G, source, target):
172: (12)                       if len(path) == n:
```

```
(16)                            yield path
(0)             def test_hamiltonian_path():
(4)                 from itertools import permutations
(4)                 G = nx.complete_graph(4)
(4)                 paths = [list(p) for p in hamiltonian_path(G, 0)]
(4)                 exact = [[0] + list(p) for p in permutations([1, 2, 3], 3)]
(4)                 assert sorted(paths) == sorted(exact)
(0)             def test_cutoff_zero():
(4)                 G = nx.complete_graph(4)
(4)                 paths = nx.all_simple_paths(G, 0, 3, cutoff=0)
(4)                 assert [list(p) for p in paths] == []
(4)                 paths = nx.all_simple_paths(nx.MultiGraph(G), 0, 3, cutoff=0)
(4)                 assert [list(p) for p in paths] == []
(0)             def test_source_missing():
(4)                 with pytest.raises(nx.NodeNotFound):
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [1, 2, 3])
(8)                     list(nx.all_simple_paths(nx.MultiGraph(G), 0, 3))
(0)             def test_target_missing():
(4)                 with pytest.raises(nx.NodeNotFound):
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [1, 2, 3])
(8)                     list(nx.all_simple_paths(nx.MultiGraph(G), 1, 4))
(0)             def test_all_simple_edge_paths():
(4)                 G = nx.path_graph(4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, 3)
(4)                 assert {tuple(p) for p in paths} == {((0, 1), (1, 2), (2, 3))}
(0)             def test_all_simple_edge_paths_empty_path():
(4)                 G = nx.empty_graph(1)
(4)                 assert list(nx.all_simple_edge_paths(G, 0, 0)) == [[]]
(0)             def test_all_simple_edge_paths_with_two_targets_emits_two_paths():
(4)                 G = nx.path_graph(4)
(4)                 G.add_edge(2, 4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, [3, 4])
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((0, 1), (1, 2), (2, 3)),
(8)                     ((0, 1), (1, 2), (2, 4)),
(4)                 }
(0)             def test_digraph_all_simple_edge_paths_with_two_targets_emits_two_paths():
(4)                 G = nx.path_graph(4, create_using=nx.DiGraph())
(4)                 G.add_edge(2, 4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, [3, 4])
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((0, 1), (1, 2), (2, 3)),
(8)                     ((0, 1), (1, 2), (2, 4)),
(4)                 }
(0)             def test_all_simple_edge_paths_with_two_targets_cutoff():
(4)                 G = nx.path_graph(4)
(4)                 G.add_edge(2, 4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, [3, 4], cutoff=3)
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((0, 1), (1, 2), (2, 3)),
(8)                     ((0, 1), (1, 2), (2, 4)),
(4)                 }
(0)             def test_digraph_all_simple_edge_paths_with_two_targets_cutoff():
(4)                 G = nx.path_graph(4, create_using=nx.DiGraph())
(4)                 G.add_edge(2, 4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, [3, 4], cutoff=3)
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((0, 1), (1, 2), (2, 3)),
(8)                     ((0, 1), (1, 2), (2, 4)),
(4)                 }
(0)             def test_all_simple_edge_paths_with_two_targets_in_line_emits_two_paths():
(4)                 G = nx.path_graph(4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, [2, 3])
(4)                 assert {tuple(p) for p in paths} == {((0, 1), (1, 2)), ((0, 1), (1, 2),
(2, 3))}
(0)             def test_all_simple_edge_paths_ignores_cycle():
(4)                 G = nx.cycle_graph(3, create_using=nx.DiGraph())
```

```
(4)                 G.add_edge(1, 3)
(4)                 paths = nx.all_simple_edge_paths(G, 0, 3)
(4)                 assert {tuple(p) for p in paths} == {((0, 1), (1, 3))}
(0)             def
test_all_simple_edge_paths_with_two_targets_inside_cycle_emits_two_paths():
(4)                 G = nx.cycle_graph(3, create_using=nx.DiGraph())
(4)                 G.add_edge(1, 3)
(4)                 paths = nx.all_simple_edge_paths(G, 0, [2, 3])
(4)                 assert {tuple(p) for p in paths} == {((0, 1), (1, 2)), ((0, 1), (1, 3))}
(0)             def test_all_simple_edge_paths_source_target():
(4)                 G = nx.path_graph(4)
(4)                 paths = nx.all_simple_edge_paths(G, 1, 1)
(4)                 assert list(paths) == [[]]
(0)             def test_all_simple_edge_paths_cutoff():
(4)                 G = nx.complete_graph(4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, 1, cutoff=1)
(4)                 assert {tuple(p) for p in paths} == {((0, 1),)}
(4)                 paths = nx.all_simple_edge_paths(G, 0, 1, cutoff=2)
(4)                 assert {tuple(p) for p in paths} == {((0, 1),), ((0, 2), (2, 1)), ((0, 3),
(3, 1))}
(0)             def test_all_simple_edge_paths_on_non_trivial_graph():
(4)                 """you may need to draw this graph to make sure it is reasonable"""
(4)                 G = nx.path_graph(5, create_using=nx.DiGraph())
(4)                 G.add_edges_from([(0, 5), (1, 5), (1, 3), (5, 4), (4, 2), (4, 3)])
(4)                 paths = nx.all_simple_edge_paths(G, 1, [2, 3])
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((1, 2),),
(8)                     ((1, 3), (3, 4), (4, 2)),
(8)                     ((1, 5), (5, 4), (4, 2)),
(8)                     ((1, 3),),
(8)                     ((1, 2), (2, 3)),
(8)                     ((1, 5), (5, 4), (4, 3)),
(8)                     ((1, 5), (5, 4), (4, 2), (2, 3)),
(4)                 }
(4)                 paths = nx.all_simple_edge_paths(G, 1, [2, 3], cutoff=3)
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((1, 2),),
(8)                     ((1, 3), (3, 4), (4, 2)),
(8)                     ((1, 5), (5, 4), (4, 2)),
(8)                     ((1, 3),),
(8)                     ((1, 2), (2, 3)),
(8)                     ((1, 5), (5, 4), (4, 3)),
(4)                 }
(4)                 paths = nx.all_simple_edge_paths(G, 1, [2, 3], cutoff=2)
(4)                 assert {tuple(p) for p in paths} == {((1, 2),), ((1, 3),), ((1, 2), (2,
3))}
(0)             def test_all_simple_edge_paths_multigraph():
(4)                 G = nx.MultiGraph([(1, 2), (1, 2)])
(4)                 paths = nx.all_simple_edge_paths(G, 1, 1)
(4)                 assert list(paths) == [[]]
(4)                 nx.add_path(G, [3, 1, 10, 2])
(4)                 paths = list(nx.all_simple_edge_paths(G, 1, 2))
(4)                 assert len(paths) == 3
(4)                 assert {tuple(p) for p in paths} == {
(8)                     ((1, 2, 0),),
(8)                     ((1, 2, 1),),
(8)                     ((1, 10, 0), (10, 2, 0)),
(4)                 }
(0)             def test_all_simple_edge_paths_multigraph_with_cutoff():
(4)                 G = nx.MultiGraph([(1, 2), (1, 2), (1, 10), (10, 2)])
(4)                 paths = list(nx.all_simple_edge_paths(G, 1, 2, cutoff=1))
(4)                 assert len(paths) == 2
(4)                 assert {tuple(p) for p in paths} == {((1, 2, 0),), ((1, 2, 1),)}
(0)             def test_all_simple_edge_paths_directed():
(4)                 G = nx.DiGraph()
(4)                 nx.add_path(G, [1, 2, 3])
(4)                 nx.add_path(G, [3, 2, 1])
(4)                 paths = nx.all_simple_edge_paths(G, 1, 3)
(4)                 assert {tuple(p) for p in paths} == {((1, 2), (2, 3))}
```

```
(0)             def test_all_simple_edge_paths_empty():
(4)                 G = nx.path_graph(4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, 3, cutoff=2)
(4)                 assert list(paths) == []
(0)             def test_all_simple_edge_paths_corner_cases():
(4)                 assert list(nx.all_simple_edge_paths(nx.empty_graph(2), 0, 0)) == [[]]
(4)                 assert list(nx.all_simple_edge_paths(nx.empty_graph(2), 0, 1)) == []
(4)                 assert list(nx.all_simple_edge_paths(nx.path_graph(9), 0, 8, 0)) == []
(0)             def test_all_simple_edge_paths_ignores_self_loop():
(4)                 G = nx.Graph([(0, 0), (0, 1), (1, 1), (1, 2)])
(4)                 assert list(nx.all_simple_edge_paths(G, 0, 2)) == [[(0, 1), (1, 2)]]
(0)             def hamiltonian_edge_path(G, source):
(4)                 source = arbitrary_element(G)
(4)                 neighbors = set(G[source]) - {source}
(4)                 n = len(G)
(4)                 for target in neighbors:
(8)                     for path in nx.all_simple_edge_paths(G, source, target):
(12)                        if len(path) == n - 1:
(16)                            yield path
(0)             def test_hamiltonian__edge_path():
(4)                 from itertools import permutations
(4)                 G = nx.complete_graph(4)
(4)                 paths = hamiltonian_edge_path(G, 0)
(4)                 exact = [list(pairwise([0] + list(p))) for p in permutations([1, 2, 3],
3)]
(4)                 assert sorted(exact) == sorted(paths)
(0)             def test_edge_cutoff_zero():
(4)                 G = nx.complete_graph(4)
(4)                 paths = nx.all_simple_edge_paths(G, 0, 3, cutoff=0)
(4)                 assert [list(p) for p in paths] == []
(4)                 paths = nx.all_simple_edge_paths(nx.MultiGraph(G), 0, 3, cutoff=0)
(4)                 assert [list(p) for p in paths] == []
(0)             def test_edge_source_missing():
(4)                 with pytest.raises(nx.NodeNotFound):
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [1, 2, 3])
(8)                     list(nx.all_simple_edge_paths(nx.MultiGraph(G), 0, 3))
(0)             def test_edge_target_missing():
(4)                 with pytest.raises(nx.NodeNotFound):
(8)                     G = nx.Graph()
(8)                     nx.add_path(G, [1, 2, 3])
(8)                     list(nx.all_simple_edge_paths(nx.MultiGraph(G), 1, 4))
(0)             def test_shortest_simple_paths():
(4)                 G = cnlti(nx.grid_2d_graph(4, 4), first_label=1, ordering="sorted")
(4)                 paths = nx.shortest_simple_paths(G, 1, 12)
(4)                 assert next(paths) == [1, 2, 3, 4, 8, 12]
(4)                 assert next(paths) == [1, 5, 6, 7, 8, 12]
(4)                 assert [len(path) for path in nx.shortest_simple_paths(G, 1, 12)] ==
sorted(
(8)                     len(path) for path in nx.all_simple_paths(G, 1, 12)
(4)                 )
(0)             def test_shortest_simple_paths_singleton_path():
(4)                 G = nx.empty_graph(3)
(4)                 assert list(nx.shortest_simple_paths(G, 0, 0)) == [[0]]
(0)             def test_shortest_simple_paths_directed():
(4)                 G = nx.cycle_graph(7, create_using=nx.DiGraph())
(4)                 paths = nx.shortest_simple_paths(G, 0, 3)
(4)                 assert list(paths) == [[0, 1, 2, 3]]
(0)             def test_shortest_simple_paths_directed_with_weight_function():
(4)                 def cost(u, v, x):
(8)                     return 1
(4)                 G = cnlti(nx.grid_2d_graph(4, 4), first_label=1, ordering="sorted")
(4)                 paths = nx.shortest_simple_paths(G, 1, 12)
(4)                 assert next(paths) == [1, 2, 3, 4, 8, 12]
(4)                 assert next(paths) == [1, 5, 6, 7, 8, 12]
(4)                 assert [
(8)                     len(path) for path in nx.shortest_simple_paths(G, 1, 12, weight=cost)
(4)                 ] == sorted(len(path) for path in nx.all_simple_paths(G, 1, 12))
(0)             def test_shortest_simple_paths_with_weight_function():
```

```
(4)     def cost(u, v, x):
(8)         return 1
(4)     G = nx.cycle_graph(7, create_using=nx.DiGraph())
(4)     paths = nx.shortest_simple_paths(G, 0, 3, weight=cost)
(4)     assert list(paths) == [[0, 1, 2, 3]]
(0) def test_Greg_Bernstein():
(4)     g1 = nx.Graph()
(4)     g1.add_nodes_from(["N0", "N1", "N2", "N3", "N4"])
(4)     g1.add_edge("N4", "N1", weight=10.0, capacity=50, name="L5")
(4)     g1.add_edge("N4", "N0", weight=7.0, capacity=40, name="L4")
(4)     g1.add_edge("N0", "N1", weight=10.0, capacity=45, name="L1")
(4)     g1.add_edge("N3", "N0", weight=10.0, capacity=50, name="L0")
(4)     g1.add_edge("N2", "N3", weight=12.0, capacity=30, name="L2")
(4)     g1.add_edge("N1", "N2", weight=15.0, capacity=42, name="L3")
(4)     solution = [["N1", "N0", "N3"], ["N1", "N2", "N3"], ["N1", "N4", "N0",
"N3"]]
(4)     result = list(nx.shortest_simple_paths(g1, "N1", "N3", weight="weight"))
(4)     assert result == solution
(0) def test_weighted_shortest_simple_path():
(4)     def cost_func(path):
(8)         return sum(G.adj[u][v]["weight"] for (u, v) in zip(path, path[1:]))
(4)     G = nx.complete_graph(5)
(4)     weight = {(u, v): random.randint(1, 100) for (u, v) in G.edges()}
(4)     nx.set_edge_attributes(G, weight, "weight")
(4)     cost = 0
(4)     for path in nx.shortest_simple_paths(G, 0, 3, weight="weight"):
(8)         this_cost = cost_func(path)
(8)         assert cost <= this_cost
(8)         cost = this_cost
(0) def test_directed_weighted_shortest_simple_path():
(4)     def cost_func(path):
(8)         return sum(G.adj[u][v]["weight"] for (u, v) in zip(path, path[1:]))
(4)     G = nx.complete_graph(5)
(4)     G = G.to_directed()
(4)     weight = {(u, v): random.randint(1, 100) for (u, v) in G.edges()}
(4)     nx.set_edge_attributes(G, weight, "weight")
(4)     cost = 0
(4)     for path in nx.shortest_simple_paths(G, 0, 3, weight="weight"):
(8)         this_cost = cost_func(path)
(8)         assert cost <= this_cost
(8)         cost = this_cost
(0) def test_weighted_shortest_simple_path_issue2427():
(4)     G = nx.Graph()
(4)     G.add_edge("IN", "OUT", weight=2)
(4)     G.add_edge("IN", "A", weight=1)
(4)     G.add_edge("IN", "B", weight=2)
(4)     G.add_edge("B", "OUT", weight=2)
(4)     assert list(nx.shortest_simple_paths(G, "IN", "OUT", weight="weight")) ==
[
(8)         ["IN", "OUT"],
(8)         ["IN", "B", "OUT"],
(4)     ]
(4)     G = nx.Graph()
(4)     G.add_edge("IN", "OUT", weight=10)
(4)     G.add_edge("IN", "A", weight=1)
(4)     G.add_edge("IN", "B", weight=1)
(4)     G.add_edge("B", "OUT", weight=1)
(4)     assert list(nx.shortest_simple_paths(G, "IN", "OUT", weight="weight")) ==
[
(8)         ["IN", "B", "OUT"],
(8)         ["IN", "OUT"],
(4)     ]
(0) def test_directed_weighted_shortest_simple_path_issue2427():
(4)     G = nx.DiGraph()
(4)     G.add_edge("IN", "OUT", weight=2)
(4)     G.add_edge("IN", "A", weight=1)
(4)     G.add_edge("IN", "B", weight=2)
(4)     G.add_edge("B", "OUT", weight=2)
(4)     assert list(nx.shortest_simple_paths(G, "IN", "OUT", weight="weight")) ==
```

```
[
440: (8)                    ["IN", "OUT"],
441: (8)                    ["IN", "B", "OUT"],
442: (4)                ]
443: (4)                G = nx.DiGraph()
444: (4)                G.add_edge("IN", "OUT", weight=10)
445: (4)                G.add_edge("IN", "A", weight=1)
446: (4)                G.add_edge("IN", "B", weight=1)
447: (4)                G.add_edge("B", "OUT", weight=1)
448: (4)                assert list(nx.shortest_simple_paths(G, "IN", "OUT", weight="weight")) ==
[
449: (8)                    ["IN", "B", "OUT"],
450: (8)                    ["IN", "OUT"],
451: (4)                ]
452: (0)            def test_weight_name():
453: (4)                G = nx.cycle_graph(7)
454: (4)                nx.set_edge_attributes(G, 1, "weight")
455: (4)                nx.set_edge_attributes(G, 1, "foo")
456: (4)                G.adj[1][2]["foo"] = 7
457: (4)                paths = list(nx.shortest_simple_paths(G, 0, 3, weight="foo"))
458: (4)                solution = [[0, 6, 5, 4, 3], [0, 1, 2, 3]]
459: (4)                assert paths == solution
460: (0)            def test_ssp_source_missing():
461: (4)                with pytest.raises(nx.NodeNotFound):
462: (8)                    G = nx.Graph()
463: (8)                    nx.add_path(G, [1, 2, 3])
464: (8)                    list(nx.shortest_simple_paths(G, 0, 3))
465: (0)            def test_ssp_target_missing():
466: (4)                with pytest.raises(nx.NodeNotFound):
467: (8)                    G = nx.Graph()
468: (8)                    nx.add_path(G, [1, 2, 3])
469: (8)                    list(nx.shortest_simple_paths(G, 1, 4))
470: (0)            def test_ssp_multigraph():
471: (4)                with pytest.raises(nx.NetworkXNotImplemented):
472: (8)                    G = nx.MultiGraph()
473: (8)                    nx.add_path(G, [1, 2, 3])
474: (8)                    list(nx.shortest_simple_paths(G, 1, 4))
475: (0)            def test_ssp_source_missing2():
476: (4)                with pytest.raises(nx.NetworkXNoPath):
477: (8)                    G = nx.Graph()
478: (8)                    nx.add_path(G, [0, 1, 2])
479: (8)                    nx.add_path(G, [3, 4, 5])
480: (8)                    list(nx.shortest_simple_paths(G, 0, 3))
481: (0)            def test_bidirectional_shortest_path_restricted_cycle():
482: (4)                cycle = nx.cycle_graph(7)
483: (4)                length, path = _bidirectional_shortest_path(cycle, 0, 3)
484: (4)                assert path == [0, 1, 2, 3]
485: (4)                length, path = _bidirectional_shortest_path(cycle, 0, 3, ignore_nodes=[1])
486: (4)                assert path == [0, 6, 5, 4, 3]
487: (0)            def test_bidirectional_shortest_path_restricted_wheel():
488: (4)                wheel = nx.wheel_graph(6)
489: (4)                length, path = _bidirectional_shortest_path(wheel, 1, 3)
490: (4)                assert path in [[1, 0, 3], [1, 2, 3]]
491: (4)                length, path = _bidirectional_shortest_path(wheel, 1, 3, ignore_nodes=[0])
492: (4)                assert path == [1, 2, 3]
493: (4)                length, path = _bidirectional_shortest_path(wheel, 1, 3, ignore_nodes=[0,
2])
494: (4)                assert path == [1, 5, 4, 3]
495: (4)                length, path = _bidirectional_shortest_path(
496: (8)                    wheel, 1, 3, ignore_edges=[(1, 0), (5, 0), (2, 3)]
497: (4)                )
498: (4)                assert path in [[1, 2, 0, 3], [1, 5, 4, 3]]
499: (0)            def test_bidirectional_shortest_path_restricted_directed_cycle():
500: (4)                directed_cycle = nx.cycle_graph(7, create_using=nx.DiGraph())
501: (4)                length, path = _bidirectional_shortest_path(directed_cycle, 0, 3)
502: (4)                assert path == [0, 1, 2, 3]
503: (4)                pytest.raises(
504: (8)                    nx.NetworkXNoPath,
505: (8)                    _bidirectional_shortest_path,
```

```
506: (8)                    directed_cycle,
507: (8)                    0,
508: (8)                    3,
509: (8)                    ignore_nodes=[1],
510: (4)                )
511: (4)                length, path = _bidirectional_shortest_path(
512: (8)                    directed_cycle, 0, 3, ignore_edges=[(2, 1)]
513: (4)                )
514: (4)                assert path == [0, 1, 2, 3]
515: (4)                pytest.raises(
516: (8)                    nx.NetworkXNoPath,
517: (8)                    _bidirectional_shortest_path,
518: (8)                    directed_cycle,
519: (8)                    0,
520: (8)                    3,
521: (8)                    ignore_edges=[(1, 2)],
522: (4)                )
523: (0)            def test_bidirectional_shortest_path_ignore():
524: (4)                G = nx.Graph()
525: (4)                nx.add_path(G, [1, 2])
526: (4)                nx.add_path(G, [1, 3])
527: (4)                nx.add_path(G, [1, 4])
528: (4)                pytest.raises(
529: (8)                    nx.NetworkXNoPath, _bidirectional_shortest_path, G, 1, 2,
ignore_nodes=[1]
530: (4)                )
531: (4)                pytest.raises(
532: (8)                    nx.NetworkXNoPath, _bidirectional_shortest_path, G, 1, 2,
ignore_nodes=[2]
533: (4)                )
534: (4)                G = nx.Graph()
535: (4)                nx.add_path(G, [1, 3])
536: (4)                nx.add_path(G, [1, 4])
537: (4)                nx.add_path(G, [3, 2])
538: (4)                pytest.raises(
539: (8)                    nx.NetworkXNoPath, _bidirectional_shortest_path, G, 1, 2,
ignore_nodes=[1, 2]
540: (4)                )
541: (0)            def validate_path(G, s, t, soln_len, path):
542: (4)                assert path[0] == s
543: (4)                assert path[-1] == t
544: (4)                assert soln_len == sum(
545: (8)                    G[u][v].get("weight", 1) for u, v in zip(path[:-1], path[1:])
546: (4)                )
547: (0)            def validate_length_path(G, s, t, soln_len, length, path):
548: (4)                assert soln_len == length
549: (4)                validate_path(G, s, t, length, path)
550: (0)            def test_bidirectional_dijkstra_restricted():
551: (4)                XG = nx.DiGraph()
552: (4)                XG.add_weighted_edges_from(
553: (8)                    [
554: (12)                       ("s", "u", 10),
555: (12)                       ("s", "x", 5),
556: (12)                       ("u", "v", 1),
557: (12)                       ("u", "x", 2),
558: (12)                       ("v", "y", 1),
559: (12)                       ("x", "u", 3),
560: (12)                       ("x", "v", 5),
561: (12)                       ("x", "y", 2),
562: (12)                       ("y", "s", 7),
563: (12)                       ("y", "v", 6),
564: (8)                    ]
565: (4)                )
566: (4)                XG3 = nx.Graph()
567: (4)                XG3.add_weighted_edges_from(
568: (8)                    [[0, 1, 2], [1, 2, 12], [2, 3, 1], [3, 4, 5], [4, 5, 1], [5, 0, 10]]
569: (4)                )
570: (4)                validate_length_path(XG, "s", "v", 9, *_bidirectional_dijkstra(XG, "s",
"v"))
```

```
571: (4)                validate_length_path(
572: (8)                    XG, "s", "v", 10, *_bidirectional_dijkstra(XG, "s", "v", ignore_nodes=
["u"])
573: (4)                )
574: (4)                validate_length_path(
575: (8)                    XG,
576: (8)                    "s",
577: (8)                    "v",
578: (8)                    11,
579: (8)                    *_bidirectional_dijkstra(XG, "s", "v", ignore_edges=[("s", "x")]),
580: (4)                )
581: (4)                pytest.raises(
582: (8)                    nx.NetworkXNoPath,
583: (8)                    _bidirectional_dijkstra,
584: (8)                    XG,
585: (8)                    "s",
586: (8)                    "v",
587: (8)                    ignore_nodes=["u"],
588: (8)                    ignore_edges=[("s", "x")],
589: (4)                )
590: (4)                validate_length_path(XG3, 0, 3, 15, *_bidirectional_dijkstra(XG3, 0, 3))
591: (4)                validate_length_path(
592: (8)                    XG3, 0, 3, 16, *_bidirectional_dijkstra(XG3, 0, 3, ignore_nodes=[1])
593: (4)                )
594: (4)                validate_length_path(
595: (8)                    XG3, 0, 3, 16, *_bidirectional_dijkstra(XG3, 0, 3, ignore_edges=[(2,
3)])
596: (4)                )
597: (4)                pytest.raises(
598: (8)                    nx.NetworkXNoPath,
599: (8)                    _bidirectional_dijkstra,
600: (8)                    XG3,
601: (8)                    0,
602: (8)                    3,
603: (8)                    ignore_nodes=[1],
604: (8)                    ignore_edges=[(5, 4)],
605: (4)                )
606: (0)            def test_bidirectional_dijkstra_no_path():
607: (4)                with pytest.raises(nx.NetworkXNoPath):
608: (8)                    G = nx.Graph()
609: (8)                    nx.add_path(G, [1, 2, 3])
610: (8)                    nx.add_path(G, [4, 5, 6])
611: (8)                    _bidirectional_dijkstra(G, 1, 6)
612: (0)            def test_bidirectional_dijkstra_ignore():
613: (4)                G = nx.Graph()
614: (4)                nx.add_path(G, [1, 2, 10])
615: (4)                nx.add_path(G, [1, 3, 10])
616: (4)                pytest.raises(nx.NetworkXNoPath, _bidirectional_dijkstra, G, 1, 2,
ignore_nodes=[1])
617: (4)                pytest.raises(nx.NetworkXNoPath, _bidirectional_dijkstra, G, 1, 2,
ignore_nodes=[2])
618: (4)                pytest.raises(
619: (8)                    nx.NetworkXNoPath, _bidirectional_dijkstra, G, 1, 2, ignore_nodes=[1,
2]
620: (4)                )

----------------------------------------

File 348 - test_smallworld.py:

1: (0)              import pytest
2: (0)              pytest.importorskip("numpy")
3: (0)              import random
4: (0)              import networkx as nx
5: (0)              from networkx import lattice_reference, omega, random_reference, sigma
6: (0)              rng = 42
7: (0)              def test_random_reference():
8: (4)                  G = nx.connected_watts_strogatz_graph(50, 6, 0.1, seed=rng)
9: (4)                  Gr = random_reference(G, niter=1, seed=rng)
```

```
10: (4)                 C = nx.average_clustering(G)
11: (4)                 Cr = nx.average_clustering(Gr)
12: (4)                 assert C > Cr
13: (4)                 with pytest.raises(nx.NetworkXError):
14: (8)                     next(random_reference(nx.Graph()))
15: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
16: (8)                     next(random_reference(nx.DiGraph()))
17: (4)                 H = nx.Graph(((0, 1), (2, 3)))
18: (4)                 Hl = random_reference(H, niter=1, seed=rng)
19: (0)             def test_lattice_reference():
20: (4)                 G = nx.connected_watts_strogatz_graph(50, 6, 1, seed=rng)
21: (4)                 Gl = lattice_reference(G, niter=1, seed=rng)
22: (4)                 L = nx.average_shortest_path_length(G)
23: (4)                 Ll = nx.average_shortest_path_length(Gl)
24: (4)                 assert Ll > L
25: (4)                 pytest.raises(nx.NetworkXError, lattice_reference, nx.Graph())
26: (4)                 pytest.raises(nx.NetworkXNotImplemented, lattice_reference, nx.DiGraph())
27: (4)                 H = nx.Graph(((0, 1), (2, 3)))
28: (4)                 Hl = lattice_reference(H, niter=1)
29: (0)             def test_sigma():
30: (4)                 Gs = nx.connected_watts_strogatz_graph(50, 6, 0.1, seed=rng)
31: (4)                 Gr = nx.connected_watts_strogatz_graph(50, 6, 1, seed=rng)
32: (4)                 sigmas = sigma(Gs, niter=1, nrand=2, seed=rng)
33: (4)                 sigmar = sigma(Gr, niter=1, nrand=2, seed=rng)
34: (4)                 assert sigmar < sigmas
35: (0)             def test_omega():
36: (4)                 Gl = nx.connected_watts_strogatz_graph(50, 6, 0, seed=rng)
37: (4)                 Gr = nx.connected_watts_strogatz_graph(50, 6, 1, seed=rng)
38: (4)                 Gs = nx.connected_watts_strogatz_graph(50, 6, 0.1, seed=rng)
39: (4)                 omegal = omega(Gl, niter=1, nrand=1, seed=rng)
40: (4)                 omegar = omega(Gr, niter=1, nrand=1, seed=rng)
41: (4)                 omegas = omega(Gs, niter=1, nrand=1, seed=rng)
42: (4)                 assert omegal < omegas and omegas < omegar
43: (4)                 G_barbell = nx.barbell_graph(5, 1)
44: (4)                 G_karate = nx.karate_club_graph()
45: (4)                 omega_barbell = nx.omega(G_barbell)
46: (4)                 omega_karate = nx.omega(G_karate, nrand=2)
47: (4)                 omegas = (omegal, omegar, omegas, omega_barbell, omega_karate)
48: (4)                 for o in omegas:
49: (8)                     assert -1 <= o <= 1
50: (0)             @pytest.mark.parametrize("f", (nx.random_reference, nx.lattice_reference))
51: (0)             def test_graph_no_edges(f):
52: (4)                 G = nx.Graph()
53: (4)                 G.add_nodes_from([0, 1, 2, 3])
54: (4)                 with pytest.raises(nx.NetworkXError, match="Graph has fewer that 2
edges"):
55: (8)                     f(G)

----------------------------------------

File 349 - test_smetric.py:

1: (0)              import warnings
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              def test_smetric():
5: (4)                  g = nx.Graph()
6: (4)                  g.add_edge(1, 2)
7: (4)                  g.add_edge(2, 3)
8: (4)                  g.add_edge(2, 4)
9: (4)                  g.add_edge(1, 4)
10: (4)                 sm = nx.s_metric(g, normalized=False)
11: (4)                 assert sm == 19.0
12: (0)             def test_normalized_deprecation_warning():
13: (4)                 """Test that a deprecation warning is raised when s_metric is called with
14: (4)                 a `normalized` kwarg."""
15: (4)                 G = nx.cycle_graph(7)
16: (4)                 with warnings.catch_warnings():
17: (8)                     warnings.simplefilter("error")  # Fail the test if warning caught
```

```
18: (8)                     assert nx.s_metric(G) == 28
19: (4)                 with pytest.deprecated_call():
20: (8)                     nx.s_metric(G, normalized=True)
21: (4)                 with pytest.raises(TypeError):
22: (8)                     nx.s_metric(G, normalize=True)
```

File 350 - test_sparsifiers.py:

```
1: (0)              """Unit tests for the sparsifier computation functions."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import py_random_state
5: (0)              _seed = 2
6: (0)              def _test_spanner(G, spanner, stretch, weight=None):
7: (4)                  """Test whether a spanner is valid.
8: (4)                  This function tests whether the given spanner is a subgraph of the
9: (4)                  given graph G with the same node set. It also tests for all shortest
10: (4)                 paths whether they adhere to the given stretch.
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 G : NetworkX graph
14: (8)                     The original graph for which the spanner was constructed.
15: (4)                 spanner : NetworkX graph
16: (8)                     The spanner to be tested.
17: (4)                 stretch : float
18: (8)                     The proclaimed stretch of the spanner.
19: (4)                 weight : object
20: (8)                     The edge attribute to use as distance.
21: (4)                 """
22: (4)                 assert set(G.nodes()) == set(spanner.nodes())
23: (4)                 for u, v in spanner.edges():
24: (8)                     assert G.has_edge(u, v)
25: (8)                     if weight:
26: (12)                        assert spanner[u][v][weight] == G[u][v][weight]
27: (4)                 original_length = dict(nx.shortest_path_length(G, weight=weight))
28: (4)                 spanner_length = dict(nx.shortest_path_length(spanner, weight=weight))
29: (4)                 for u in G.nodes():
30: (8)                     for v in G.nodes():
31: (12)                        if u in original_length and v in original_length[u]:
32: (16)                            assert spanner_length[u][v] <= stretch * original_length[u][v]
33: (0)             @py_random_state(1)
34: (0)             def _assign_random_weights(G, seed=None):
35: (4)                 """Assigns random weights to the edges of a graph.
36: (4)                 Parameters
37: (4)                 ----------
38: (4)                 G : NetworkX graph
39: (8)                     The original graph for which the spanner was constructed.
40: (4)                 seed : integer, random_state, or None (default)
41: (8)                     Indicator of random number generation state.
42: (8)                     See :ref:`Randomness<randomness>`.
43: (4)                 """
44: (4)                 for u, v in G.edges():
45: (8)                     G[u][v]["weight"] = seed.random()
46: (0)             def test_spanner_trivial():
47: (4)                 """Test a trivial spanner with stretch 1."""
48: (4)                 G = nx.complete_graph(20)
49: (4)                 spanner = nx.spanner(G, 1, seed=_seed)
50: (4)                 for u, v in G.edges:
51: (8)                     assert spanner.has_edge(u, v)
52: (0)             def test_spanner_unweighted_complete_graph():
53: (4)                 """Test spanner construction on a complete unweighted graph."""
54: (4)                 G = nx.complete_graph(20)
55: (4)                 spanner = nx.spanner(G, 4, seed=_seed)
56: (4)                 _test_spanner(G, spanner, 4)
57: (4)                 spanner = nx.spanner(G, 10, seed=_seed)
58: (4)                 _test_spanner(G, spanner, 10)
59: (0)             def test_spanner_weighted_complete_graph():
```

```
60: (4)                 """Test spanner construction on a complete weighted graph."""
61: (4)                 G = nx.complete_graph(20)
62: (4)                 _assign_random_weights(G, seed=_seed)
63: (4)                 spanner = nx.spanner(G, 4, weight="weight", seed=_seed)
64: (4)                 _test_spanner(G, spanner, 4, weight="weight")
65: (4)                 spanner = nx.spanner(G, 10, weight="weight", seed=_seed)
66: (4)                 _test_spanner(G, spanner, 10, weight="weight")
67: (0)             def test_spanner_unweighted_gnp_graph():
68: (4)                 """Test spanner construction on an unweighted gnp graph."""
69: (4)                 G = nx.gnp_random_graph(20, 0.4, seed=_seed)
70: (4)                 spanner = nx.spanner(G, 4, seed=_seed)
71: (4)                 _test_spanner(G, spanner, 4)
72: (4)                 spanner = nx.spanner(G, 10, seed=_seed)
73: (4)                 _test_spanner(G, spanner, 10)
74: (0)             def test_spanner_weighted_gnp_graph():
75: (4)                 """Test spanner construction on an weighted gnp graph."""
76: (4)                 G = nx.gnp_random_graph(20, 0.4, seed=_seed)
77: (4)                 _assign_random_weights(G, seed=_seed)
78: (4)                 spanner = nx.spanner(G, 4, weight="weight", seed=_seed)
79: (4)                 _test_spanner(G, spanner, 4, weight="weight")
80: (4)                 spanner = nx.spanner(G, 10, weight="weight", seed=_seed)
81: (4)                 _test_spanner(G, spanner, 10, weight="weight")
82: (0)             def test_spanner_unweighted_disconnected_graph():
83: (4)                 """Test spanner construction on a disconnected graph."""
84: (4)                 G = nx.disjoint_union(nx.complete_graph(10), nx.complete_graph(10))
85: (4)                 spanner = nx.spanner(G, 4, seed=_seed)
86: (4)                 _test_spanner(G, spanner, 4)
87: (4)                 spanner = nx.spanner(G, 10, seed=_seed)
88: (4)                 _test_spanner(G, spanner, 10)
89: (0)             def test_spanner_invalid_stretch():
90: (4)                 """Check whether an invalid stretch is caught."""
91: (4)                 with pytest.raises(ValueError):
92: (8)                     G = nx.empty_graph()
93: (8)                     nx.spanner(G, 0)

----------------------------------------

File 351 - test_structuralholes.py:

1: (0)              """Unit tests for the :mod:`networkx.algorithms.structuralholes` module."""
2: (0)              import math
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.classes.tests import dispatch_interface
6: (0)              class TestStructuralHoles:
7: (4)                  """Unit tests for computing measures of structural holes.
8: (4)                  The expected values for these functions were originally computed using the
9: (4)                  proprietary software `UCINET`_ and the free software `IGraph`_ , and then
10: (4)                 computed by hand to make sure that the results are correct.
11: (4)                 .. _UCINET: https://sites.google.com/site/ucinetsoftware/home
12: (4)                 .. _IGraph: http://igraph.org/
13: (4)                 """
14: (4)                 def setup_method(self):
15: (8)                     self.D = nx.DiGraph()
16: (8)                     self.D.add_edges_from([(0, 1), (0, 2), (1, 0), (2, 1)])
17: (8)                     self.D_weights = {(0, 1): 2, (0, 2): 2, (1, 0): 1, (2, 1): 1}
18: (8)                     self.G = nx.Graph()
19: (8)                     self.G.add_edges_from(
20: (12)                        [
21: (16)                            ("A", "B"),
22: (16)                            ("A", "F"),
23: (16)                            ("A", "G"),
24: (16)                            ("A", "E"),
25: (16)                            ("E", "G"),
26: (16)                            ("F", "G"),
27: (16)                            ("B", "G"),
28: (16)                            ("B", "D"),
29: (16)                            ("D", "G"),
30: (16)                            ("G", "C"),
```

```
(12)                        ]
(8)                     )
(8)                     self.G_weights = {
(12)                        ("A", "B"): 2,
(12)                        ("A", "F"): 3,
(12)                        ("A", "G"): 5,
(12)                        ("A", "E"): 2,
(12)                        ("E", "G"): 8,
(12)                        ("F", "G"): 3,
(12)                        ("B", "G"): 4,
(12)                        ("B", "D"): 1,
(12)                        ("D", "G"): 3,
(12)                        ("G", "C"): 10,
(8)                     }
(4)                 @pytest.mark.parametrize("wrapper", [lambda x: x,
dispatch_interface.convert])
(4)                 def test_constraint_directed(self, wrapper):
(8)                     constraint = nx.constraint(wrapper(self.D))
(8)                     assert constraint[0] == pytest.approx(1.003, abs=1e-3)
(8)                     assert constraint[1] == pytest.approx(1.003, abs=1e-3)
(8)                     assert constraint[2] == pytest.approx(1.389, abs=1e-3)
(4)                 def test_effective_size_directed(self):
(8)                     effective_size = nx.effective_size(self.D)
(8)                     assert effective_size[0] == pytest.approx(1.167, abs=1e-3)
(8)                     assert effective_size[1] == pytest.approx(1.167, abs=1e-3)
(8)                     assert effective_size[2] == pytest.approx(1, abs=1e-3)
(4)                 def test_constraint_weighted_directed(self):
(8)                     D = self.D.copy()
(8)                     nx.set_edge_attributes(D, self.D_weights, "weight")
(8)                     constraint = nx.constraint(D, weight="weight")
(8)                     assert constraint[0] == pytest.approx(0.840, abs=1e-3)
(8)                     assert constraint[1] == pytest.approx(1.143, abs=1e-3)
(8)                     assert constraint[2] == pytest.approx(1.378, abs=1e-3)
(4)                 def test_effective_size_weighted_directed(self):
(8)                     D = self.D.copy()
(8)                     nx.set_edge_attributes(D, self.D_weights, "weight")
(8)                     effective_size = nx.effective_size(D, weight="weight")
(8)                     assert effective_size[0] == pytest.approx(1.567, abs=1e-3)
(8)                     assert effective_size[1] == pytest.approx(1.083, abs=1e-3)
(8)                     assert effective_size[2] == pytest.approx(1, abs=1e-3)
(4)                 def test_constraint_undirected(self):
(8)                     constraint = nx.constraint(self.G)
(8)                     assert constraint["G"] == pytest.approx(0.400, abs=1e-3)
(8)                     assert constraint["A"] == pytest.approx(0.595, abs=1e-3)
(8)                     assert constraint["C"] == pytest.approx(1, abs=1e-3)
(4)                 def test_effective_size_undirected_borgatti(self):
(8)                     effective_size = nx.effective_size(self.G)
(8)                     assert effective_size["G"] == pytest.approx(4.67, abs=1e-2)
(8)                     assert effective_size["A"] == pytest.approx(2.50, abs=1e-2)
(8)                     assert effective_size["C"] == pytest.approx(1, abs=1e-2)
(4)                 def test_effective_size_undirected(self):
(8)                     G = self.G.copy()
(8)                     nx.set_edge_attributes(G, 1, "weight")
(8)                     effective_size = nx.effective_size(G, weight="weight")
(8)                     assert effective_size["G"] == pytest.approx(4.67, abs=1e-2)
(8)                     assert effective_size["A"] == pytest.approx(2.50, abs=1e-2)
(8)                     assert effective_size["C"] == pytest.approx(1, abs=1e-2)
(4)                 def test_constraint_weighted_undirected(self):
(8)                     G = self.G.copy()
(8)                     nx.set_edge_attributes(G, self.G_weights, "weight")
(8)                     constraint = nx.constraint(G, weight="weight")
(8)                     assert constraint["G"] == pytest.approx(0.299, abs=1e-3)
(8)                     assert constraint["A"] == pytest.approx(0.795, abs=1e-3)
(8)                     assert constraint["C"] == pytest.approx(1, abs=1e-3)
(4)                 def test_effective_size_weighted_undirected(self):
(8)                     G = self.G.copy()
(8)                     nx.set_edge_attributes(G, self.G_weights, "weight")
(8)                     effective_size = nx.effective_size(G, weight="weight")
(8)                     assert effective_size["G"] == pytest.approx(5.47, abs=1e-2)
```

```
99: (8)                     assert effective_size["A"] == pytest.approx(2.47, abs=1e-2)
100: (8)                    assert effective_size["C"] == pytest.approx(1, abs=1e-2)
101: (4)                def test_constraint_isolated(self):
102: (8)                    G = self.G.copy()
103: (8)                    G.add_node(1)
104: (8)                    constraint = nx.constraint(G)
105: (8)                    assert math.isnan(constraint[1])
106: (4)                def test_effective_size_isolated(self):
107: (8)                    G = self.G.copy()
108: (8)                    G.add_node(1)
109: (8)                    nx.set_edge_attributes(G, self.G_weights, "weight")
110: (8)                    effective_size = nx.effective_size(G, weight="weight")
111: (8)                    assert math.isnan(effective_size[1])
112: (4)                def test_effective_size_borgatti_isolated(self):
113: (8)                    G = self.G.copy()
114: (8)                    G.add_node(1)
115: (8)                    effective_size = nx.effective_size(G)
116: (8)                    assert math.isnan(effective_size[1])

----------------------------------------

File 352 - test_summarization.py:

1: (0)              """
2: (0)              Unit tests for dedensification and graph summarization
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              class TestDirectedDedensification:
7: (4)                  def build_original_graph(self):
8: (8)                      original_matrix = [
9: (12)                         ("1", "BC"),
10: (12)                        ("2", "ABC"),
11: (12)                        ("3", ["A", "B", "6"]),
12: (12)                        ("4", "ABC"),
13: (12)                        ("5", "AB"),
14: (12)                        ("6", ["5"]),
15: (12)                        ("A", ["6"]),
16: (8)                     ]
17: (8)                     graph = nx.DiGraph()
18: (8)                     for source, targets in original_matrix:
19: (12)                        for target in targets:
20: (16)                            graph.add_edge(source, target)
21: (8)                     return graph
22: (4)                 def build_compressed_graph(self):
23: (8)                     compressed_matrix = [
24: (12)                        ("1", "BC"),
25: (12)                        ("2", ["ABC"]),
26: (12)                        ("3", ["A", "B", "6"]),
27: (12)                        ("4", ["ABC"]),
28: (12)                        ("5", "AB"),
29: (12)                        ("6", ["5"]),
30: (12)                        ("A", ["6"]),
31: (12)                        ("ABC", "ABC"),
32: (8)                     ]
33: (8)                     compressed_graph = nx.DiGraph()
34: (8)                     for source, targets in compressed_matrix:
35: (12)                        for target in targets:
36: (16)                            compressed_graph.add_edge(source, target)
37: (8)                     return compressed_graph
38: (4)                 def test_empty(self):
39: (8)                     """
40: (8)                     Verify that an empty directed graph results in no compressor nodes
41: (8)                     """
42: (8)                     G = nx.DiGraph()
43: (8)                     compressed_graph, c_nodes = nx.dedensify(G, threshold=2)
44: (8)                     assert c_nodes == set()
45: (4)                 @staticmethod
46: (4)                 def densify(G, compressor_nodes, copy=True):
```

```
47: (8)                     """
48: (8)                     Reconstructs the original graph from a dedensified, directed graph
49: (8)                     Parameters
50: (8)                     ----------
51: (8)                     G: dedensified graph
52: (11)                       A networkx graph
53: (8)                     compressor_nodes: iterable
54: (11)                       Iterable of compressor nodes in the dedensified graph
55: (8)                     inplace: bool, optional (default: False)
56: (11)                       Indicates if densification should be done inplace
57: (8)                     Returns
58: (8)                     -------
59: (8)                     G: graph
60: (11)                       A densified networkx graph
61: (8)                     """
62: (8)                     if copy:
63: (12)                        G = G.copy()
64: (8)                     for compressor_node in compressor_nodes:
65: (12)                        all_neighbors = set(nx.all_neighbors(G, compressor_node))
66: (12)                        out_neighbors = set(G.neighbors(compressor_node))
67: (12)                        for out_neighbor in out_neighbors:
68: (16)                            G.remove_edge(compressor_node, out_neighbor)
69: (12)                        in_neighbors = all_neighbors - out_neighbors
70: (12)                        for in_neighbor in in_neighbors:
71: (16)                            G.remove_edge(in_neighbor, compressor_node)
72: (16)                            for out_neighbor in out_neighbors:
73: (20)                                G.add_edge(in_neighbor, out_neighbor)
74: (12)                        G.remove_node(compressor_node)
75: (8)                     return G
76: (4)                 def setup_method(self):
77: (8)                     self.c_nodes = ("ABC",)
78: (4)                 def test_dedensify_edges(self):
79: (8)                     """
80: (8)                     Verifies that dedensify produced the correct edges to/from compressor
81: (8)                     nodes in a directed graph
82: (8)                     """
83: (8)                     G = self.build_original_graph()
84: (8)                     compressed_G = self.build_compressed_graph()
85: (8)                     compressed_graph, c_nodes = nx.dedensify(G, threshold=2)
86: (8)                     for s, t in compressed_graph.edges():
87: (12)                        o_s = "".join(sorted(s))
88: (12)                        o_t = "".join(sorted(t))
89: (12)                        compressed_graph_exists = compressed_graph.has_edge(s, t)
90: (12)                        verified_compressed_exists = compressed_G.has_edge(o_s, o_t)
91: (12)                        assert compressed_graph_exists == verified_compressed_exists
92: (8)                     assert len(c_nodes) == len(self.c_nodes)
93: (4)                 def test_dedensify_edge_count(self):
94: (8)                     """
95: (8)                     Verifies that dedensify produced the correct number of compressor
nodes
96: (8)                     in a directed graph
97: (8)                     """
98: (8)                     G = self.build_original_graph()
99: (8)                     original_edge_count = len(G.edges())
100: (8)                    c_G, c_nodes = nx.dedensify(G, threshold=2)
101: (8)                    compressed_edge_count = len(c_G.edges())
102: (8)                    assert compressed_edge_count <= original_edge_count
103: (8)                    compressed_G = self.build_compressed_graph()
104: (8)                    assert compressed_edge_count == len(compressed_G.edges())
105: (4)                def test_densify_edges(self):
106: (8)                    """
107: (8)                    Verifies that densification produces the correct edges from the
108: (8)                    original directed graph
109: (8)                    """
110: (8)                    compressed_G = self.build_compressed_graph()
111: (8)                    original_graph = self.densify(compressed_G, self.c_nodes, copy=True)
112: (8)                    G = self.build_original_graph()
113: (8)                    for s, t in G.edges():
114: (12)                       assert G.has_edge(s, t) == original_graph.has_edge(s, t)
```

```
115: (4)                def test_densify_edge_count(self):
116: (8)                    """
117: (8)                    Verifies that densification produces the correct number of edges in
the
118: (8)                    original directed graph
119: (8)                    """
120: (8)                    compressed_G = self.build_compressed_graph()
121: (8)                    compressed_edge_count = len(compressed_G.edges())
122: (8)                    original_graph = self.densify(compressed_G, self.c_nodes)
123: (8)                    original_edge_count = len(original_graph.edges())
124: (8)                    assert compressed_edge_count <= original_edge_count
125: (8)                    G = self.build_original_graph()
126: (8)                    assert original_edge_count == len(G.edges())
127: (0)            class TestUnDirectedDedensification:
128: (4)                def build_original_graph(self):
129: (8)                    """
130: (8)                    Builds graph shown in the original research paper
131: (8)                    """
132: (8)                    original_matrix = [
133: (12)                       ("1", "CB"),
134: (12)                       ("2", "ABC"),
135: (12)                       ("3", ["A", "B", "6"]),
136: (12)                       ("4", "ABC"),
137: (12)                       ("5", "AB"),
138: (12)                       ("6", ["5"]),
139: (12)                       ("A", ["6"]),
140: (8)                    ]
141: (8)                    graph = nx.Graph()
142: (8)                    for source, targets in original_matrix:
143: (12)                       for target in targets:
144: (16)                           graph.add_edge(source, target)
145: (8)                    return graph
146: (4)                def test_empty(self):
147: (8)                    """
148: (8)                    Verify that an empty undirected graph results in no compressor nodes
149: (8)                    """
150: (8)                    G = nx.Graph()
151: (8)                    compressed_G, c_nodes = nx.dedensify(G, threshold=2)
152: (8)                    assert c_nodes == set()
153: (4)                def setup_method(self):
154: (8)                    self.c_nodes = ("6AB", "ABC")
155: (4)                def build_compressed_graph(self):
156: (8)                    compressed_matrix = [
157: (12)                       ("1", ["B", "C"]),
158: (12)                       ("2", ["ABC"]),
159: (12)                       ("3", ["6AB"]),
160: (12)                       ("4", ["ABC"]),
161: (12)                       ("5", ["6AB"]),
162: (12)                       ("6", ["6AB", "A"]),
163: (12)                       ("A", ["6AB", "ABC"]),
164: (12)                       ("B", ["ABC", "6AB"]),
165: (12)                       ("C", ["ABC"]),
166: (8)                    ]
167: (8)                    compressed_graph = nx.Graph()
168: (8)                    for source, targets in compressed_matrix:
169: (12)                       for target in targets:
170: (16)                           compressed_graph.add_edge(source, target)
171: (8)                    return compressed_graph
172: (4)                def test_dedensify_edges(self):
173: (8)                    """
174: (8)                    Verifies that dedensify produced correct compressor nodes and the
175: (8)                    correct edges to/from the compressor nodes in an undirected graph
176: (8)                    """
177: (8)                    G = self.build_original_graph()
178: (8)                    c_G, c_nodes = nx.dedensify(G, threshold=2)
179: (8)                    v_compressed_G = self.build_compressed_graph()
180: (8)                    for s, t in c_G.edges():
181: (12)                       o_s = "".join(sorted(s))
182: (12)                       o_t = "".join(sorted(t))
```

```
183: (12)                        has_compressed_edge = c_G.has_edge(s, t)
184: (12)                        verified_has_compressed_edge = v_compressed_G.has_edge(o_s, o_t)
185: (12)                        assert has_compressed_edge == verified_has_compressed_edge
186: (8)                    assert len(c_nodes) == len(self.c_nodes)
187: (4)                def test_dedensify_edge_count(self):
188: (8)                    """
189: (8)                    Verifies that dedensify produced the correct number of edges in an
190: (8)                    undirected graph
191: (8)                    """
192: (8)                    G = self.build_original_graph()
193: (8)                    c_G, c_nodes = nx.dedensify(G, threshold=2, copy=True)
194: (8)                    compressed_edge_count = len(c_G.edges())
195: (8)                    verified_original_edge_count = len(G.edges())
196: (8)                    assert compressed_edge_count <= verified_original_edge_count
197: (8)                    verified_compressed_G = self.build_compressed_graph()
198: (8)                    verified_compressed_edge_count = len(verified_compressed_G.edges())
199: (8)                    assert compressed_edge_count == verified_compressed_edge_count
200: (0)            @pytest.mark.parametrize(
201: (4)                "graph_type", [nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph]
202: (0)            )
203: (0)            def test_summarization_empty(graph_type):
204: (4)                G = graph_type()
205: (4)                summary_graph = nx.snap_aggregation(G, node_attributes=("color",))
206: (4)                assert nx.is_isomorphic(summary_graph, G)
207: (0)            class AbstractSNAP:
208: (4)                node_attributes = ("color",)
209: (4)                def build_original_graph(self):
210: (8)                    pass
211: (4)                def build_summary_graph(self):
212: (8)                    pass
213: (4)                def test_summary_graph(self):
214: (8)                    original_graph = self.build_original_graph()
215: (8)                    summary_graph = self.build_summary_graph()
216: (8)                    relationship_attributes = ("type",)
217: (8)                    generated_summary_graph = nx.snap_aggregation(
218: (12)                       original_graph, self.node_attributes, relationship_attributes
219: (8)                    )
220: (8)                    relabeled_summary_graph =
self.deterministic_labels(generated_summary_graph)
221: (8)                    assert nx.is_isomorphic(summary_graph, relabeled_summary_graph)
222: (4)                def deterministic_labels(self, G):
223: (8)                    node_labels = list(G.nodes)
224: (8)                    node_labels = sorted(node_labels, key=lambda n: sorted(G.nodes[n]
["group"])[0])
225: (8)                    node_labels.sort()
226: (8)                    label_mapping = {}
227: (8)                    for index, node in enumerate(node_labels):
228: (12)                       label = "Supernode-%s" % index
229: (12)                       label_mapping[node] = label
230: (8)                    return nx.relabel_nodes(G, label_mapping)
231: (0)            class TestSNAPNoEdgeTypes(AbstractSNAP):
232: (4)                relationship_attributes = ()
233: (4)                def test_summary_graph(self):
234: (8)                    original_graph = self.build_original_graph()
235: (8)                    summary_graph = self.build_summary_graph()
236: (8)                    relationship_attributes = ("type",)
237: (8)                    generated_summary_graph = nx.snap_aggregation(
238: (12)                       original_graph, self.node_attributes
239: (8)                    )
240: (8)                    relabeled_summary_graph =
self.deterministic_labels(generated_summary_graph)
241: (8)                    assert nx.is_isomorphic(summary_graph, relabeled_summary_graph)
242: (4)                def build_original_graph(self):
243: (8)                    nodes = {
244: (12)                       "A": {"color": "Red"},
245: (12)                       "B": {"color": "Red"},
246: (12)                       "C": {"color": "Red"},
247: (12)                       "D": {"color": "Red"},
248: (12)                       "E": {"color": "Blue"},
```

```
(12)                        "F": {"color": "Blue"},
(12)                        "G": {"color": "Blue"},
(12)                        "H": {"color": "Blue"},
(12)                        "I": {"color": "Yellow"},
(12)                        "J": {"color": "Yellow"},
(12)                        "K": {"color": "Yellow"},
(12)                        "L": {"color": "Yellow"},
(8)                     }
(8)                     edges = [
(12)                        ("A", "B"),
(12)                        ("A", "C"),
(12)                        ("A", "E"),
(12)                        ("A", "I"),
(12)                        ("B", "D"),
(12)                        ("B", "J"),
(12)                        ("B", "F"),
(12)                        ("C", "G"),
(12)                        ("D", "H"),
(12)                        ("I", "J"),
(12)                        ("J", "K"),
(12)                        ("I", "L"),
(8)                     ]
(8)                     G = nx.Graph()
(8)                     for node in nodes:
(12)                        attributes = nodes[node]
(12)                        G.add_node(node, **attributes)
(8)                     for source, target in edges:
(12)                        G.add_edge(source, target)
(8)                     return G
(4)                 def build_summary_graph(self):
(8)                     nodes = {
(12)                        "Supernode-0": {"color": "Red"},
(12)                        "Supernode-1": {"color": "Red"},
(12)                        "Supernode-2": {"color": "Blue"},
(12)                        "Supernode-3": {"color": "Blue"},
(12)                        "Supernode-4": {"color": "Yellow"},
(12)                        "Supernode-5": {"color": "Yellow"},
(8)                     }
(8)                     edges = [
(12)                        ("Supernode-0", "Supernode-0"),
(12)                        ("Supernode-0", "Supernode-1"),
(12)                        ("Supernode-0", "Supernode-2"),
(12)                        ("Supernode-0", "Supernode-4"),
(12)                        ("Supernode-1", "Supernode-3"),
(12)                        ("Supernode-4", "Supernode-4"),
(12)                        ("Supernode-4", "Supernode-5"),
(8)                     ]
(8)                     G = nx.Graph()
(8)                     for node in nodes:
(12)                        attributes = nodes[node]
(12)                        G.add_node(node, **attributes)
(8)                     for source, target in edges:
(12)                        G.add_edge(source, target)
(8)                     supernodes = {
(12)                        "Supernode-0": {"A", "B"},
(12)                        "Supernode-1": {"C", "D"},
(12)                        "Supernode-2": {"E", "F"},
(12)                        "Supernode-3": {"G", "H"},
(12)                        "Supernode-4": {"I", "J"},
(12)                        "Supernode-5": {"K", "L"},
(8)                     }
(8)                     nx.set_node_attributes(G, supernodes, "group")
(8)                     return G
(0)             class TestSNAPUndirected(AbstractSNAP):
(4)                 def build_original_graph(self):
(8)                     nodes = {
(12)                        "A": {"color": "Red"},
(12)                        "B": {"color": "Red"},
(12)                        "C": {"color": "Red"},
```

```
(12)                        "D": {"color": "Red"},
(12)                        "E": {"color": "Blue"},
(12)                        "F": {"color": "Blue"},
(12)                        "G": {"color": "Blue"},
(12)                        "H": {"color": "Blue"},
(12)                        "I": {"color": "Yellow"},
(12)                        "J": {"color": "Yellow"},
(12)                        "K": {"color": "Yellow"},
(12)                        "L": {"color": "Yellow"},
(8)                     }
(8)                     edges = [
(12)                        ("A", "B", "Strong"),
(12)                        ("A", "C", "Weak"),
(12)                        ("A", "E", "Strong"),
(12)                        ("A", "I", "Weak"),
(12)                        ("B", "D", "Weak"),
(12)                        ("B", "J", "Weak"),
(12)                        ("B", "F", "Strong"),
(12)                        ("C", "G", "Weak"),
(12)                        ("D", "H", "Weak"),
(12)                        ("I", "J", "Strong"),
(12)                        ("J", "K", "Strong"),
(12)                        ("I", "L", "Strong"),
(8)                     ]
(8)                     G = nx.Graph()
(8)                     for node in nodes:
(12)                        attributes = nodes[node]
(12)                        G.add_node(node, **attributes)
(8)                     for source, target, type in edges:
(12)                        G.add_edge(source, target, type=type)
(8)                     return G
(4)                 def build_summary_graph(self):
(8)                     nodes = {
(12)                        "Supernode-0": {"color": "Red"},
(12)                        "Supernode-1": {"color": "Red"},
(12)                        "Supernode-2": {"color": "Blue"},
(12)                        "Supernode-3": {"color": "Blue"},
(12)                        "Supernode-4": {"color": "Yellow"},
(12)                        "Supernode-5": {"color": "Yellow"},
(8)                     }
(8)                     edges = [
(12)                        ("Supernode-0", "Supernode-0", "Strong"),
(12)                        ("Supernode-0", "Supernode-1", "Weak"),
(12)                        ("Supernode-0", "Supernode-2", "Strong"),
(12)                        ("Supernode-0", "Supernode-4", "Weak"),
(12)                        ("Supernode-1", "Supernode-3", "Weak"),
(12)                        ("Supernode-4", "Supernode-4", "Strong"),
(12)                        ("Supernode-4", "Supernode-5", "Strong"),
(8)                     ]
(8)                     G = nx.Graph()
(8)                     for node in nodes:
(12)                        attributes = nodes[node]
(12)                        G.add_node(node, **attributes)
(8)                     for source, target, type in edges:
(12)                        G.add_edge(source, target, types=[{"type": type}])
(8)                     supernodes = {
(12)                        "Supernode-0": {"A", "B"},
(12)                        "Supernode-1": {"C", "D"},
(12)                        "Supernode-2": {"E", "F"},
(12)                        "Supernode-3": {"G", "H"},
(12)                        "Supernode-4": {"I", "J"},
(12)                        "Supernode-5": {"K", "L"},
(8)                     }
(8)                     nx.set_node_attributes(G, supernodes, "group")
(8)                     return G
(0)             class TestSNAPDirected(AbstractSNAP):
(4)                 def build_original_graph(self):
(8)                     nodes = {
(12)                        "A": {"color": "Red"},
```

```
387: (12)                        "B": {"color": "Red"},
388: (12)                        "C": {"color": "Green"},
389: (12)                        "D": {"color": "Green"},
390: (12)                        "E": {"color": "Blue"},
391: (12)                        "F": {"color": "Blue"},
392: (12)                        "G": {"color": "Yellow"},
393: (12)                        "H": {"color": "Yellow"},
394: (8)                    }
395: (8)                    edges = [
396: (12)                        ("A", "C", "Strong"),
397: (12)                        ("A", "E", "Strong"),
398: (12)                        ("A", "F", "Weak"),
399: (12)                        ("B", "D", "Strong"),
400: (12)                        ("B", "E", "Weak"),
401: (12)                        ("B", "F", "Strong"),
402: (12)                        ("C", "G", "Strong"),
403: (12)                        ("C", "F", "Strong"),
404: (12)                        ("D", "E", "Strong"),
405: (12)                        ("D", "H", "Strong"),
406: (12)                        ("G", "E", "Strong"),
407: (12)                        ("H", "F", "Strong"),
408: (8)                    ]
409: (8)                    G = nx.DiGraph()
410: (8)                    for node in nodes:
411: (12)                        attributes = nodes[node]
412: (12)                        G.add_node(node, **attributes)
413: (8)                    for source, target, type in edges:
414: (12)                        G.add_edge(source, target, type=type)
415: (8)                    return G
416: (4)                def build_summary_graph(self):
417: (8)                    nodes = {
418: (12)                        "Supernode-0": {"color": "Red"},
419: (12)                        "Supernode-1": {"color": "Green"},
420: (12)                        "Supernode-2": {"color": "Blue"},
421: (12)                        "Supernode-3": {"color": "Yellow"},
422: (8)                    }
423: (8)                    edges = [
424: (12)                        ("Supernode-0", "Supernode-1", [{"type": "Strong"}]),
425: (12)                        ("Supernode-0", "Supernode-2", [{"type": "Weak"}, {"type":
"Strong"}]),
426: (12)                        ("Supernode-1", "Supernode-2", [{"type": "Strong"}]),
427: (12)                        ("Supernode-1", "Supernode-3", [{"type": "Strong"}]),
428: (12)                        ("Supernode-3", "Supernode-2", [{"type": "Strong"}]),
429: (8)                    ]
430: (8)                    G = nx.DiGraph()
431: (8)                    for node in nodes:
432: (12)                        attributes = nodes[node]
433: (12)                        G.add_node(node, **attributes)
434: (8)                    for source, target, types in edges:
435: (12)                        G.add_edge(source, target, types=types)
436: (8)                    supernodes = {
437: (12)                        "Supernode-0": {"A", "B"},
438: (12)                        "Supernode-1": {"C", "D"},
439: (12)                        "Supernode-2": {"E", "F"},
440: (12)                        "Supernode-3": {"G", "H"},
441: (12)                        "Supernode-4": {"I", "J"},
442: (12)                        "Supernode-5": {"K", "L"},
443: (8)                    }
444: (8)                    nx.set_node_attributes(G, supernodes, "group")
445: (8)                    return G
446: (0)            class TestSNAPUndirectedMulti(AbstractSNAP):
447: (4)                def build_original_graph(self):
448: (8)                    nodes = {
449: (12)                        "A": {"color": "Red"},
450: (12)                        "B": {"color": "Red"},
451: (12)                        "C": {"color": "Red"},
452: (12)                        "D": {"color": "Blue"},
453: (12)                        "E": {"color": "Blue"},
454: (12)                        "F": {"color": "Blue"},
```

```
455: (12)                        "G": {"color": "Yellow"},
456: (12)                        "H": {"color": "Yellow"},
457: (12)                        "I": {"color": "Yellow"},
458: (8)                    }
459: (8)                    edges = [
460: (12)                        ("A", "D", ["Weak", "Strong"]),
461: (12)                        ("B", "E", ["Weak", "Strong"]),
462: (12)                        ("D", "I", ["Strong"]),
463: (12)                        ("E", "H", ["Strong"]),
464: (12)                        ("F", "G", ["Weak"]),
465: (12)                        ("I", "G", ["Weak", "Strong"]),
466: (12)                        ("I", "H", ["Weak", "Strong"]),
467: (12)                        ("G", "H", ["Weak", "Strong"]),
468: (8)                    ]
469: (8)                    G = nx.MultiGraph()
470: (8)                    for node in nodes:
471: (12)                        attributes = nodes[node]
472: (12)                        G.add_node(node, **attributes)
473: (8)                    for source, target, types in edges:
474: (12)                        for type in types:
475: (16)                            G.add_edge(source, target, type=type)
476: (8)                    return G
477: (4)                def build_summary_graph(self):
478: (8)                    nodes = {
479: (12)                        "Supernode-0": {"color": "Red"},
480: (12)                        "Supernode-1": {"color": "Blue"},
481: (12)                        "Supernode-2": {"color": "Yellow"},
482: (12)                        "Supernode-3": {"color": "Blue"},
483: (12)                        "Supernode-4": {"color": "Yellow"},
484: (12)                        "Supernode-5": {"color": "Red"},
485: (8)                    }
486: (8)                    edges = [
487: (12)                        ("Supernode-1", "Supernode-2", [{"type": "Weak"}]),
488: (12)                        ("Supernode-2", "Supernode-4", [{"type": "Weak"}, {"type":
"Strong"}]),
489: (12)                        ("Supernode-3", "Supernode-4", [{"type": "Strong"}]),
490: (12)                        ("Supernode-3", "Supernode-5", [{"type": "Weak"}, {"type":
"Strong"}]),
491: (12)                        ("Supernode-4", "Supernode-4", [{"type": "Weak"}, {"type":
"Strong"}]),
492: (8)                    ]
493: (8)                    G = nx.MultiGraph()
494: (8)                    for node in nodes:
495: (12)                        attributes = nodes[node]
496: (12)                        G.add_node(node, **attributes)
497: (8)                    for source, target, types in edges:
498: (12)                        for type in types:
499: (16)                            G.add_edge(source, target, type=type)
500: (8)                    supernodes = {
501: (12)                        "Supernode-0": {"A", "B"},
502: (12)                        "Supernode-1": {"C", "D"},
503: (12)                        "Supernode-2": {"E", "F"},
504: (12)                        "Supernode-3": {"G", "H"},
505: (12)                        "Supernode-4": {"I", "J"},
506: (12)                        "Supernode-5": {"K", "L"},
507: (8)                    }
508: (8)                    nx.set_node_attributes(G, supernodes, "group")
509: (8)                    return G
510: (0)            class TestSNAPDirectedMulti(AbstractSNAP):
511: (4)                def build_original_graph(self):
512: (8)                    nodes = {
513: (12)                        "A": {"color": "Red"},
514: (12)                        "B": {"color": "Red"},
515: (12)                        "C": {"color": "Green"},
516: (12)                        "D": {"color": "Green"},
517: (12)                        "E": {"color": "Blue"},
518: (12)                        "F": {"color": "Blue"},
519: (12)                        "G": {"color": "Yellow"},
520: (12)                        "H": {"color": "Yellow"},
```

```
521: (8)                    }
522: (8)                    edges = [
523: (12)                       ("A", "C", ["Weak", "Strong"]),
524: (12)                       ("A", "E", ["Strong"]),
525: (12)                       ("A", "F", ["Weak"]),
526: (12)                       ("B", "D", ["Weak", "Strong"]),
527: (12)                       ("B", "E", ["Weak"]),
528: (12)                       ("B", "F", ["Strong"]),
529: (12)                       ("C", "G", ["Weak", "Strong"]),
530: (12)                       ("C", "F", ["Strong"]),
531: (12)                       ("D", "E", ["Strong"]),
532: (12)                       ("D", "H", ["Weak", "Strong"]),
533: (12)                       ("G", "E", ["Strong"]),
534: (12)                       ("H", "F", ["Strong"]),
535: (8)                    ]
536: (8)                    G = nx.MultiDiGraph()
537: (8)                    for node in nodes:
538: (12)                       attributes = nodes[node]
539: (12)                       G.add_node(node, **attributes)
540: (8)                    for source, target, types in edges:
541: (12)                       for type in types:
542: (16)                           G.add_edge(source, target, type=type)
543: (8)                    return G
544: (4)                def build_summary_graph(self):
545: (8)                    nodes = {
546: (12)                       "Supernode-0": {"color": "Red"},
547: (12)                       "Supernode-1": {"color": "Blue"},
548: (12)                       "Supernode-2": {"color": "Yellow"},
549: (12)                       "Supernode-3": {"color": "Blue"},
550: (8)                    }
551: (8)                    edges = [
552: (12)                       ("Supernode-0", "Supernode-1", ["Weak", "Strong"]),
553: (12)                       ("Supernode-0", "Supernode-2", ["Weak", "Strong"]),
554: (12)                       ("Supernode-1", "Supernode-2", ["Strong"]),
555: (12)                       ("Supernode-1", "Supernode-3", ["Weak", "Strong"]),
556: (12)                       ("Supernode-3", "Supernode-2", ["Strong"]),
557: (8)                    ]
558: (8)                    G = nx.MultiDiGraph()
559: (8)                    for node in nodes:
560: (12)                       attributes = nodes[node]
561: (12)                       G.add_node(node, **attributes)
562: (8)                    for source, target, types in edges:
563: (12)                       for type in types:
564: (16)                           G.add_edge(source, target, type=type)
565: (8)                    supernodes = {
566: (12)                       "Supernode-0": {"A", "B"},
567: (12)                       "Supernode-1": {"C", "D"},
568: (12)                       "Supernode-2": {"E", "F"},
569: (12)                       "Supernode-3": {"G", "H"},
570: (8)                    }
571: (8)                    nx.set_node_attributes(G, supernodes, "group")
572: (8)                    return G
```

File 353 - test_swap.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              cycle = nx.cycle_graph(5, create_using=nx.DiGraph)
4: (0)              tree = nx.random_tree(10, create_using=nx.DiGraph, seed=42)
5: (0)              path = nx.path_graph(5, create_using=nx.DiGraph)
6: (0)              binomial = nx.binomial_tree(3, create_using=nx.DiGraph)
7: (0)              HH = nx.directed_havel_hakimi_graph([1, 2, 1, 2, 2, 2], [3, 1, 0, 1, 2, 3])
8: (0)              balanced_tree = nx.balanced_tree(2, 3, create_using=nx.DiGraph)
9: (0)              @pytest.mark.parametrize("G", [path, binomial, HH, cycle, tree,
balanced_tree])
10: (0)             def test_directed_edge_swap(G):
11: (4)                 in_degree = set(G.in_degree)
```

```
 12: (4)                 out_degree = set(G.out_degree)
 13: (4)                 edges = set(G.edges)
 14: (4)                 nx.directed_edge_swap(G, nswap=1, max_tries=100, seed=1)
 15: (4)                 assert in_degree == set(G.in_degree)
 16: (4)                 assert out_degree == set(G.out_degree)
 17: (4)                 assert edges != set(G.edges)
 18: (4)                 assert 3 == sum(e not in edges for e in G.edges)
 19: (0)             def test_directed_edge_swap_undo_previous_swap():
 20: (4)                 G = nx.DiGraph(nx.path_graph(4).edges)  # only 1 swap possible
 21: (4)                 edges = set(G.edges)
 22: (4)                 nx.directed_edge_swap(G, nswap=2, max_tries=100)
 23: (4)                 assert edges == set(G.edges)
 24: (4)                 nx.directed_edge_swap(G, nswap=1, max_tries=100, seed=1)
 25: (4)                 assert {(0, 2), (1, 3), (2, 1)} == set(G.edges)
 26: (4)                 nx.directed_edge_swap(G, nswap=1, max_tries=100, seed=1)
 27: (4)                 assert edges == set(G.edges)
 28: (0)             def test_edge_cases_directed_edge_swap():
 29: (4)                 e = (
 30: (8)                     "Maximum number of swap attempts \\(11\\) exceeded "
 31: (8)                     "before desired swaps achieved \\(\\d\\)."
 32: (4)                 )
 33: (4)                 graph = nx.DiGraph([(0, 0), (0, 1), (1, 0), (2, 3), (3, 2)])
 34: (4)                 with pytest.raises(nx.NetworkXAlgorithmError, match=e):
 35: (8)                     nx.directed_edge_swap(graph, nswap=1, max_tries=10, seed=1)
 36: (0)             def test_double_edge_swap():
 37: (4)                 graph = nx.barabasi_albert_graph(200, 1)
 38: (4)                 degrees = sorted(d for n, d in graph.degree())
 39: (4)                 G = nx.double_edge_swap(graph, 40)
 40: (4)                 assert degrees == sorted(d for n, d in graph.degree())
 41: (0)             def test_double_edge_swap_seed():
 42: (4)                 graph = nx.barabasi_albert_graph(200, 1)
 43: (4)                 degrees = sorted(d for n, d in graph.degree())
 44: (4)                 G = nx.double_edge_swap(graph, 40, seed=1)
 45: (4)                 assert degrees == sorted(d for n, d in graph.degree())
 46: (0)             def test_connected_double_edge_swap():
 47: (4)                 graph = nx.barabasi_albert_graph(200, 1)
 48: (4)                 degrees = sorted(d for n, d in graph.degree())
 49: (4)                 G = nx.connected_double_edge_swap(graph, 40, seed=1)
 50: (4)                 assert nx.is_connected(graph)
 51: (4)                 assert degrees == sorted(d for n, d in graph.degree())
 52: (0)             def test_connected_double_edge_swap_low_window_threshold():
 53: (4)                 graph = nx.barabasi_albert_graph(200, 1)
 54: (4)                 degrees = sorted(d for n, d in graph.degree())
 55: (4)                 G = nx.connected_double_edge_swap(graph, 40, _window_threshold=0, seed=1)
 56: (4)                 assert nx.is_connected(graph)
 57: (4)                 assert degrees == sorted(d for n, d in graph.degree())
 58: (0)             def test_connected_double_edge_swap_star():
 59: (4)                 graph = nx.star_graph(40)
 60: (4)                 degrees = sorted(d for n, d in graph.degree())
 61: (4)                 G = nx.connected_double_edge_swap(graph, 1, seed=4)
 62: (4)                 assert nx.is_connected(graph)
 63: (4)                 assert degrees == sorted(d for n, d in graph.degree())
 64: (0)             def test_connected_double_edge_swap_star_low_window_threshold():
 65: (4)                 graph = nx.star_graph(40)
 66: (4)                 degrees = sorted(d for n, d in graph.degree())
 67: (4)                 G = nx.connected_double_edge_swap(graph, 1, _window_threshold=0, seed=4)
 68: (4)                 assert nx.is_connected(graph)
 69: (4)                 assert degrees == sorted(d for n, d in graph.degree())
 70: (0)             def test_directed_edge_swap_small():
 71: (4)                 with pytest.raises(nx.NetworkXError):
 72: (8)                     G = nx.directed_edge_swap(nx.path_graph(3, create_using=nx.DiGraph))
 73: (0)             def test_directed_edge_swap_tries():
 74: (4)                 with pytest.raises(nx.NetworkXError):
 75: (8)                     G = nx.directed_edge_swap(
 76: (12)                        nx.path_graph(3, create_using=nx.DiGraph), nswap=1, max_tries=0
 77: (8)                     )
 78: (0)             def test_directed_exception_undirected():
 79: (4)                 graph = nx.Graph([(0, 1), (2, 3)])
 80: (4)                 with pytest.raises(nx.NetworkXNotImplemented):
```

```
81: (8)                     G = nx.directed_edge_swap(graph)
82: (0)             def test_directed_edge_max_tries():
83: (4)                 with pytest.raises(nx.NetworkXAlgorithmError):
84: (8)                     G = nx.directed_edge_swap(
85: (12)                        nx.complete_graph(4, nx.DiGraph()), nswap=1, max_tries=5
86: (8)                     )
87: (0)             def test_double_edge_swap_small():
88: (4)                 with pytest.raises(nx.NetworkXError):
89: (8)                     G = nx.double_edge_swap(nx.path_graph(3))
90: (0)             def test_double_edge_swap_tries():
91: (4)                 with pytest.raises(nx.NetworkXError):
92: (8)                     G = nx.double_edge_swap(nx.path_graph(10), nswap=1, max_tries=0)
93: (0)             def test_double_edge_directed():
94: (4)                 graph = nx.DiGraph([(0, 1), (2, 3)])
95: (4)                 with pytest.raises(nx.NetworkXError, match="not defined for directed
graphs."):
96: (8)                     G = nx.double_edge_swap(graph)
97: (0)             def test_double_edge_max_tries():
98: (4)                 with pytest.raises(nx.NetworkXAlgorithmError):
99: (8)                     G = nx.double_edge_swap(nx.complete_graph(4), nswap=1, max_tries=5)
100: (0)            def test_connected_double_edge_swap_small():
101: (4)                with pytest.raises(nx.NetworkXError):
102: (8)                    G = nx.connected_double_edge_swap(nx.path_graph(3))
103: (0)            def test_connected_double_edge_swap_not_connected():
104: (4)                with pytest.raises(nx.NetworkXError):
105: (8)                    G = nx.path_graph(3)
106: (8)                    nx.add_path(G, [10, 11, 12])
107: (8)                    G = nx.connected_double_edge_swap(G)
108: (0)            def test_degree_seq_c4():
109: (4)                G = nx.cycle_graph(4)
110: (4)                degrees = sorted(d for n, d in G.degree())
111: (4)                G = nx.double_edge_swap(G, 1, 100)
112: (4)                assert degrees == sorted(d for n, d in G.degree())
113: (0)            def test_fewer_than_4_nodes():
114: (4)                G = nx.DiGraph()
115: (4)                G.add_nodes_from([0, 1, 2])
116: (4)                with pytest.raises(nx.NetworkXError, match=".*fewer than four nodes."):
117: (8)                    nx.directed_edge_swap(G)
118: (0)            def test_less_than_3_edges():
119: (4)                G = nx.DiGraph([(0, 1), (1, 2)])
120: (4)                G.add_nodes_from([3, 4])
121: (4)                with pytest.raises(nx.NetworkXError, match=".*fewer than 3 edges"):
122: (8)                    nx.directed_edge_swap(G)
123: (4)                G = nx.Graph()
124: (4)                G.add_nodes_from([0, 1, 2, 3])
125: (4)                with pytest.raises(nx.NetworkXError, match=".*fewer than 2 edges"):
126: (8)                    nx.double_edge_swap(G)

----------------------------------------

File 354 - test_threshold.py:

1: (0)              """
2: (0)              Threshold Graphs
3: (0)              ================
4: (0)              """
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              import networkx.algorithms.threshold as nxt
8: (0)              from networkx.algorithms.isomorphism.isomorph import graph_could_be_isomorphic
9: (0)              cnlti = nx.convert_node_labels_to_integers
10: (0)             class TestGeneratorThreshold:
11: (4)                 def test_threshold_sequence_graph_test(self):
12: (8)                     G = nx.star_graph(10)
13: (8)                     assert nxt.is_threshold_graph(G)
14: (8)                     assert nxt.is_threshold_sequence([d for n, d in G.degree()])
15: (8)                     G = nx.complete_graph(10)
16: (8)                     assert nxt.is_threshold_graph(G)
17: (8)                     assert nxt.is_threshold_sequence([d for n, d in G.degree()])
```

```
 18: (8)                    deg = [3, 2, 2, 1, 1, 1]
 19: (8)                    assert not nxt.is_threshold_sequence(deg)
 20: (8)                    deg = [3, 2, 2, 1]
 21: (8)                    assert nxt.is_threshold_sequence(deg)
 22: (8)                    G = nx.generators.havel_hakimi_graph(deg)
 23: (8)                    assert nxt.is_threshold_graph(G)
 24: (4)                def test_creation_sequences(self):
 25: (8)                    deg = [3, 2, 2, 1]
 26: (8)                    G = nx.generators.havel_hakimi_graph(deg)
 27: (8)                    with pytest.raises(ValueError):
 28: (12)                       nxt.creation_sequence(deg, with_labels=True, compact=True)
 29: (8)                    cs0 = nxt.creation_sequence(deg)
 30: (8)                    H0 = nxt.threshold_graph(cs0)
 31: (8)                    assert "".join(cs0) == "ddid"
 32: (8)                    cs1 = nxt.creation_sequence(deg, with_labels=True)
 33: (8)                    H1 = nxt.threshold_graph(cs1)
 34: (8)                    assert cs1 == [(1, "d"), (2, "d"), (3, "i"), (0, "d")]
 35: (8)                    cs2 = nxt.creation_sequence(deg, compact=True)
 36: (8)                    H2 = nxt.threshold_graph(cs2)
 37: (8)                    assert cs2 == [2, 1, 1]
 38: (8)                    assert "".join(nxt.uncompact(cs2)) == "ddid"
 39: (8)                    assert graph_could_be_isomorphic(H0, G)
 40: (8)                    assert graph_could_be_isomorphic(H0, H1)
 41: (8)                    assert graph_could_be_isomorphic(H0, H2)
 42: (4)                def test_make_compact(self):
 43: (8)                    assert nxt.make_compact(["d", "d", "d", "i", "d", "d"]) == [3, 1, 2]
 44: (8)                    assert nxt.make_compact([3, 1, 2]) == [3, 1, 2]
 45: (8)                    assert pytest.raises(TypeError, nxt.make_compact, [3.0, 1.0, 2.0])
 46: (4)                def test_uncompact(self):
 47: (8)                    assert nxt.uncompact([3, 1, 2]) == ["d", "d", "d", "i", "d", "d"]
 48: (8)                    assert nxt.uncompact(["d", "d", "i", "d"]) == ["d", "d", "i", "d"]
 49: (8)                    assert nxt.uncompact(
 50: (12)                       nxt.uncompact([(1, "d"), (2, "d"), (3, "i"), (0, "d")])
 51: (8)                    ) == nxt.uncompact([(1, "d"), (2, "d"), (3, "i"), (0, "d")])
 52: (8)                    assert pytest.raises(TypeError, nxt.uncompact, [3.0, 1.0, 2.0])
 53: (4)                def test_creation_sequence_to_weights(self):
 54: (8)                    assert nxt.creation_sequence_to_weights([3, 1, 2]) == [
 55: (12)                       0.5,
 56: (12)                       0.5,
 57: (12)                       0.5,
 58: (12)                       0.25,
 59: (12)                       0.75,
 60: (12)                       0.75,
 61: (8)                    ]
 62: (8)                    assert pytest.raises(
 63: (12)                       TypeError, nxt.creation_sequence_to_weights, [3.0, 1.0, 2.0]
 64: (8)                    )
 65: (4)                def test_weights_to_creation_sequence(self):
 66: (8)                    deg = [3, 2, 2, 1]
 67: (8)                    with pytest.raises(ValueError):
 68: (12)                       nxt.weights_to_creation_sequence(deg, with_labels=True,
compact=True)
 69: (8)                    assert nxt.weights_to_creation_sequence(deg, with_labels=True) == [
 70: (12)                       (3, "d"),
 71: (12)                       (1, "d"),
 72: (12)                       (2, "d"),
 73: (12)                       (0, "d"),
 74: (8)                    ]
 75: (8)                    assert nxt.weights_to_creation_sequence(deg, compact=True) == [4]
 76: (4)                def test_find_alternating_4_cycle(self):
 77: (8)                    G = nx.Graph()
 78: (8)                    G.add_edge(1, 2)
 79: (8)                    assert not nxt.find_alternating_4_cycle(G)
 80: (4)                def test_shortest_path(self):
 81: (8)                    deg = [3, 2, 2, 1]
 82: (8)                    G = nx.generators.havel_hakimi_graph(deg)
 83: (8)                    cs1 = nxt.creation_sequence(deg, with_labels=True)
 84: (8)                    for n, m in [(3, 0), (0, 3), (0, 2), (0, 1), (1, 3), (3, 1), (1, 2),
(2, 3)]:
```

```
85: (12)                        assert nxt.shortest_path(cs1, n, m) == nx.shortest_path(G, n, m)
86: (8)                     spl = nxt.shortest_path_length(cs1, 3)
87: (8)                     spl2 = nxt.shortest_path_length([t for v, t in cs1], 2)
88: (8)                     assert spl == spl2
89: (8)                     spld = {}
90: (8)                     for j, pl in enumerate(spl):
91: (12)                        n = cs1[j][0]
92: (12)                        spld[n] = pl
93: (8)                     assert spld == nx.single_source_shortest_path_length(G, 3)
94: (8)                     assert nxt.shortest_path(["d", "d", "d", "i", "d", "d"], 1, 2) == [1,
2]
95: (8)                     assert nxt.shortest_path([3, 1, 2], 1, 2) == [1, 2]
96: (8)                     assert pytest.raises(TypeError, nxt.shortest_path, [3.0, 1.0, 2.0], 1,
2)
97: (8)                     assert pytest.raises(ValueError, nxt.shortest_path, [3, 1, 2], "a", 2)
98: (8)                     assert pytest.raises(ValueError, nxt.shortest_path, [3, 1, 2], 1, "b")
99: (8)                     assert nxt.shortest_path([3, 1, 2], 1, 1) == [1]
100: (4)                def test_shortest_path_length(self):
101: (8)                    assert nxt.shortest_path_length([3, 1, 2], 1) == [1, 0, 1, 2, 1, 1]
102: (8)                    assert nxt.shortest_path_length(["d", "d", "d", "i", "d", "d"], 1) ==
[
103: (12)                       1,
104: (12)                       0,
105: (12)                       1,
106: (12)                       2,
107: (12)                       1,
108: (12)                       1,
109: (8)                    ]
110: (8)                    assert nxt.shortest_path_length(("d", "d", "d", "i", "d", "d"), 1) ==
[
111: (12)                       1,
112: (12)                       0,
113: (12)                       1,
114: (12)                       2,
115: (12)                       1,
116: (12)                       1,
117: (8)                    ]
118: (8)                    assert pytest.raises(TypeError, nxt.shortest_path, [3.0, 1.0, 2.0], 1)
119: (4)                def test_random_threshold_sequence(self):
120: (8)                    assert len(nxt.random_threshold_sequence(10, 0.5)) == 10
121: (8)                    assert nxt.random_threshold_sequence(10, 0.5, seed=42) == [
122: (12)                       "d",
123: (12)                       "i",
124: (12)                       "d",
125: (12)                       "d",
126: (12)                       "d",
127: (12)                       "i",
128: (12)                       "i",
129: (12)                       "i",
130: (12)                       "d",
131: (12)                       "d",
132: (8)                    ]
133: (8)                    assert pytest.raises(ValueError, nxt.random_threshold_sequence, 10,
1.5)
134: (4)                def test_right_d_threshold_sequence(self):
135: (8)                    assert nxt.right_d_threshold_sequence(3, 2) == ["d", "i", "d"]
136: (8)                    assert pytest.raises(ValueError, nxt.right_d_threshold_sequence, 2, 3)
137: (4)                def test_left_d_threshold_sequence(self):
138: (8)                    assert nxt.left_d_threshold_sequence(3, 2) == ["d", "i", "d"]
139: (8)                    assert pytest.raises(ValueError, nxt.left_d_threshold_sequence, 2, 3)
140: (4)                def test_weights_thresholds(self):
141: (8)                    wseq = [3, 4, 3, 3, 5, 6, 5, 4, 5, 6]
142: (8)                    cs = nxt.weights_to_creation_sequence(wseq, threshold=10)
143: (8)                    wseq = nxt.creation_sequence_to_weights(cs)
144: (8)                    cs2 = nxt.weights_to_creation_sequence(wseq)
145: (8)                    assert cs == cs2
146: (8)                    wseq = nxt.creation_sequence_to_weights(nxt.uncompact([3, 1, 2, 3, 3,
2, 3]))
147: (8)                    assert wseq == [
```

```
148: (12)                        s * 0.125 for s in [4, 4, 4, 3, 5, 5, 2, 2, 2, 6, 6, 6, 1, 1, 7,
7, 7]
149: (8)                    ]
150: (8)                    wseq = nxt.creation_sequence_to_weights([3, 1, 2, 3, 3, 2, 3])
151: (8)                    assert wseq == [
152: (12)                        s * 0.125 for s in [4, 4, 4, 3, 5, 5, 2, 2, 2, 6, 6, 6, 1, 1, 7,
7, 7]
153: (8)                    ]
154: (8)                    wseq =
nxt.creation_sequence_to_weights(list(enumerate("ddidiiidididi")))
155: (8)                    assert wseq == [s * 0.1 for s in [5, 5, 4, 6, 3, 3, 3, 7, 2, 8, 1, 9,
0]]
156: (8)                    wseq = nxt.creation_sequence_to_weights("ddidiiidididi")
157: (8)                    assert wseq == [s * 0.1 for s in [5, 5, 4, 6, 3, 3, 3, 7, 2, 8, 1, 9,
0]]
158: (8)                    wseq = nxt.creation_sequence_to_weights("ddidiiidididid")
159: (8)                    ws = [s / 12 for s in [6, 6, 5, 7, 4, 4, 4, 8, 3, 9, 2, 10, 1, 11]]
160: (8)                    assert sum(abs(c - d) for c, d in zip(wseq, ws)) < 1e-14
161: (4)                def test_finding_routines(self):
162: (8)                    G = nx.Graph({1: [2], 2: [3], 3: [4], 4: [5], 5: [6]})
163: (8)                    G.add_edge(2, 4)
164: (8)                    G.add_edge(2, 5)
165: (8)                    G.add_edge(2, 7)
166: (8)                    G.add_edge(3, 6)
167: (8)                    G.add_edge(4, 6)
168: (8)                    assert nxt.find_alternating_4_cycle(G) == [1, 2, 3, 6]
169: (8)                    TG = nxt.find_threshold_graph(G)
170: (8)                    assert nxt.is_threshold_graph(TG)
171: (8)                    assert sorted(TG.nodes()) == [1, 2, 3, 4, 5, 7]
172: (8)                    cs = nxt.creation_sequence(dict(TG.degree()), with_labels=True)
173: (8)                    assert nxt.find_creation_sequence(G) == cs
174: (4)                def test_fast_versions_properties_threshold_graphs(self):
175: (8)                    cs = "ddiiddid"
176: (8)                    G = nxt.threshold_graph(cs)
177: (8)                    assert nxt.density("ddiiddid") == nx.density(G)
178: (8)                    assert sorted(nxt.degree_sequence(cs)) == sorted(d for n, d in
G.degree())
179: (8)                    ts = nxt.triangle_sequence(cs)
180: (8)                    assert ts == list(nx.triangles(G).values())
181: (8)                    assert sum(ts) // 3 == nxt.triangles(cs)
182: (8)                    c1 = nxt.cluster_sequence(cs)
183: (8)                    c2 = list(nx.clustering(G).values())
184: (8)                    assert sum(abs(c - d) for c, d in zip(c1, c2)) == pytest.approx(0,
abs=1e-7)
185: (8)                    b1 = nx.betweenness_centrality(G).values()
186: (8)                    b2 = nxt.betweenness_sequence(cs)
187: (8)                    assert sum(abs(c - d) for c, d in zip(b1, b2)) < 1e-7
188: (8)                    assert nxt.eigenvalues(cs) == [0, 1, 3, 3, 5, 7, 7, 8]
189: (8)                    assert abs(nxt.degree_correlation(cs) + 0.593038821954) < 1e-12
190: (8)                    assert nxt.degree_correlation("diiiddi") == -0.8
191: (8)                    assert nxt.degree_correlation("did") == -1.0
192: (8)                    assert nxt.degree_correlation("ddd") == 1.0
193: (8)                    assert nxt.eigenvalues("dddiii") == [0, 0, 0, 0, 3, 3]
194: (8)                    assert nxt.eigenvalues("dddiiid") == [0, 1, 1, 1, 4, 4, 7]
195: (4)                def test_tg_creation_routines(self):
196: (8)                    s = nxt.left_d_threshold_sequence(5, 7)
197: (8)                    s = nxt.right_d_threshold_sequence(5, 7)
198: (8)                    s1 = nxt.swap_d(s, 1.0, 1.0)
199: (8)                    s1 = nxt.swap_d(s, 1.0, 1.0, seed=1)
200: (4)                def test_eigenvectors(self):
201: (8)                    np = pytest.importorskip("numpy")
202: (8)                    eigenval = np.linalg.eigvals
203: (8)                    pytest.importorskip("scipy")
204: (8)                    cs = "ddiiddid"
205: (8)                    G = nxt.threshold_graph(cs)
206: (8)                    (tgeval, tgevec) = nxt.eigenvectors(cs)
207: (8)                    np.testing.assert_allclose([np.dot(lv, lv) for lv in tgevec], 1.0,
rtol=1e-9)
208: (8)                    lapl = nx.laplacian_matrix(G)
```

```
209: (4)                def test_create_using(self):
210: (8)                    cs = "ddiiddid"
211: (8)                    G = nxt.threshold_graph(cs)
212: (8)                    assert pytest.raises(
213: (12)                       nx.exception.NetworkXError,
214: (12)                       nxt.threshold_graph,
215: (12)                       cs,
216: (12)                       create_using=nx.DiGraph(),
217: (8)                    )
218: (8)                    MG = nxt.threshold_graph(cs, create_using=nx.MultiGraph())
219: (8)                    assert sorted(MG.edges()) == sorted(G.edges())
```

File 355 - test_time_dependent.py:

```
1: (0)              """Unit testing for time dependent algorithms."""
2: (0)              from datetime import datetime, timedelta
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              _delta = timedelta(days=5 * 365)
6: (0)              class TestCdIndex:
7: (4)                  """Unit testing for the cd index function."""
8: (4)                  def test_common_graph(self):
9: (8)                      G = nx.DiGraph()
10: (8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
11: (8)                     G.add_edge(4, 2)
12: (8)                     G.add_edge(4, 0)
13: (8)                     G.add_edge(4, 1)
14: (8)                     G.add_edge(4, 3)
15: (8)                     G.add_edge(5, 2)
16: (8)                     G.add_edge(6, 2)
17: (8)                     G.add_edge(6, 4)
18: (8)                     G.add_edge(7, 4)
19: (8)                     G.add_edge(8, 4)
20: (8)                     G.add_edge(9, 4)
21: (8)                     G.add_edge(9, 1)
22: (8)                     G.add_edge(9, 3)
23: (8)                     G.add_edge(10, 4)
24: (8)                     node_attrs = {
25: (12)                        0: {"time": datetime(1992, 1, 1)},
26: (12)                        1: {"time": datetime(1992, 1, 1)},
27: (12)                        2: {"time": datetime(1993, 1, 1)},
28: (12)                        3: {"time": datetime(1993, 1, 1)},
29: (12)                        4: {"time": datetime(1995, 1, 1)},
30: (12)                        5: {"time": datetime(1997, 1, 1)},
31: (12)                        6: {"time": datetime(1998, 1, 1)},
32: (12)                        7: {"time": datetime(1999, 1, 1)},
33: (12)                        8: {"time": datetime(1999, 1, 1)},
34: (12)                        9: {"time": datetime(1998, 1, 1)},
35: (12)                        10: {"time": datetime(1997, 4, 1)},
36: (8)                     }
37: (8)                     nx.set_node_attributes(G, node_attrs)
38: (8)                     assert nx.cd_index(G, 4, time_delta=_delta) == 0.17
39: (4)                 def test_common_graph_with_given_attributes(self):
40: (8)                     G = nx.DiGraph()
41: (8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
42: (8)                     G.add_edge(4, 2)
43: (8)                     G.add_edge(4, 0)
44: (8)                     G.add_edge(4, 1)
45: (8)                     G.add_edge(4, 3)
46: (8)                     G.add_edge(5, 2)
47: (8)                     G.add_edge(6, 2)
48: (8)                     G.add_edge(6, 4)
49: (8)                     G.add_edge(7, 4)
50: (8)                     G.add_edge(8, 4)
51: (8)                     G.add_edge(9, 4)
52: (8)                     G.add_edge(9, 1)
53: (8)                     G.add_edge(9, 3)
```

```
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"date": datetime(1992, 1, 1)},
(12)                        1: {"date": datetime(1992, 1, 1)},
(12)                        2: {"date": datetime(1993, 1, 1)},
(12)                        3: {"date": datetime(1993, 1, 1)},
(12)                        4: {"date": datetime(1995, 1, 1)},
(12)                        5: {"date": datetime(1997, 1, 1)},
(12)                        6: {"date": datetime(1998, 1, 1)},
(12)                        7: {"date": datetime(1999, 1, 1)},
(12)                        8: {"date": datetime(1999, 1, 1)},
(12)                        9: {"date": datetime(1998, 1, 1)},
(12)                        10: {"date": datetime(1997, 4, 1)},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
(8)                     assert nx.cd_index(G, 4, time_delta=_delta, time="date") == 0.17
(4)                 def test_common_graph_with_int_attributes(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)                     G.add_edge(4, 2)
(8)                     G.add_edge(4, 0)
(8)                     G.add_edge(4, 1)
(8)                     G.add_edge(4, 3)
(8)                     G.add_edge(5, 2)
(8)                     G.add_edge(6, 2)
(8)                     G.add_edge(6, 4)
(8)                     G.add_edge(7, 4)
(8)                     G.add_edge(8, 4)
(8)                     G.add_edge(9, 4)
(8)                     G.add_edge(9, 1)
(8)                     G.add_edge(9, 3)
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"time": 20},
(12)                        1: {"time": 20},
(12)                        2: {"time": 30},
(12)                        3: {"time": 30},
(12)                        4: {"time": 50},
(12)                        5: {"time": 70},
(12)                        6: {"time": 80},
(12)                        7: {"time": 90},
(12)                        8: {"time": 90},
(12)                        9: {"time": 80},
(12)                        10: {"time": 74},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
(8)                     assert nx.cd_index(G, 4, time_delta=50) == 0.17
(4)                 def test_common_graph_with_float_attributes(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)                     G.add_edge(4, 2)
(8)                     G.add_edge(4, 0)
(8)                     G.add_edge(4, 1)
(8)                     G.add_edge(4, 3)
(8)                     G.add_edge(5, 2)
(8)                     G.add_edge(6, 2)
(8)                     G.add_edge(6, 4)
(8)                     G.add_edge(7, 4)
(8)                     G.add_edge(8, 4)
(8)                     G.add_edge(9, 4)
(8)                     G.add_edge(9, 1)
(8)                     G.add_edge(9, 3)
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"time": 20.2},
(12)                        1: {"time": 20.2},
(12)                        2: {"time": 30.7},
(12)                        3: {"time": 30.7},
(12)                        4: {"time": 50.9},
```

```
(12)                        5: {"time": 70.1},
(12)                        6: {"time": 80.6},
(12)                        7: {"time": 90.7},
(12)                        8: {"time": 90.7},
(12)                        9: {"time": 80.6},
(12)                        10: {"time": 74.2},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
(8)                     assert nx.cd_index(G, 4, time_delta=50) == 0.17
(4)                 def test_common_graph_with_weights(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)                     G.add_edge(4, 2)
(8)                     G.add_edge(4, 0)
(8)                     G.add_edge(4, 1)
(8)                     G.add_edge(4, 3)
(8)                     G.add_edge(5, 2)
(8)                     G.add_edge(6, 2)
(8)                     G.add_edge(6, 4)
(8)                     G.add_edge(7, 4)
(8)                     G.add_edge(8, 4)
(8)                     G.add_edge(9, 4)
(8)                     G.add_edge(9, 1)
(8)                     G.add_edge(9, 3)
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"time": datetime(1992, 1, 1)},
(12)                        1: {"time": datetime(1992, 1, 1)},
(12)                        2: {"time": datetime(1993, 1, 1)},
(12)                        3: {"time": datetime(1993, 1, 1)},
(12)                        4: {"time": datetime(1995, 1, 1)},
(12)                        5: {"time": datetime(1997, 1, 1)},
(12)                        6: {"time": datetime(1998, 1, 1), "weight": 5},
(12)                        7: {"time": datetime(1999, 1, 1), "weight": 2},
(12)                        8: {"time": datetime(1999, 1, 1), "weight": 6},
(12)                        9: {"time": datetime(1998, 1, 1), "weight": 3},
(12)                        10: {"time": datetime(1997, 4, 1), "weight": 10},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
(8)                     assert nx.cd_index(G, 4, time_delta=_delta, weight="weight") == 0.04
(4)                 def test_node_with_no_predecessors(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)                     G.add_edge(4, 2)
(8)                     G.add_edge(4, 0)
(8)                     G.add_edge(4, 3)
(8)                     G.add_edge(5, 2)
(8)                     G.add_edge(6, 2)
(8)                     G.add_edge(6, 4)
(8)                     G.add_edge(7, 4)
(8)                     G.add_edge(8, 4)
(8)                     G.add_edge(9, 4)
(8)                     G.add_edge(9, 1)
(8)                     G.add_edge(9, 3)
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"time": datetime(1992, 1, 1)},
(12)                        1: {"time": datetime(1992, 1, 1)},
(12)                        2: {"time": datetime(1993, 1, 1)},
(12)                        3: {"time": datetime(1993, 1, 1)},
(12)                        4: {"time": datetime(1995, 1, 1)},
(12)                        5: {"time": datetime(2005, 1, 1)},
(12)                        6: {"time": datetime(2010, 1, 1)},
(12)                        7: {"time": datetime(2001, 1, 1)},
(12)                        8: {"time": datetime(2020, 1, 1)},
(12)                        9: {"time": datetime(2017, 1, 1)},
(12)                        10: {"time": datetime(2004, 4, 1)},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
```

```
(8)         assert nx.cd_index(G, 4, time_delta=_delta) == 0.0
(4)     def test_node_with_no_successors(self):
(8)         G = nx.DiGraph()
(8)         G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)         G.add_edge(8, 2)
(8)         G.add_edge(6, 0)
(8)         G.add_edge(6, 3)
(8)         G.add_edge(5, 2)
(8)         G.add_edge(6, 2)
(8)         G.add_edge(6, 4)
(8)         G.add_edge(7, 4)
(8)         G.add_edge(8, 4)
(8)         G.add_edge(9, 4)
(8)         G.add_edge(9, 1)
(8)         G.add_edge(9, 3)
(8)         G.add_edge(10, 4)
(8)         node_attrs = {
(12)            0: {"time": datetime(1992, 1, 1)},
(12)            1: {"time": datetime(1992, 1, 1)},
(12)            2: {"time": datetime(1993, 1, 1)},
(12)            3: {"time": datetime(1993, 1, 1)},
(12)            4: {"time": datetime(1995, 1, 1)},
(12)            5: {"time": datetime(1997, 1, 1)},
(12)            6: {"time": datetime(1998, 1, 1)},
(12)            7: {"time": datetime(1999, 1, 1)},
(12)            8: {"time": datetime(1999, 1, 1)},
(12)            9: {"time": datetime(1998, 1, 1)},
(12)            10: {"time": datetime(1997, 4, 1)},
(8)         }
(8)         nx.set_node_attributes(G, node_attrs)
(8)         assert nx.cd_index(G, 4, time_delta=_delta) == 1.0
(4)     def test_n_equals_zero(self):
(8)         G = nx.DiGraph()
(8)         G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)         G.add_edge(4, 2)
(8)         G.add_edge(4, 0)
(8)         G.add_edge(4, 3)
(8)         G.add_edge(6, 4)
(8)         G.add_edge(7, 4)
(8)         G.add_edge(8, 4)
(8)         G.add_edge(9, 4)
(8)         G.add_edge(9, 1)
(8)         G.add_edge(10, 4)
(8)         node_attrs = {
(12)            0: {"time": datetime(1992, 1, 1)},
(12)            1: {"time": datetime(1992, 1, 1)},
(12)            2: {"time": datetime(1993, 1, 1)},
(12)            3: {"time": datetime(1993, 1, 1)},
(12)            4: {"time": datetime(1995, 1, 1)},
(12)            5: {"time": datetime(2005, 1, 1)},
(12)            6: {"time": datetime(2010, 1, 1)},
(12)            7: {"time": datetime(2001, 1, 1)},
(12)            8: {"time": datetime(2020, 1, 1)},
(12)            9: {"time": datetime(2017, 1, 1)},
(12)            10: {"time": datetime(2004, 4, 1)},
(8)         }
(8)         nx.set_node_attributes(G, node_attrs)
(8)         with pytest.raises(
(12)            nx.NetworkXError, match="The cd index cannot be defined."
(8)         ) as ve:
(12)            nx.cd_index(G, 4, time_delta=_delta)
(4)     def test_time_timedelta_compatibility(self):
(8)         G = nx.DiGraph()
(8)         G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)         G.add_edge(4, 2)
(8)         G.add_edge(4, 0)
(8)         G.add_edge(4, 3)
(8)         G.add_edge(6, 4)
(8)         G.add_edge(7, 4)
```

```
(8)                     G.add_edge(8, 4)
(8)                     G.add_edge(9, 4)
(8)                     G.add_edge(9, 1)
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"time": 20.2},
(12)                        1: {"time": 20.2},
(12)                        2: {"time": 30.7},
(12)                        3: {"time": 30.7},
(12)                        4: {"time": 50.9},
(12)                        5: {"time": 70.1},
(12)                        6: {"time": 80.6},
(12)                        7: {"time": 90.7},
(12)                        8: {"time": 90.7},
(12)                        9: {"time": 80.6},
(12)                        10: {"time": 74.2},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
(8)                     with pytest.raises(
(12)                        nx.NetworkXError,
(12)                        match="Addition and comparison are not supported between",
(8)                     ) as ve:
(12)                        nx.cd_index(G, 4, time_delta=_delta)
(4)                 def test_node_with_no_time(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
(8)                     G.add_edge(8, 2)
(8)                     G.add_edge(6, 0)
(8)                     G.add_edge(6, 3)
(8)                     G.add_edge(5, 2)
(8)                     G.add_edge(6, 2)
(8)                     G.add_edge(6, 4)
(8)                     G.add_edge(7, 4)
(8)                     G.add_edge(8, 4)
(8)                     G.add_edge(9, 4)
(8)                     G.add_edge(9, 1)
(8)                     G.add_edge(9, 3)
(8)                     G.add_edge(10, 4)
(8)                     node_attrs = {
(12)                        0: {"time": datetime(1992, 1, 1)},
(12)                        1: {"time": datetime(1992, 1, 1)},
(12)                        2: {"time": datetime(1993, 1, 1)},
(12)                        3: {"time": datetime(1993, 1, 1)},
(12)                        4: {"time": datetime(1995, 1, 1)},
(12)                        6: {"time": datetime(1998, 1, 1)},
(12)                        7: {"time": datetime(1999, 1, 1)},
(12)                        8: {"time": datetime(1999, 1, 1)},
(12)                        9: {"time": datetime(1998, 1, 1)},
(12)                        10: {"time": datetime(1997, 4, 1)},
(8)                     }
(8)                     nx.set_node_attributes(G, node_attrs)
(8)                     with pytest.raises(
(12)                        nx.NetworkXError, match="Not all nodes have a 'time' attribute."
(8)                     ) as ve:
(12)                        nx.cd_index(G, 4, time_delta=_delta)
(4)                 def test_maximally_consolidating(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
(8)                     G.add_edge(5, 1)
(8)                     G.add_edge(5, 2)
(8)                     G.add_edge(5, 3)
(8)                     G.add_edge(5, 4)
(8)                     G.add_edge(6, 1)
(8)                     G.add_edge(6, 5)
(8)                     G.add_edge(7, 1)
(8)                     G.add_edge(7, 5)
(8)                     G.add_edge(8, 2)
(8)                     G.add_edge(8, 5)
(8)                     G.add_edge(9, 5)
```

```
330: (8)                    G.add_edge(9, 3)
331: (8)                    G.add_edge(10, 5)
332: (8)                    G.add_edge(10, 3)
333: (8)                    G.add_edge(10, 4)
334: (8)                    G.add_edge(11, 5)
335: (8)                    G.add_edge(11, 4)
336: (8)                    node_attrs = {
337: (12)                       0: {"time": datetime(1992, 1, 1)},
338: (12)                       1: {"time": datetime(1992, 1, 1)},
339: (12)                       2: {"time": datetime(1993, 1, 1)},
340: (12)                       3: {"time": datetime(1993, 1, 1)},
341: (12)                       4: {"time": datetime(1995, 1, 1)},
342: (12)                       5: {"time": datetime(1997, 1, 1)},
343: (12)                       6: {"time": datetime(1998, 1, 1)},
344: (12)                       7: {"time": datetime(1999, 1, 1)},
345: (12)                       8: {"time": datetime(1999, 1, 1)},
346: (12)                       9: {"time": datetime(1998, 1, 1)},
347: (12)                       10: {"time": datetime(1997, 4, 1)},
348: (12)                       11: {"time": datetime(1998, 5, 1)},
349: (8)                    }
350: (8)                    nx.set_node_attributes(G, node_attrs)
351: (8)                    assert nx.cd_index(G, 5, time_delta=_delta) == -1
352: (4)                def test_maximally_destabilizing(self):
353: (8)                    G = nx.DiGraph()
354: (8)                    G.add_nodes_from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
355: (8)                    G.add_edge(5, 1)
356: (8)                    G.add_edge(5, 2)
357: (8)                    G.add_edge(5, 3)
358: (8)                    G.add_edge(5, 4)
359: (8)                    G.add_edge(6, 5)
360: (8)                    G.add_edge(7, 5)
361: (8)                    G.add_edge(8, 5)
362: (8)                    G.add_edge(9, 5)
363: (8)                    G.add_edge(10, 5)
364: (8)                    G.add_edge(11, 5)
365: (8)                    node_attrs = {
366: (12)                       0: {"time": datetime(1992, 1, 1)},
367: (12)                       1: {"time": datetime(1992, 1, 1)},
368: (12)                       2: {"time": datetime(1993, 1, 1)},
369: (12)                       3: {"time": datetime(1993, 1, 1)},
370: (12)                       4: {"time": datetime(1995, 1, 1)},
371: (12)                       5: {"time": datetime(1997, 1, 1)},
372: (12)                       6: {"time": datetime(1998, 1, 1)},
373: (12)                       7: {"time": datetime(1999, 1, 1)},
374: (12)                       8: {"time": datetime(1999, 1, 1)},
375: (12)                       9: {"time": datetime(1998, 1, 1)},
376: (12)                       10: {"time": datetime(1997, 4, 1)},
377: (12)                       11: {"time": datetime(1998, 5, 1)},
378: (8)                    }
379: (8)                    nx.set_node_attributes(G, node_attrs)
380: (8)                    assert nx.cd_index(G, 5, time_delta=_delta) == 1

----------------------------------------

File 356 - test_triads.py:

1: (0)              """Tests for the :mod:`networkx.algorithms.triads` module."""
2: (0)              import itertools
3: (0)              from collections import defaultdict
4: (0)              from random import sample
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              def test_all_triplets_deprecated():
8: (4)                  G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
9: (4)                  with pytest.deprecated_call():
10: (8)                     nx.all_triplets(G)
11: (0)             def test_random_triad_deprecated():
12: (4)                 G = nx.path_graph(3, create_using=nx.DiGraph)
13: (4)                 with pytest.deprecated_call():
```

```
 14: (8)                    nx.random_triad(G)
 15: (0)             def test_triadic_census():
 16: (4)                 """Tests the triadic_census function."""
 17: (4)                 G = nx.DiGraph()
 18: (4)                 G.add_edges_from(["01", "02", "03", "04", "05", "12", "16", "51", "56",
"65"])
 19: (4)                 expected = {
 20: (8)                     "030T": 2,
 21: (8)                     "120C": 1,
 22: (8)                     "210": 0,
 23: (8)                     "120U": 0,
 24: (8)                     "012": 9,
 25: (8)                     "102": 3,
 26: (8)                     "021U": 0,
 27: (8)                     "111U": 0,
 28: (8)                     "003": 8,
 29: (8)                     "030C": 0,
 30: (8)                     "021D": 9,
 31: (8)                     "201": 0,
 32: (8)                     "111D": 1,
 33: (8)                     "300": 0,
 34: (8)                     "120D": 0,
 35: (8)                     "021C": 2,
 36: (4)                 }
 37: (4)                 actual = nx.triadic_census(G)
 38: (4)                 assert expected == actual
 39: (0)             def test_is_triad():
 40: (4)                 """Tests the is_triad function"""
 41: (4)                 G = nx.karate_club_graph()
 42: (4)                 G = G.to_directed()
 43: (4)                 for i in range(100):
 44: (8)                     nodes = sample(sorted(G.nodes()), 3)
 45: (8)                     G2 = G.subgraph(nodes)
 46: (8)                     assert nx.is_triad(G2)
 47: (0)             def test_all_triplets():
 48: (4)                 """Tests the all_triplets function."""
 49: (4)                 G = nx.DiGraph()
 50: (4)                 G.add_edges_from(["01", "02", "03", "04", "05", "12", "16", "51", "56",
"65"])
 51: (4)                 expected = [
 52: (8)                     f"{i},{j},{k}"
 53: (8)                     for i in range(7)
 54: (8)                     for j in range(i + 1, 7)
 55: (8)                     for k in range(j + 1, 7)
 56: (4)                 ]
 57: (4)                 expected = [set(x.split(",")) for x in expected]
 58: (4)                 actual = [set(x) for x in nx.all_triplets(G)]
 59: (4)                 assert all(any(s1 == s2 for s1 in expected) for s2 in actual)
 60: (0)             def test_all_triads():
 61: (4)                 """Tests the all_triplets function."""
 62: (4)                 G = nx.DiGraph()
 63: (4)                 G.add_edges_from(["01", "02", "03", "04", "05", "12", "16", "51", "56",
"65"])
 64: (4)                 expected = [
 65: (8)                     f"{i},{j},{k}"
 66: (8)                     for i in range(7)
 67: (8)                     for j in range(i + 1, 7)
 68: (8)                     for k in range(j + 1, 7)
 69: (4)                 ]
 70: (4)                 expected = [G.subgraph(x.split(",")) for x in expected]
 71: (4)                 actual = list(nx.all_triads(G))
 72: (4)                 assert all(any(nx.is_isomorphic(G1, G2) for G1 in expected) for G2 in
actual)
 73: (0)             def test_triad_type():
 74: (4)                 """Tests the triad_type function."""
 75: (4)                 G = nx.DiGraph({0: [], 1: [], 2: []})
 76: (4)                 assert nx.triad_type(G) == "003"
 77: (4)                 G = nx.DiGraph({0: [1], 1: [], 2: []})
 78: (4)                 assert nx.triad_type(G) == "012"
```

```
79: (4)                 G = nx.DiGraph([(0, 1), (0, 2)])
80: (4)                 assert nx.triad_type(G) == "021D"
81: (4)                 G = nx.DiGraph({0: [1], 1: [0], 2: []})
82: (4)                 assert nx.triad_type(G) == "102"
83: (4)                 G = nx.DiGraph([(0, 1), (2, 1)])
84: (4)                 assert nx.triad_type(G) == "021U"
85: (4)                 G = nx.DiGraph([(0, 1), (1, 2)])
86: (4)                 assert nx.triad_type(G) == "021C"
87: (4)                 G = nx.DiGraph([(0, 1), (1, 0), (2, 1)])
88: (4)                 assert nx.triad_type(G) == "111D"
89: (4)                 G = nx.DiGraph([(0, 1), (1, 0), (1, 2)])
90: (4)                 assert nx.triad_type(G) == "111U"
91: (4)                 G = nx.DiGraph([(0, 1), (1, 2), (0, 2)])
92: (4)                 assert nx.triad_type(G) == "030T"
93: (4)                 G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
94: (4)                 assert nx.triad_type(G) == "030C"
95: (4)                 G = nx.DiGraph([(0, 1), (1, 0), (2, 0), (0, 2)])
96: (4)                 assert nx.triad_type(G) == "201"
97: (4)                 G = nx.DiGraph([(0, 1), (1, 0), (2, 0), (2, 1)])
98: (4)                 assert nx.triad_type(G) == "120D"
99: (4)                 G = nx.DiGraph([(0, 1), (1, 0), (0, 2), (1, 2)])
100: (4)                assert nx.triad_type(G) == "120U"
101: (4)                G = nx.DiGraph([(0, 1), (1, 0), (0, 2), (2, 1)])
102: (4)                assert nx.triad_type(G) == "120C"
103: (4)                G = nx.DiGraph([(0, 1), (1, 0), (2, 1), (1, 2), (0, 2)])
104: (4)                assert nx.triad_type(G) == "210"
105: (4)                G = nx.DiGraph([(0, 1), (1, 0), (1, 2), (2, 1), (0, 2), (2, 0)])
106: (4)                assert nx.triad_type(G) == "300"
107: (0)            def test_triads_by_type():
108: (4)                """Tests the all_triplets function."""
109: (4)                G = nx.DiGraph()
110: (4)                G.add_edges_from(["01", "02", "03", "04", "05", "12", "16", "51", "56",
"65"])
111: (4)                all_triads = nx.all_triads(G)
112: (4)                expected = defaultdict(list)
113: (4)                for triad in all_triads:
114: (8)                    name = nx.triad_type(triad)
115: (8)                    expected[name].append(triad)
116: (4)                actual = nx.triads_by_type(G)
117: (4)                assert set(actual.keys()) == set(expected.keys())
118: (4)                for tri_type, actual_Gs in actual.items():
119: (8)                    expected_Gs = expected[tri_type]
120: (8)                    for a in actual_Gs:
121: (12)                       assert any(nx.is_isomorphic(a, e) for e in expected_Gs)
122: (0)            def test_random_triad():
123: (4)                """Tests the random_triad function"""
124: (4)                G = nx.karate_club_graph()
125: (4)                G = G.to_directed()
126: (4)                for i in range(100):
127: (8)                    assert nx.is_triad(nx.random_triad(G))
128: (4)                G = nx.DiGraph()
129: (4)                msg = "at least 3 nodes to form a triad"
130: (4)                with pytest.raises(nx.NetworkXError, match=msg):
131: (8)                    nx.random_triad(G)
132: (0)            def test_triadic_census_short_path_nodelist():
133: (4)                G = nx.path_graph("abc", create_using=nx.DiGraph)
134: (4)                expected = {"021C": 1}
135: (4)                for nl in ["a", "b", "c", "ab", "ac", "bc", "abc"]:
136: (8)                    triad_census = nx.triadic_census(G, nodelist=nl)
137: (8)                    assert expected == {typ: cnt for typ, cnt in triad_census.items() if
cnt > 0}
138: (0)            def test_triadic_census_correct_nodelist_values():
139: (4)                G = nx.path_graph(5, create_using=nx.DiGraph)
140: (4)                msg = r"nodelist includes duplicate nodes or nodes not in G"
141: (4)                with pytest.raises(ValueError, match=msg):
142: (8)                    nx.triadic_census(G, [1, 2, 2, 3])
143: (4)                with pytest.raises(ValueError, match=msg):
144: (8)                    nx.triadic_census(G, [1, 2, "a", 3])
145: (0)            def test_triadic_census_tiny_graphs():
```

```
146: (4)                tc = nx.triadic_census(nx.empty_graph(0, create_using=nx.DiGraph))
147: (4)                assert {} == {typ: cnt for typ, cnt in tc.items() if cnt > 0}
148: (4)                tc = nx.triadic_census(nx.empty_graph(1, create_using=nx.DiGraph))
149: (4)                assert {} == {typ: cnt for typ, cnt in tc.items() if cnt > 0}
150: (4)                tc = nx.triadic_census(nx.empty_graph(2, create_using=nx.DiGraph))
151: (4)                assert {} == {typ: cnt for typ, cnt in tc.items() if cnt > 0}
152: (4)                tc = nx.triadic_census(nx.DiGraph([(1, 2)]))
153: (4)                assert {} == {typ: cnt for typ, cnt in tc.items() if cnt > 0}
154: (0)            def test_triadic_census_selfloops():
155: (4)                GG = nx.path_graph("abc", create_using=nx.DiGraph)
156: (4)                expected = {"021C": 1}
157: (4)                for n in GG:
158: (8)                    G = GG.copy()
159: (8)                    G.add_edge(n, n)
160: (8)                    tc = nx.triadic_census(G)
161: (8)                    assert expected == {typ: cnt for typ, cnt in tc.items() if cnt > 0}
162: (4)                GG = nx.path_graph("abcde", create_using=nx.DiGraph)
163: (4)                tbt = nx.triads_by_type(GG)
164: (4)                for n in GG:
165: (8)                    GG.add_edge(n, n)
166: (4)                tc = nx.triadic_census(GG)
167: (4)                assert tc == {tt: len(tbt[tt]) for tt in tc}
168: (0)            def test_triadic_census_four_path():
169: (4)                G = nx.path_graph("abcd", create_using=nx.DiGraph)
170: (4)                expected = {"012": 2, "021C": 2}
171: (4)                triad_census = nx.triadic_census(G)
172: (4)                assert expected == {typ: cnt for typ, cnt in triad_census.items() if cnt >
0}
173: (0)            def test_triadic_census_four_path_nodelist():
174: (4)                G = nx.path_graph("abcd", create_using=nx.DiGraph)
175: (4)                expected_end = {"012": 2, "021C": 1}
176: (4)                expected_mid = {"012": 1, "021C": 2}
177: (4)                a_triad_census = nx.triadic_census(G, nodelist=["a"])
178: (4)                assert expected_end == {typ: cnt for typ, cnt in a_triad_census.items() if
cnt > 0}
179: (4)                b_triad_census = nx.triadic_census(G, nodelist=["b"])
180: (4)                assert expected_mid == {typ: cnt for typ, cnt in b_triad_census.items() if
cnt > 0}
181: (4)                c_triad_census = nx.triadic_census(G, nodelist=["c"])
182: (4)                assert expected_mid == {typ: cnt for typ, cnt in c_triad_census.items() if
cnt > 0}
183: (4)                d_triad_census = nx.triadic_census(G, nodelist=["d"])
184: (4)                assert expected_end == {typ: cnt for typ, cnt in d_triad_census.items() if
cnt > 0}
185: (0)            def test_triadic_census_nodelist():
186: (4)                """Tests the triadic_census function."""
187: (4)                G = nx.DiGraph()
188: (4)                G.add_edges_from(["01", "02", "03", "04", "05", "12", "16", "51", "56",
"65"])
189: (4)                expected = {
190: (8)                    "030T": 2,
191: (8)                    "120C": 1,
192: (8)                    "210": 0,
193: (8)                    "120U": 0,
194: (8)                    "012": 9,
195: (8)                    "102": 3,
196: (8)                    "021U": 0,
197: (8)                    "111U": 0,
198: (8)                    "003": 8,
199: (8)                    "030C": 0,
200: (8)                    "021D": 9,
201: (8)                    "201": 0,
202: (8)                    "111D": 1,
203: (8)                    "300": 0,
204: (8)                    "120D": 0,
205: (8)                    "021C": 2,
206: (4)                }
207: (4)                actual = {k: 0 for k in expected}
208: (4)                for node in G.nodes():
```

```
209: (8)                    node_triad_census = nx.triadic_census(G, nodelist=[node])
210: (8)                    for triad_key in expected:
211: (12)                       actual[triad_key] += node_triad_census[triad_key]
212: (4)                for k, v in actual.items():
213: (8)                    actual[k] //= 3
214: (4)                assert expected == actual
215: (0)            @pytest.mark.parametrize("N", [5, 10])
216: (0)            def test_triadic_census_on_random_graph(N):
217: (4)                G = nx.binomial_graph(N, 0.3, directed=True, seed=42)
218: (4)                tc1 = nx.triadic_census(G)
219: (4)                tbt = nx.triads_by_type(G)
220: (4)                tc2 = {tt: len(tbt[tt]) for tt in tc1}
221: (4)                assert tc1 == tc2
222: (4)                for n in G:
223: (8)                    tc1 = nx.triadic_census(G, nodelist={n})
224: (8)                    tc2 = {tt: sum(1 for t in tbt.get(tt, []) if n in t) for tt in tc1}
225: (8)                    assert tc1 == tc2
226: (4)                for ns in itertools.combinations(G, 2):
227: (8)                    ns = set(ns)
228: (8)                    tc1 = nx.triadic_census(G, nodelist=ns)
229: (8)                    tc2 = {
230: (12)                       tt: sum(1 for t in tbt.get(tt, []) if any(n in ns for n in t)) for
tt in tc1
231: (8)                    }
232: (8)                    assert tc1 == tc2
233: (4)                for ns in itertools.combinations(G, 3):
234: (8)                    ns = set(ns)
235: (8)                    tc1 = nx.triadic_census(G, nodelist=ns)
236: (8)                    tc2 = {
237: (12)                       tt: sum(1 for t in tbt.get(tt, []) if any(n in ns for n in t)) for
tt in tc1
238: (8)                    }
239: (8)                    assert tc1 == tc2
```

File 357 - test_tournament.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.tournament` module."""
2: (0)              from itertools import combinations
3: (0)              import pytest
4: (0)              from networkx import DiGraph
5: (0)              from networkx.algorithms.tournament import (
6: (4)                  hamiltonian_path,
7: (4)                  index_satisfying,
8: (4)                  is_reachable,
9: (4)                  is_strongly_connected,
10: (4)                 is_tournament,
11: (4)                 random_tournament,
12: (4)                 score_sequence,
13: (4)                 tournament_matrix,
14: (0)             )
15: (0)             def test_condition_not_satisfied():
16: (4)                 condition = lambda x: x > 0
17: (4)                 iter_in = [0]
18: (4)                 assert index_satisfying(iter_in, condition) == 1
19: (0)             def test_empty_iterable():
20: (4)                 condition = lambda x: x > 0
21: (4)                 with pytest.raises(ValueError):
22: (8)                     index_satisfying([], condition)
23: (0)             def test_is_tournament():
24: (4)                 G = DiGraph()
25: (4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0), (1, 3), (0, 2)])
26: (4)                 assert is_tournament(G)
27: (0)             def test_self_loops():
28: (4)                 """A tournament must have no self-loops."""
29: (4)                 G = DiGraph()
30: (4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0), (1, 3), (0, 2)])
31: (4)                 G.add_edge(0, 0)
```

```
(4)                 assert not is_tournament(G)
(0)             def test_missing_edges():
(4)                 """A tournament must not have any pair of nodes without at least
(4)                 one edge joining the pair.
(4)                 """
(4)                 G = DiGraph()
(4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0), (1, 3)])
(4)                 assert not is_tournament(G)
(0)             def test_bidirectional_edges():
(4)                 """A tournament must not have any pair of nodes with greater
(4)                 than one edge joining the pair.
(4)                 """
(4)                 G = DiGraph()
(4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0), (1, 3), (0, 2)])
(4)                 G.add_edge(1, 0)
(4)                 assert not is_tournament(G)
(0)             def test_graph_is_tournament():
(4)                 for _ in range(10):
(8)                     G = random_tournament(5)
(8)                     assert is_tournament(G)
(0)             def test_graph_is_tournament_seed():
(4)                 for _ in range(10):
(8)                     G = random_tournament(5, seed=1)
(8)                     assert is_tournament(G)
(0)             def test_graph_is_tournament_one_node():
(4)                 G = random_tournament(1)
(4)                 assert is_tournament(G)
(0)             def test_graph_is_tournament_zero_node():
(4)                 G = random_tournament(0)
(4)                 assert is_tournament(G)
(0)             def test_hamiltonian_empty_graph():
(4)                 path = hamiltonian_path(DiGraph())
(4)                 assert len(path) == 0
(0)             def test_path_is_hamiltonian():
(4)                 G = DiGraph()
(4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0), (1, 3), (0, 2)])
(4)                 path = hamiltonian_path(G)
(4)                 assert len(path) == 4
(4)                 assert all(v in G[u] for u, v in zip(path, path[1:]))
(0)             def test_hamiltonian_cycle():
(4)                 """Tests that :func:`networkx.tournament.hamiltonian_path`
(4)                 returns a Hamiltonian cycle when provided a strongly connected
(4)                 tournament.
(4)                 """
(4)                 G = DiGraph()
(4)                 G.add_edges_from([(0, 1), (1, 2), (2, 3), (3, 0), (1, 3), (0, 2)])
(4)                 path = hamiltonian_path(G)
(4)                 assert len(path) == 4
(4)                 assert all(v in G[u] for u, v in zip(path, path[1:]))
(4)                 assert path[0] in G[path[-1]]
(0)             def test_score_sequence_edge():
(4)                 G = DiGraph([(0, 1)])
(4)                 assert score_sequence(G) == [0, 1]
(0)             def test_score_sequence_triangle():
(4)                 G = DiGraph([(0, 1), (1, 2), (2, 0)])
(4)                 assert score_sequence(G) == [1, 1, 1]
(0)             def test_tournament_matrix():
(4)                 np = pytest.importorskip("numpy")
(4)                 pytest.importorskip("scipy")
(4)                 npt = np.testing
(4)                 G = DiGraph([(0, 1)])
(4)                 m = tournament_matrix(G)
(4)                 npt.assert_array_equal(m.todense(), np.array([[0, 1], [-1, 0]]))
(0)             def test_reachable_pair():
(4)                 """Tests for a reachable pair of nodes."""
(4)                 G = DiGraph([(0, 1), (1, 2), (2, 0)])
(4)                 assert is_reachable(G, 0, 2)
(0)             def test_same_node_is_reachable():
(4)                 """Tests that a node is always reachable from it."""
```

```
101: (4)                G = DiGraph(sorted(p) for p in combinations(range(10), 2))
102: (4)                assert all(is_reachable(G, v, v) for v in G)
103: (0)            def test_unreachable_pair():
104: (4)                """Tests for an unreachable pair of nodes."""
105: (4)                G = DiGraph([(0, 1), (0, 2), (1, 2)])
106: (4)                assert not is_reachable(G, 1, 0)
107: (0)            def test_is_strongly_connected():
108: (4)                """Tests for a strongly connected tournament."""
109: (4)                G = DiGraph([(0, 1), (1, 2), (2, 0)])
110: (4)                assert is_strongly_connected(G)
111: (0)            def test_not_strongly_connected():
112: (4)                """Tests for a tournament that is not strongly connected."""
113: (4)                G = DiGraph([(0, 1), (0, 2), (1, 2)])
114: (4)                assert not is_strongly_connected(G)
```

File 358 - test_voronoi.py:

```
1: (0)              import networkx as nx
2: (0)              from networkx.utils import pairwise
3: (0)              class TestVoronoiCells:
4: (4)                  """Unit tests for the Voronoi cells function."""
5: (4)                  def test_isolates(self):
6: (8)                      """Tests that a graph with isolated nodes has all isolates in
7: (8)                      one block of the partition.
8: (8)                      """
9: (8)                      G = nx.empty_graph(5)
10: (8)                     cells = nx.voronoi_cells(G, {0, 2, 4})
11: (8)                     expected = {0: {0}, 2: {2}, 4: {4}, "unreachable": {1, 3}}
12: (8)                     assert expected == cells
13: (4)                 def test_undirected_unweighted(self):
14: (8)                     G = nx.cycle_graph(6)
15: (8)                     cells = nx.voronoi_cells(G, {0, 3})
16: (8)                     expected = {0: {0, 1, 5}, 3: {2, 3, 4}}
17: (8)                     assert expected == cells
18: (4)                 def test_directed_unweighted(self):
19: (8)                     G = nx.DiGraph(pairwise(range(6), cyclic=True))
20: (8)                     cells = nx.voronoi_cells(G, {0, 3})
21: (8)                     expected = {0: {0, 1, 2}, 3: {3, 4, 5}}
22: (8)                     assert expected == cells
23: (4)                 def test_directed_inward(self):
24: (8)                     """Tests that reversing the graph gives the "inward" Voronoi
25: (8)                     partition.
26: (8)                     """
27: (8)                     G = nx.DiGraph(pairwise(range(6), cyclic=True))
28: (8)                     G = G.reverse(copy=False)
29: (8)                     cells = nx.voronoi_cells(G, {0, 3})
30: (8)                     expected = {0: {0, 4, 5}, 3: {1, 2, 3}}
31: (8)                     assert expected == cells
32: (4)                 def test_undirected_weighted(self):
33: (8)                     edges = [(0, 1, 10), (1, 2, 1), (2, 3, 1)]
34: (8)                     G = nx.Graph()
35: (8)                     G.add_weighted_edges_from(edges)
36: (8)                     cells = nx.voronoi_cells(G, {0, 3})
37: (8)                     expected = {0: {0}, 3: {1, 2, 3}}
38: (8)                     assert expected == cells
39: (4)                 def test_directed_weighted(self):
40: (8)                     edges = [(0, 1, 10), (1, 2, 1), (2, 3, 1), (3, 2, 1), (2, 1, 1)]
41: (8)                     G = nx.DiGraph()
42: (8)                     G.add_weighted_edges_from(edges)
43: (8)                     cells = nx.voronoi_cells(G, {0, 3})
44: (8)                     expected = {0: {0}, 3: {1, 2, 3}}
45: (8)                     assert expected == cells
46: (4)                 def test_multigraph_unweighted(self):
47: (8)                     """Tests that the Voronoi cells for a multigraph are the same as
48: (8)                     for a simple graph.
49: (8)                     """
50: (8)                     edges = [(0, 1), (1, 2), (2, 3)]
```

```
51: (8)                     G = nx.MultiGraph(2 * edges)
52: (8)                     H = nx.Graph(G)
53: (8)                     G_cells = nx.voronoi_cells(G, {0, 3})
54: (8)                     H_cells = nx.voronoi_cells(H, {0, 3})
55: (8)                     assert G_cells == H_cells
56: (4)                 def test_multidigraph_unweighted(self):
57: (8)                     edges = list(pairwise(range(6), cyclic=True))
58: (8)                     G = nx.MultiDiGraph(2 * edges)
59: (8)                     H = nx.DiGraph(G)
60: (8)                     G_cells = nx.voronoi_cells(G, {0, 3})
61: (8)                     H_cells = nx.voronoi_cells(H, {0, 3})
62: (8)                     assert G_cells == H_cells
63: (4)                 def test_multigraph_weighted(self):
64: (8)                     edges = [(0, 1, 10), (0, 1, 10), (1, 2, 1), (1, 2, 100), (2, 3, 1),
(2, 3, 100)]
65: (8)                     G = nx.MultiGraph()
66: (8)                     G.add_weighted_edges_from(edges)
67: (8)                     cells = nx.voronoi_cells(G, {0, 3})
68: (8)                     expected = {0: {0}, 3: {1, 2, 3}}
69: (8)                     assert expected == cells
70: (4)                 def test_multidigraph_weighted(self):
71: (8)                     edges = [
72: (12)                        (0, 1, 10),
73: (12)                        (0, 1, 10),
74: (12)                        (1, 2, 1),
75: (12)                        (2, 3, 1),
76: (12)                        (3, 2, 10),
77: (12)                        (3, 2, 1),
78: (12)                        (2, 1, 10),
79: (12)                        (2, 1, 1),
80: (8)                     ]
81: (8)                     G = nx.MultiDiGraph()
82: (8)                     G.add_weighted_edges_from(edges)
83: (8)                     cells = nx.voronoi_cells(G, {0, 3})
84: (8)                     expected = {0: {0}, 3: {1, 2, 3}}
85: (8)                     assert expected == cells

----------------------------------------

File 359 - test_vitality.py:

1: (0)              import networkx as nx
2: (0)              class TestClosenessVitality:
3: (4)                  def test_unweighted(self):
4: (8)                      G = nx.cycle_graph(3)
5: (8)                      vitality = nx.closeness_vitality(G)
6: (8)                      assert vitality == {0: 2, 1: 2, 2: 2}
7: (4)                  def test_weighted(self):
8: (8)                      G = nx.Graph()
9: (8)                      nx.add_cycle(G, [0, 1, 2], weight=2)
10: (8)                     vitality = nx.closeness_vitality(G, weight="weight")
11: (8)                     assert vitality == {0: 4, 1: 4, 2: 4}
12: (4)                 def test_unweighted_digraph(self):
13: (8)                     G = nx.DiGraph(nx.cycle_graph(3))
14: (8)                     vitality = nx.closeness_vitality(G)
15: (8)                     assert vitality == {0: 4, 1: 4, 2: 4}
16: (4)                 def test_weighted_digraph(self):
17: (8)                     G = nx.DiGraph()
18: (8)                     nx.add_cycle(G, [0, 1, 2], weight=2)
19: (8)                     nx.add_cycle(G, [2, 1, 0], weight=2)
20: (8)                     vitality = nx.closeness_vitality(G, weight="weight")
21: (8)                     assert vitality == {0: 8, 1: 8, 2: 8}
22: (4)                 def test_weighted_multidigraph(self):
23: (8)                     G = nx.MultiDiGraph()
24: (8)                     nx.add_cycle(G, [0, 1, 2], weight=2)
25: (8)                     nx.add_cycle(G, [2, 1, 0], weight=2)
26: (8)                     vitality = nx.closeness_vitality(G, weight="weight")
27: (8)                     assert vitality == {0: 8, 1: 8, 2: 8}
28: (4)                 def test_disconnecting_graph(self):
```

```
29: (8)                     """Tests that the closeness vitality of a node whose removal
30: (8)                     disconnects the graph is negative infinity.
31: (8)                     """
32: (8)                     G = nx.path_graph(3)
33: (8)                     assert nx.closeness_vitality(G, node=1) == -float("inf")
```

File 360 - test_walks.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.walks` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              pytest.importorskip("numpy")
5: (0)              pytest.importorskip("scipy")
6: (0)              def test_directed():
7: (4)                  G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
8: (4)                  num_walks = nx.number_of_walks(G, 3)
9: (4)                  expected = {0: {0: 1, 1: 0, 2: 0}, 1: {0: 0, 1: 1, 2: 0}, 2: {0: 0, 1: 0,
2: 1}}
10: (4)                 assert num_walks == expected
11: (0)             def test_undirected():
12: (4)                 G = nx.cycle_graph(3)
13: (4)                 num_walks = nx.number_of_walks(G, 3)
14: (4)                 expected = {0: {0: 2, 1: 3, 2: 3}, 1: {0: 3, 1: 2, 2: 3}, 2: {0: 3, 1: 3,
2: 2}}
15: (4)                 assert num_walks == expected
16: (0)             def test_non_integer_nodes():
17: (4)                 G = nx.DiGraph([("A", "B"), ("B", "C"), ("C", "A")])
18: (4)                 num_walks = nx.number_of_walks(G, 2)
19: (4)                 expected = {
20: (8)                     "A": {"A": 0, "B": 0, "C": 1},
21: (8)                     "B": {"A": 1, "B": 0, "C": 0},
22: (8)                     "C": {"A": 0, "B": 1, "C": 0},
23: (4)                 }
24: (4)                 assert num_walks == expected
25: (0)             def test_zero_length():
26: (4)                 G = nx.cycle_graph(3)
27: (4)                 num_walks = nx.number_of_walks(G, 0)
28: (4)                 expected = {0: {0: 1, 1: 0, 2: 0}, 1: {0: 0, 1: 1, 2: 0}, 2: {0: 0, 1: 0,
2: 1}}
29: (4)                 assert num_walks == expected
30: (0)             def test_negative_length_exception():
31: (4)                 G = nx.cycle_graph(3)
32: (4)                 with pytest.raises(ValueError):
33: (8)                     nx.number_of_walks(G, -1)
34: (0)             def test_hidden_weight_attr():
35: (4)                 G = nx.cycle_graph(3)
36: (4)                 G.add_edge(1, 2, weight=5)
37: (4)                 num_walks = nx.number_of_walks(G, 3)
38: (4)                 expected = {0: {0: 2, 1: 3, 2: 3}, 1: {0: 3, 1: 2, 2: 3}, 2: {0: 3, 1: 3,
2: 2}}
39: (4)                 assert num_walks == expected
```

File 361 - test_wiener.py:

```
1: (0)              import networkx as nx
2: (0)              def test_wiener_index_of_disconnected_graph():
3: (4)                  assert nx.wiener_index(nx.empty_graph(2)) == float("inf")
4: (0)              def test_wiener_index_of_directed_graph():
5: (4)                  G = nx.complete_graph(3)
6: (4)                  H = nx.DiGraph(G)
7: (4)                  assert (2 * nx.wiener_index(G)) == nx.wiener_index(H)
8: (0)              def test_wiener_index_of_complete_graph():
9: (4)                  n = 10
10: (4)                 G = nx.complete_graph(n)
11: (4)                 assert nx.wiener_index(G) == (n * (n - 1) / 2)
```

```
12: (0)             def test_wiener_index_of_path_graph():
13: (4)                 n = 9
14: (4)                 G = nx.path_graph(n)
15: (4)                 expected = 2 * sum(i * (n - i) for i in range(1, (n // 2) + 1))
16: (4)                 actual = nx.wiener_index(G)
17: (4)                 assert expected == actual
18: (0)             def test_schultz_and_gutman_index_of_disconnected_graph():
19: (4)                 n = 4
20: (4)                 G = nx.Graph()
21: (4)                 G.add_nodes_from(list(range(1, n + 1)))
22: (4)                 expected = float("inf")
23: (4)                 G.add_edge(1, 2)
24: (4)                 G.add_edge(3, 4)
25: (4)                 actual_1 = nx.schultz_index(G)
26: (4)                 actual_2 = nx.gutman_index(G)
27: (4)                 assert expected == actual_1
28: (4)                 assert expected == actual_2
29: (0)             def test_schultz_and_gutman_index_of_complete_bipartite_graph_1():
30: (4)                 n = 3
31: (4)                 m = 3
32: (4)                 cbg = nx.complete_bipartite_graph(n, m)
33: (4)                 expected_1 = n * m * (n + m) + 2 * n * (n - 1) * m + 2 * m * (m - 1) * n
34: (4)                 actual_1 = nx.schultz_index(cbg)
35: (4)                 expected_2 = n * m * (n * m) + n * (n - 1) * m * m + m * (m - 1) * n * n
36: (4)                 actual_2 = nx.gutman_index(cbg)
37: (4)                 assert expected_1 == actual_1
38: (4)                 assert expected_2 == actual_2
39: (0)             def test_schultz_and_gutman_index_of_complete_bipartite_graph_2():
40: (4)                 n = 2
41: (4)                 m = 5
42: (4)                 cbg = nx.complete_bipartite_graph(n, m)
43: (4)                 expected_1 = n * m * (n + m) + 2 * n * (n - 1) * m + 2 * m * (m - 1) * n
44: (4)                 actual_1 = nx.schultz_index(cbg)
45: (4)                 expected_2 = n * m * (n * m) + n * (n - 1) * m * m + m * (m - 1) * n * n
46: (4)                 actual_2 = nx.gutman_index(cbg)
47: (4)                 assert expected_1 == actual_1
48: (4)                 assert expected_2 == actual_2
49: (0)             def test_schultz_and_gutman_index_of_complete_graph():
50: (4)                 n = 5
51: (4)                 cg = nx.complete_graph(n)
52: (4)                 expected_1 = n * (n - 1) * (n - 1)
53: (4)                 actual_1 = nx.schultz_index(cg)
54: (4)                 assert expected_1 == actual_1
55: (4)                 expected_2 = n * (n - 1) * (n - 1) * (n - 1) / 2
56: (4)                 actual_2 = nx.gutman_index(cg)
57: (4)                 assert expected_2 == actual_2
58: (0)             def test_schultz_and_gutman_index_of_odd_cycle_graph():
59: (4)                 k = 5
60: (4)                 n = 2 * k + 1
61: (4)                 ocg = nx.cycle_graph(n)
62: (4)                 expected_1 = 2 * n * k * (k + 1)
63: (4)                 actual_1 = nx.schultz_index(ocg)
64: (4)                 expected_2 = 2 * n * k * (k + 1)
65: (4)                 actual_2 = nx.gutman_index(ocg)
66: (4)                 assert expected_1 == actual_1
67: (4)                 assert expected_2 == actual_2
```

File 362 - __init__.py:

```
1: (0)             from .beamsearch import *
2: (0)             from .breadth_first_search import *
3: (0)             from .depth_first_search import *
4: (0)             from .edgedfs import *
5: (0)             from .edgebfs import *
```

File 363 - beamsearch.py:

```
1: (0)              """Basic algorithms for breadth-first searching the nodes of a graph."""
2: (0)              import networkx as nx
3: (0)              __all__ = ["bfs_beam_edges"]
4: (0)              @nx._dispatchable
5: (0)              def bfs_beam_edges(G, source, value, width=None):
6: (4)                  """Iterates over edges in a beam search.
7: (4)                  The beam search is a generalized breadth-first search in which only
8: (4)                  the "best" *w* neighbors of the current node are enqueued, where *w*
9: (4)                  is the beam width and "best" is an application-specific
10: (4)                 heuristic. In general, a beam search with a small beam width might
11: (4)                 not visit each node in the graph.
12: (4)                 .. note::
13: (7)                    With the default value of ``width=None`` or `width` greater than the
14: (7)                    maximum degree of the graph, this function equates to a slower
15: (7)                    version of
`~networkx.algorithms.traversal.breadth_first_search.bfs_edges`.
16: (7)                    All nodes will be visited, though the order of the reported edges may
17: (7)                    vary. In such cases, `value` has no effect - consider using `bfs_edges`
18: (7)                    directly instead.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : NetworkX graph
22: (4)                 source : node
23: (8)                     Starting node for the breadth-first search; this function
24: (8)                     iterates over only those edges in the component reachable from
25: (8)                     this node.
26: (4)                 value : function
27: (8)                     A function that takes a node of the graph as input and returns a
28: (8)                     real number indicating how "good" it is. A higher value means it
29: (8)                     is more likely to be visited sooner during the search. When
30: (8)                     visiting a new node, only the `width` neighbors with the highest
31: (8)                     `value` are enqueued (in decreasing order of `value`).
32: (4)                 width : int (default = None)
33: (8)                     The beam width for the search. This is the number of neighbors
34: (8)                     (ordered by `value`) to enqueue when visiting each new node.
35: (4)                 Yields
36: (4)                 ------
37: (4)                 edge
38: (8)                     Edges in the beam search starting from `source`, given as a pair
39: (8)                     of nodes.
40: (4)                 Examples
41: (4)                 --------
42: (4)                 To give nodes with, for example, a higher centrality precedence
43: (4)                 during the search, set the `value` function to return the centrality
44: (4)                 value of the node:
45: (4)                 >>> G = nx.karate_club_graph()
46: (4)                 >>> centrality = nx.eigenvector_centrality(G)
47: (4)                 >>> list(nx.bfs_beam_edges(G, source=0, value=centrality.get, width=3))
48: (4)                 [(0, 2), (0, 1), (0, 8), (2, 32), (1, 13), (8, 33)]
49: (4)                 """
50: (4)                 if width is None:
51: (8)                     width = len(G)
52: (4)                 def successors(v):
53: (8)                     """Returns a list of the best neighbors of a node.
54: (8)                     `v` is a node in the graph `G`.
55: (8)                     The "best" neighbors are chosen according to the `value`
56: (8)                     function (higher is better). Only the `width` best neighbors of
57: (8)                     `v` are returned.
58: (8)                     """
59: (8)                     return iter(sorted(G.neighbors(v), key=value, reverse=True)[:width])
60: (4)                 yield from nx.generic_bfs_edges(G, source, successors)
```

File 364 - breadth_first_search.py:

```
1: (0)              """Basic algorithms for breadth-first searching the nodes of a graph."""
```

```
2: (0)              from collections import deque
3: (0)              import networkx as nx
4: (0)              __all__ = [
5: (4)                  "bfs_edges",
6: (4)                  "bfs_tree",
7: (4)                  "bfs_predecessors",
8: (4)                  "bfs_successors",
9: (4)                  "descendants_at_distance",
10: (4)                 "bfs_layers",
11: (4)                 "bfs_labeled_edges",
12: (4)                 "generic_bfs_edges",
13: (0)             ]
14: (0)             @nx._dispatchable
15: (0)             def generic_bfs_edges(G, source, neighbors=None, depth_limit=None,
sort_neighbors=None):
16: (4)                 """Iterate over edges in a breadth-first search.
17: (4)                 The breadth-first search begins at `source` and enqueues the
18: (4)                 neighbors of newly visited nodes specified by the `neighbors`
19: (4)                 function.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : NetworkX graph
23: (4)                 source : node
24: (8)                     Starting node for the breadth-first search; this function
25: (8)                     iterates over only those edges in the component reachable from
26: (8)                     this node.
27: (4)                 neighbors : function
28: (8)                     A function that takes a newly visited node of the graph as input
29: (8)                     and returns an *iterator* (not just a list) of nodes that are
30: (8)                     neighbors of that node with custom ordering. If not specified, this is
31: (8)                     just the ``G.neighbors`` method, but in general it can be any function
32: (8)                     that returns an iterator over some or all of the neighbors of a
33: (8)                     given node, in any order.
34: (4)                 depth_limit : int, optional(default=len(G))
35: (8)                     Specify the maximum search depth.
36: (4)                 sort_neighbors : Callable (default=None)
37: (8)                     .. deprecated:: 3.2
38: (11)                       The sort_neighbors parameter is deprecated and will be removed in
39: (11)                       version 3.4. A custom (e.g. sorted) ordering of neighbors can be
40: (11)                       specified with the `neighbors` parameter.
41: (8)                     A function that takes an iterator over nodes as the input, and
42: (8)                     returns an iterable of the same nodes with a custom ordering.
43: (8)                     For example, `sorted` will sort the nodes in increasing order.
44: (4)                 Yields
45: (4)                 ------
46: (4)                 edge
47: (8)                     Edges in the breadth-first search starting from `source`.
48: (4)                 Examples
49: (4)                 --------
50: (4)                 >>> G = nx.path_graph(7)
51: (4)                 >>> list(nx.generic_bfs_edges(G, source=0))
52: (4)                 [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
53: (4)                 >>> list(nx.generic_bfs_edges(G, source=2))
54: (4)                 [(2, 1), (2, 3), (1, 0), (3, 4), (4, 5), (5, 6)]
55: (4)                 >>> list(nx.generic_bfs_edges(G, source=2, depth_limit=2))
56: (4)                 [(2, 1), (2, 3), (1, 0), (3, 4)]
57: (4)                 The `neighbors` param can be used to specify the visitation order of each
58: (4)                 node's neighbors generically. In the following example, we modify the
default
59: (4)                 neighbor to return *odd* nodes first:
60: (4)                 >>> def odd_first(n):
61: (4)                 ...     return sorted(G.neighbors(n), key=lambda x: x % 2, reverse=True)
62: (4)                 >>> G = nx.star_graph(5)
63: (4)                 >>> list(nx.generic_bfs_edges(G, source=0))  # Default neighbor ordering
64: (4)                 [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]
65: (4)                 >>> list(nx.generic_bfs_edges(G, source=0, neighbors=odd_first))
66: (4)                 [(0, 1), (0, 3), (0, 5), (0, 2), (0, 4)]
67: (4)                 Notes
68: (4)                 -----
```

```
69: (4)                 This implementation is from `PADS`_, which was in the public domain
70: (4)                 when it was first accessed in July, 2004.  The modifications
71: (4)                 to allow depth limits are based on the Wikipedia article
72: (4)                 "`Depth-limited-search`_".
73: (4)                 .. _PADS: http://www.ics.uci.edu/~eppstein/PADS/BFS.py
74: (4)                 .. _Depth-limited-search: https://en.wikipedia.org/wiki/Depth-
limited_search
75: (4)                 """
76: (4)                 if neighbors is None:
77: (8)                     neighbors = G.neighbors
78: (4)                 if sort_neighbors is not None:
79: (8)                     import warnings
80: (8)                     warnings.warn(
81: (12)                        (
82: (16)                            "The sort_neighbors parameter is deprecated and will be
removed\n"
83: (16)                            "in NetworkX 3.4, use the neighbors parameter instead."
84: (12)                        ),
85: (12)                        DeprecationWarning,
86: (12)                        stacklevel=2,
87: (8)                     )
88: (8)                     _neighbors = neighbors
89: (8)                     neighbors = lambda node: iter(sort_neighbors(_neighbors(node)))
90: (4)                 if depth_limit is None:
91: (8)                     depth_limit = len(G)
92: (4)                 seen = {source}
93: (4)                 n = len(G)
94: (4)                 depth = 0
95: (4)                 next_parents_children = [(source, neighbors(source))]
96: (4)                 while next_parents_children and depth < depth_limit:
97: (8)                     this_parents_children = next_parents_children
98: (8)                     next_parents_children = []
99: (8)                     for parent, children in this_parents_children:
100: (12)                       for child in children:
101: (16)                           if child not in seen:
102: (20)                               seen.add(child)
103: (20)                               next_parents_children.append((child, neighbors(child)))
104: (20)                               yield parent, child
105: (12)                       if len(seen) == n:
106: (16)                           return
107: (8)                    depth += 1
108: (0)            @nx._dispatchable
109: (0)            def bfs_edges(G, source, reverse=False, depth_limit=None,
sort_neighbors=None):
110: (4)                """Iterate over edges in a breadth-first-search starting at source.
111: (4)                Parameters
112: (4)                ----------
113: (4)                G : NetworkX graph
114: (4)                source : node
115: (7)                   Specify starting node for breadth-first search; this function
116: (7)                   iterates over only those edges in the component reachable from
117: (7)                   this node.
118: (4)                reverse : bool, optional
119: (7)                   If True traverse a directed graph in the reverse direction
120: (4)                depth_limit : int, optional(default=len(G))
121: (8)                    Specify the maximum search depth
122: (4)                sort_neighbors : function (default=None)
123: (8)                    A function that takes an iterator over nodes as the input, and
124: (8)                    returns an iterable of the same nodes with a custom ordering.
125: (8)                    For example, `sorted` will sort the nodes in increasing order.
126: (4)                Yields
127: (4)                ------
128: (4)                edge: 2-tuple of nodes
129: (7)                   Yields edges resulting from the breadth-first search.
130: (4)                Examples
131: (4)                --------
132: (4)                To get the edges in a breadth-first search::
133: (8)                    >>> G = nx.path_graph(3)
134: (8)                    >>> list(nx.bfs_edges(G, 0))
```

```
(8)                 [(0, 1), (1, 2)]
(8)                 >>> list(nx.bfs_edges(G, source=0, depth_limit=1))
(8)                 [(0, 1)]
(4)             To get the nodes in a breadth-first search order::
(8)                 >>> G = nx.path_graph(3)
(8)                 >>> root = 2
(8)                 >>> edges = nx.bfs_edges(G, root)
(8)                 >>> nodes = [root] + [v for u, v in edges]
(8)                 >>> nodes
(8)                 [2, 1, 0]
(4)             Notes
(4)             -----
(4)             The naming of this function is very similar to
(4)             :func:`~networkx.algorithms.traversal.edgebfs.edge_bfs`. The difference
(4)             is that ``edge_bfs`` yields edges even if they extend back to an already
(4)             explored node while this generator yields the edges of the tree that
results
(4)             from a breadth-first-search (BFS) so no edges are reported if they extend
(4)             to already explored nodes. That means ``edge_bfs`` reports all edges while
(4)             ``bfs_edges`` only reports those traversed by a node-based BFS. Yet
another
(4)             description is that ``bfs_edges`` reports the edges traversed during BFS
(4)             while ``edge_bfs`` reports all edges in the order they are explored.
(4)             Based on the breadth-first search implementation in PADS [1]_
(4)             by D. Eppstein, July 2004; with modifications to allow depth limits
(4)             as described in [2]_.
(4)             References
(4)             ----------
(4)             .. [1] http://www.ics.uci.edu/~eppstein/PADS/BFS.py.
(4)             .. [2] https://en.wikipedia.org/wiki/Depth-limited_search
(4)             See Also
(4)             --------
(4)             bfs_tree
(4)             :func:`~networkx.algorithms.traversal.depth_first_search.dfs_edges`
(4)             :func:`~networkx.algorithms.traversal.edgebfs.edge_bfs`
(4)             """
(4)             if reverse and G.is_directed():
(8)                 successors = G.predecessors
(4)             else:
(8)                 successors = G.neighbors
(4)             if sort_neighbors is not None:
(8)                 yield from generic_bfs_edges(
(12)                    G, source, lambda node: iter(sort_neighbors(successors(node))),
depth_limit
(8)                 )
(4)             else:
(8)                 yield from generic_bfs_edges(G, source, successors, depth_limit)
(0)         @nx._dispatchable(returns_graph=True)
(0)         def bfs_tree(G, source, reverse=False, depth_limit=None, sort_neighbors=None):
(4)             """Returns an oriented tree constructed from of a breadth-first-search
(4)             starting at source.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source : node
(7)                Specify starting node for breadth-first search
(4)             reverse : bool, optional
(7)                If True traverse a directed graph in the reverse direction
(4)             depth_limit : int, optional(default=len(G))
(8)                 Specify the maximum search depth
(4)             sort_neighbors : function (default=None)
(8)                 A function that takes an iterator over nodes as the input, and
(8)                 returns an iterable of the same nodes with a custom ordering.
(8)                 For example, `sorted` will sort the nodes in increasing order.
(4)             Returns
(4)             -------
(4)             T: NetworkX DiGraph
(7)                An oriented tree
(4)             Examples
```

```
    --------
    >>> G = nx.path_graph(3)
    >>> list(nx.bfs_tree(G, 1).edges())
    [(1, 0), (1, 2)]
    >>> H = nx.Graph()
    >>> nx.add_path(H, [0, 1, 2, 3, 4, 5, 6])
    >>> nx.add_path(H, [2, 7, 8, 9, 10])
    >>> sorted(list(nx.bfs_tree(H, source=3, depth_limit=3).edges()))
    [(1, 0), (2, 1), (2, 7), (3, 2), (3, 4), (4, 5), (5, 6), (7, 8)]
    Notes
    -----
    Based on http://www.ics.uci.edu/~eppstein/PADS/BFS.py
    by D. Eppstein, July 2004. The modifications
    to allow depth limits based on the Wikipedia article
    "`Depth-limited-search`_".
    .. _Depth-limited-search: https://en.wikipedia.org/wiki/Depth-limited_search
    See Also
    --------
    dfs_tree
    bfs_edges
    edge_bfs
    """
    T = nx.DiGraph()
    T.add_node(source)
    edges_gen = bfs_edges(
        G,
        source,
        reverse=reverse,
        depth_limit=depth_limit,
        sort_neighbors=sort_neighbors,
    )
    T.add_edges_from(edges_gen)
    return T
@nx._dispatchable
def bfs_predecessors(G, source, depth_limit=None, sort_neighbors=None):
    """Returns an iterator of predecessors in breadth-first-search from source.
    Parameters
    ----------
    G : NetworkX graph
    source : node
       Specify starting node for breadth-first search
    depth_limit : int, optional(default=len(G))
        Specify the maximum search depth
    sort_neighbors : function (default=None)
        A function that takes an iterator over nodes as the input, and
        returns an iterable of the same nodes with a custom ordering.
        For example, `sorted` will sort the nodes in increasing order.
    Returns
    -------
    pred: iterator
        (node, predecessor) iterator where `predecessor` is the predecessor of
        `node` in a breadth first search starting from `source`.
    Examples
    --------
    >>> G = nx.path_graph(3)
    >>> dict(nx.bfs_predecessors(G, 0))
    {1: 0, 2: 1}
    >>> H = nx.Graph()
    >>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
    >>> dict(nx.bfs_predecessors(H, 0))
    {1: 0, 2: 0, 3: 1, 4: 1, 5: 2, 6: 2}
    >>> M = nx.Graph()
    >>> nx.add_path(M, [0, 1, 2, 3, 4, 5, 6])
    >>> nx.add_path(M, [2, 7, 8, 9, 10])
    >>> sorted(nx.bfs_predecessors(M, source=1, depth_limit=3))
    [(0, 1), (2, 1), (3, 2), (4, 3), (7, 2), (8, 7)]
    >>> N = nx.DiGraph()
```

```
(4)                >>> nx.add_path(N, [0, 1, 2, 3, 4, 7])
(4)                >>> nx.add_path(N, [3, 5, 6, 7])
(4)                >>> sorted(nx.bfs_predecessors(N, source=2))
(4)                [(3, 2), (4, 3), (5, 3), (6, 5), (7, 4)]
(4)                Notes
(4)                -----
(4)                Based on http://www.ics.uci.edu/~eppstein/PADS/BFS.py
(4)                by D. Eppstein, July 2004. The modifications
(4)                to allow depth limits based on the Wikipedia article
(4)                "`Depth-limited-search`_".
(4)                .. _Depth-limited-search: https://en.wikipedia.org/wiki/Depth-limited_search
(4)                See Also
(4)                --------
(4)                bfs_tree
(4)                bfs_edges
(4)                edge_bfs
(4)                """
(4)                for s, t in bfs_edges(
(8)                    G, source, depth_limit=depth_limit, sort_neighbors=sort_neighbors
(4)                ):
(8)                    yield (t, s)
(0)            @nx._dispatchable
(0)            def bfs_successors(G, source, depth_limit=None, sort_neighbors=None):
(4)                """Returns an iterator of successors in breadth-first-search from source.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX graph
(4)                source : node
(7)                   Specify starting node for breadth-first search
(4)                depth_limit : int, optional(default=len(G))
(8)                    Specify the maximum search depth
(4)                sort_neighbors : function (default=None)
(8)                    A function that takes an iterator over nodes as the input, and
(8)                    returns an iterable of the same nodes with a custom ordering.
(8)                    For example, `sorted` will sort the nodes in increasing order.
(4)                Returns
(4)                -------
(4)                succ: iterator
(7)                   (node, successors) iterator where `successors` is the non-empty list of
(7)                   successors of `node` in a breadth first search from `source`.
(7)                   To appear in the iterator, `node` must have successors.
(4)                Examples
(4)                --------
(4)                >>> G = nx.path_graph(3)
(4)                >>> dict(nx.bfs_successors(G, 0))
(4)                {0: [1], 1: [2]}
(4)                >>> H = nx.Graph()
(4)                >>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
(4)                >>> dict(nx.bfs_successors(H, 0))
(4)                {0: [1, 2], 1: [3, 4], 2: [5, 6]}
(4)                >>> G = nx.Graph()
(4)                >>> nx.add_path(G, [0, 1, 2, 3, 4, 5, 6])
(4)                >>> nx.add_path(G, [2, 7, 8, 9, 10])
(4)                >>> dict(nx.bfs_successors(G, source=1, depth_limit=3))
(4)                {1: [0, 2], 2: [3, 7], 3: [4], 7: [8]}
(4)                >>> G = nx.DiGraph()
(4)                >>> nx.add_path(G, [0, 1, 2, 3, 4, 5])
(4)                >>> dict(nx.bfs_successors(G, source=3))
(4)                {3: [4], 4: [5]}
(4)                Notes
(4)                -----
(4)                Based on http://www.ics.uci.edu/~eppstein/PADS/BFS.py
(4)                by D. Eppstein, July 2004.The modifications
(4)                to allow depth limits based on the Wikipedia article
(4)                "`Depth-limited-search`_".
(4)                .. _Depth-limited-search: https://en.wikipedia.org/wiki/Depth-limited_search
(4)                See Also
```

```
(4)                 --------
(4)                 bfs_tree
(4)                 bfs_edges
(4)                 edge_bfs
(4)                 """
(4)                 parent = source
(4)                 children = []
(4)                 for p, c in bfs_edges(
(8)                     G, source, depth_limit=depth_limit, sort_neighbors=sort_neighbors
(4)                 ):
(8)                     if p == parent:
(12)                        children.append(c)
(12)                        continue
(8)                     yield (parent, children)
(8)                     children = [c]
(8)                     parent = p
(4)                 yield (parent, children)
(0)             @nx._dispatchable
(0)             def bfs_layers(G, sources):
(4)                 """Returns an iterator of all the layers in breadth-first search traversal.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     A graph over which to find the layers using breadth-first search.
(4)                 sources : node in `G` or list of nodes in `G`
(8)                     Specify starting nodes for single source or multiple sources breadth-first search
(4)                 Yields
(4)                 ------
(4)                 layer: list of nodes
(8)                     Yields list of nodes at the same distance from sources
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> dict(enumerate(nx.bfs_layers(G, [0, 4])))
(4)                 {0: [0, 4], 1: [1, 3], 2: [2]}
(4)                 >>> H = nx.Graph()
(4)                 >>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
(4)                 >>> dict(enumerate(nx.bfs_layers(H, [1])))
(4)                 {0: [1], 1: [0, 3, 4], 2: [2], 3: [5, 6]}
(4)                 >>> dict(enumerate(nx.bfs_layers(H, [1, 6])))
(4)                 {0: [1, 6], 1: [0, 3, 4, 2], 2: [5]}
(4)                 """
(4)                 if sources in G:
(8)                     sources = [sources]
(4)                 current_layer = list(sources)
(4)                 visited = set(sources)
(4)                 for source in current_layer:
(8)                     if source not in G:
(12)                        raise nx.NetworkXError(f"The node {source} is not in the graph.")
(4)                 while current_layer:
(8)                     yield current_layer
(8)                     next_layer = []
(8)                     for node in current_layer:
(12)                        for child in G[node]:
(16)                            if child not in visited:
(20)                                visited.add(child)
(20)                                next_layer.append(child)
(8)                     current_layer = next_layer
(0)             REVERSE_EDGE = "reverse"
(0)             TREE_EDGE = "tree"
(0)             FORWARD_EDGE = "forward"
(0)             LEVEL_EDGE = "level"
(0)             @nx._dispatchable
(0)             def bfs_labeled_edges(G, sources):
(4)                 """Iterate over edges in a breadth-first search (BFS) labeled by type.
(4)                 We generate triple of the form (*u*, *v*, *d*), where (*u*, *v*) is the
(4)                 edge being explored in the breadth-first search and *d* is one of the
```

```
    strings 'tree', 'forward', 'level', or 'reverse'.  A 'tree' edge is one in
    which *v* is first discovered and placed into the layer below *u*.  A
    'forward' edge is one in which *u* is on the layer above *v* and *v* has
    already been discovered.  A 'level' edge is one in which both *u* and *v*
    occur on the same layer.  A 'reverse' edge is one in which *u* is on a layer

    below *v*.
    We emit each edge exactly once.  In an undirected graph, 'reverse' edges do

    not occur, because each is discovered either as a 'tree' or 'forward' edge.
    Parameters
    ----------
    G : NetworkX graph
        A graph over which to find the layers using breadth-first search.
    sources : node in `G` or list of nodes in `G`
        Starting nodes for single source or multiple sources breadth-first search

    Yields
    ------
    edges: generator
       A generator of triples (*u*, *v*, *d*) where (*u*, *v*) is the edge being

       explored and *d* is described above.
    Examples
    --------
    >>> G = nx.cycle_graph(4, create_using=nx.DiGraph)
    >>> list(nx.bfs_labeled_edges(G, 0))
    [(0, 1, 'tree'), (1, 2, 'tree'), (2, 3, 'tree'), (3, 0, 'reverse')]
    >>> G = nx.complete_graph(3)
    >>> list(nx.bfs_labeled_edges(G, 0))
    [(0, 1, 'tree'), (0, 2, 'tree'), (1, 2, 'level')]
    >>> list(nx.bfs_labeled_edges(G, [0, 1]))
    [(0, 1, 'level'), (0, 2, 'tree'), (1, 2, 'forward')]
    """
    if sources in G:
        sources = [sources]
    neighbors = G._adj
    directed = G.is_directed()
    visited = set()
    visit = visited.discard if directed else visited.add
    depth = {s: 0 for s in sources}
    queue = deque(depth.items())
    push = queue.append
    pop = queue.popleft
    while queue:
        u, du = pop()
        for v in neighbors[u]:
            if v not in depth:
                depth[v] = dv = du + 1
                push((v, dv))
                yield u, v, TREE_EDGE
            else:
                dv = depth[v]
                if du == dv:
                    if v not in visited:
                        yield u, v, LEVEL_EDGE
                elif du < dv:
                    yield u, v, FORWARD_EDGE
                elif directed:
                    yield u, v, REVERSE_EDGE
        visit(u)
@nx._dispatchable
def descendants_at_distance(G, source, distance):
    """Returns all nodes at a fixed `distance` from `source` in `G`.
    Parameters
    ----------
    G : NetworkX graph
        A graph
```

```
466: (4)                source : node in `G`
467: (4)                distance : the distance of the wanted nodes from `source`
468: (4)                Returns
469: (4)                -------
470: (4)                set()
471: (8)                    The descendants of `source` in `G` at the given `distance` from
`source`
472: (4)                Examples
473: (4)                --------
474: (4)                >>> G = nx.path_graph(5)
475: (4)                >>> nx.descendants_at_distance(G, 2, 2)
476: (4)                {0, 4}
477: (4)                >>> H = nx.DiGraph()
478: (4)                >>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
479: (4)                >>> nx.descendants_at_distance(H, 0, 2)
480: (4)                {3, 4, 5, 6}
481: (4)                >>> nx.descendants_at_distance(H, 5, 0)
482: (4)                {5}
483: (4)                >>> nx.descendants_at_distance(H, 5, 1)
484: (4)                set()
485: (4)                """
486: (4)                if source not in G:
487: (8)                    raise nx.NetworkXError(f"The node {source} is not in the graph.")
488: (4)                bfs_generator = nx.bfs_layers(G, source)
489: (4)                for i, layer in enumerate(bfs_generator):
490: (8)                    if i == distance:
491: (12)                       return set(layer)
492: (4)                return set()
```

File 365 - depth_first_search.py:

```
1: (0)              """Basic algorithms for depth-first searching the nodes of a graph."""
2: (0)              from collections import defaultdict
3: (0)              import networkx as nx
4: (0)              __all__ = [
5: (4)                  "dfs_edges",
6: (4)                  "dfs_tree",
7: (4)                  "dfs_predecessors",
8: (4)                  "dfs_successors",
9: (4)                  "dfs_preorder_nodes",
10: (4)                 "dfs_postorder_nodes",
11: (4)                 "dfs_labeled_edges",
12: (0)             ]
13: (0)             @nx._dispatchable
14: (0)             def dfs_edges(G, source=None, depth_limit=None, *, sort_neighbors=None):
15: (4)                 """Iterate over edges in a depth-first-search (DFS).
16: (4)                 Perform a depth-first-search over the nodes of `G` and yield
17: (4)                 the edges in order. This may not generate all edges in `G`
18: (4)                 (see `~networkx.algorithms.traversal.edgedfs.edge_dfs`).
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : NetworkX graph
22: (4)                 source : node, optional
23: (7)                    Specify starting node for depth-first search and yield edges in
24: (7)                    the component reachable from source.
25: (4)                 depth_limit : int, optional (default=len(G))
26: (7)                    Specify the maximum search depth.
27: (4)                 sort_neighbors : function (default=None)
28: (8)                     A function that takes an iterator over nodes as the input, and
29: (8)                     returns an iterable of the same nodes with a custom ordering.
30: (8)                     For example, `sorted` will sort the nodes in increasing order.
31: (4)                 Yields
32: (4)                 ------
33: (4)                 edge: 2-tuple of nodes
34: (7)                    Yields edges resulting from the depth-first-search.
35: (4)                 Examples
36: (4)                 --------
```

```
 37: (4)                >>> G = nx.path_graph(5)
 38: (4)                >>> list(nx.dfs_edges(G, source=0))
 39: (4)                [(0, 1), (1, 2), (2, 3), (3, 4)]
 40: (4)                >>> list(nx.dfs_edges(G, source=0, depth_limit=2))
 41: (4)                [(0, 1), (1, 2)]
 42: (4)                Notes
 43: (4)                -----
 44: (4)                If a source is not specified then a source is chosen arbitrarily and
 45: (4)                repeatedly until all components in the graph are searched.
 46: (4)                The implementation of this function is adapted from David Eppstein's
 47: (4)                depth-first search function in PADS [1]_, with modifications
 48: (4)                to allow depth limits based on the Wikipedia article
 49: (4)                "Depth-limited search" [2]_.
 50: (4)                See Also
 51: (4)                --------
 52: (4)                dfs_preorder_nodes
 53: (4)                dfs_postorder_nodes
 54: (4)                dfs_labeled_edges
 55: (4)                :func:`~networkx.algorithms.traversal.edgedfs.edge_dfs`
 56: (4)                :func:`~networkx.algorithms.traversal.breadth_first_search.bfs_edges`
 57: (4)                References
 58: (4)                ----------
 59: (4)                .. [1] http://www.ics.uci.edu/~eppstein/PADS
 60: (4)                .. [2] https://en.wikipedia.org/wiki/Depth-limited_search
 61: (4)                """
 62: (4)                if source is None:
 63: (8)                    nodes = G
 64: (4)                else:
 65: (8)                    nodes = [source]
 66: (4)                if depth_limit is None:
 67: (8)                    depth_limit = len(G)
 68: (4)                get_children = (
 69: (8)                    G.neighbors
 70: (8)                    if sort_neighbors is None
 71: (8)                    else lambda n: iter(sort_neighbors(G.neighbors(n)))
 72: (4)                )
 73: (4)                visited = set()
 74: (4)                for start in nodes:
 75: (8)                    if start in visited:
 76: (12)                       continue
 77: (8)                    visited.add(start)
 78: (8)                    stack = [(start, get_children(start))]
 79: (8)                    depth_now = 1
 80: (8)                    while stack:
 81: (12)                       parent, children = stack[-1]
 82: (12)                       for child in children:
 83: (16)                           if child not in visited:
 84: (20)                               yield parent, child
 85: (20)                               visited.add(child)
 86: (20)                               if depth_now < depth_limit:
 87: (24)                                   stack.append((child, get_children(child)))
 88: (24)                                   depth_now += 1
 89: (24)                                   break
 90: (12)                       else:
 91: (16)                           stack.pop()
 92: (16)                           depth_now -= 1
 93: (0)            @nx._dispatchable(returns_graph=True)
 94: (0)            def dfs_tree(G, source=None, depth_limit=None, *, sort_neighbors=None):
 95: (4)                """Returns oriented tree constructed from a depth-first-search from
source.
 96: (4)                Parameters
 97: (4)                ----------
 98: (4)                G : NetworkX graph
 99: (4)                source : node, optional
100: (7)                   Specify starting node for depth-first search.
101: (4)                depth_limit : int, optional (default=len(G))
102: (7)                   Specify the maximum search depth.
103: (4)                sort_neighbors : function (default=None)
104: (8)                    A function that takes an iterator over nodes as the input, and
```

```
105: (8)                    returns an iterable of the same nodes with a custom ordering.
106: (8)                    For example, `sorted` will sort the nodes in increasing order.
107: (4)                Returns
108: (4)                -------
109: (4)                T : NetworkX DiGraph
110: (7)                   An oriented tree
111: (4)                Examples
112: (4)                --------
113: (4)                >>> G = nx.path_graph(5)
114: (4)                >>> T = nx.dfs_tree(G, source=0, depth_limit=2)
115: (4)                >>> list(T.edges())
116: (4)                [(0, 1), (1, 2)]
117: (4)                >>> T = nx.dfs_tree(G, source=0)
118: (4)                >>> list(T.edges())
119: (4)                [(0, 1), (1, 2), (2, 3), (3, 4)]
120: (4)                See Also
121: (4)                --------
122: (4)                dfs_preorder_nodes
123: (4)                dfs_postorder_nodes
124: (4)                dfs_labeled_edges
125: (4)                :func:`~networkx.algorithms.traversal.edgedfs.edge_dfs`
126: (4)                :func:`~networkx.algorithms.traversal.breadth_first_search.bfs_tree`
127: (4)                """
128: (4)                T = nx.DiGraph()
129: (4)                if source is None:
130: (8)                    T.add_nodes_from(G)
131: (4)                else:
132: (8)                    T.add_node(source)
133: (4)                T.add_edges_from(dfs_edges(G, source, depth_limit,
sort_neighbors=sort_neighbors))
134: (4)                return T
135: (0)            @nx._dispatchable
136: (0)            def dfs_predecessors(G, source=None, depth_limit=None, *,
sort_neighbors=None):
137: (4)                """Returns dictionary of predecessors in depth-first-search from source.
138: (4)                Parameters
139: (4)                ----------
140: (4)                G : NetworkX graph
141: (4)                source : node, optional
142: (7)                   Specify starting node for depth-first search.
143: (7)                   Note that you will get predecessors for all nodes in the
144: (7)                   component containing `source`. This input only specifies
145: (7)                   where the DFS starts.
146: (4)                depth_limit : int, optional (default=len(G))
147: (7)                   Specify the maximum search depth.
148: (4)                sort_neighbors : function (default=None)
149: (8)                    A function that takes an iterator over nodes as the input, and
150: (8)                    returns an iterable of the same nodes with a custom ordering.
151: (8)                    For example, `sorted` will sort the nodes in increasing order.
152: (4)                Returns
153: (4)                -------
154: (4)                pred: dict
155: (7)                   A dictionary with nodes as keys and predecessor nodes as values.
156: (4)                Examples
157: (4)                --------
158: (4)                >>> G = nx.path_graph(4)
159: (4)                >>> nx.dfs_predecessors(G, source=0)
160: (4)                {1: 0, 2: 1, 3: 2}
161: (4)                >>> nx.dfs_predecessors(G, source=0, depth_limit=2)
162: (4)                {1: 0, 2: 1}
163: (4)                Notes
164: (4)                -----
165: (4)                If a source is not specified then a source is chosen arbitrarily and
166: (4)                repeatedly until all components in the graph are searched.
167: (4)                The implementation of this function is adapted from David Eppstein's
168: (4)                depth-first search function in `PADS`_, with modifications
169: (4)                to allow depth limits based on the Wikipedia article
170: (4)                "`Depth-limited search`_".
171: (4)                .. _PADS: http://www.ics.uci.edu/~eppstein/PADS
```

```
(4)                 .. _Depth-limited search: https://en.wikipedia.org/wiki/Depth-limited_search
(4)                 See Also
(4)                 --------
(4)                 dfs_preorder_nodes
(4)                 dfs_postorder_nodes
(4)                 dfs_labeled_edges
(4)                 :func:`~networkx.algorithms.traversal.edgedfs.edge_dfs`
(4)                 :func:`~networkx.algorithms.traversal.breadth_first_search.bfs_tree`
(4)                 """
(4)                 return {
(8)                     t: s
(8)                     for s, t in dfs_edges(G, source, depth_limit, sort_neighbors=sort_neighbors)
(4)                 }
(0)             @nx._dispatchable
(0)             def dfs_successors(G, source=None, depth_limit=None, *, sort_neighbors=None):
(4)                 """Returns dictionary of successors in depth-first-search from source.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 source : node, optional
(7)                    Specify starting node for depth-first search.
(7)                    Note that you will get successors for all nodes in the
(7)                    component containing `source`. This input only specifies
(7)                    where the DFS starts.
(4)                 depth_limit : int, optional (default=len(G))
(7)                    Specify the maximum search depth.
(4)                 sort_neighbors : function (default=None)
(8)                     A function that takes an iterator over nodes as the input, and
(8)                     returns an iterable of the same nodes with a custom ordering.
(8)                     For example, `sorted` will sort the nodes in increasing order.
(4)                 Returns
(4)                 -------
(4)                 succ: dict
(7)                    A dictionary with nodes as keys and list of successor nodes as values.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> nx.dfs_successors(G, source=0)
(4)                 {0: [1], 1: [2], 2: [3], 3: [4]}
(4)                 >>> nx.dfs_successors(G, source=0, depth_limit=2)
(4)                 {0: [1], 1: [2]}
(4)                 Notes
(4)                 -----
(4)                 If a source is not specified then a source is chosen arbitrarily and
(4)                 repeatedly until all components in the graph are searched.
(4)                 The implementation of this function is adapted from David Eppstein's
(4)                 depth-first search function in `PADS`_, with modifications
(4)                 to allow depth limits based on the Wikipedia article
(4)                 "`Depth-limited search`_".
(4)                 .. _PADS: http://www.ics.uci.edu/~eppstein/PADS
(4)                 .. _Depth-limited search: https://en.wikipedia.org/wiki/Depth-limited_search
(4)                 See Also
(4)                 --------
(4)                 dfs_preorder_nodes
(4)                 dfs_postorder_nodes
(4)                 dfs_labeled_edges
(4)                 :func:`~networkx.algorithms.traversal.edgedfs.edge_dfs`
(4)                 :func:`~networkx.algorithms.traversal.breadth_first_search.bfs_tree`
(4)                 """
(4)                 d = defaultdict(list)
(4)                 for s, t in dfs_edges(
(8)                     G,
(8)                     source=source,
(8)                     depth_limit=depth_limit,
(8)                     sort_neighbors=sort_neighbors,
(4)                 ):
```

```
(8)                 d[s].append(t)
(4)             return dict(d)
(0)         @nx._dispatchable
(0)         def dfs_postorder_nodes(G, source=None, depth_limit=None, *, sort_neighbors=None):
(4)             """Generate nodes in a depth-first-search post-ordering starting at source.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source : node, optional
(7)                Specify starting node for depth-first search.
(4)             depth_limit : int, optional (default=len(G))
(7)                Specify the maximum search depth.
(4)             sort_neighbors : function (default=None)
(8)                 A function that takes an iterator over nodes as the input, and
(8)                 returns an iterable of the same nodes with a custom ordering.
(8)                 For example, `sorted` will sort the nodes in increasing order.
(4)             Returns
(4)             -------
(4)             nodes: generator
(7)                A generator of nodes in a depth-first-search post-ordering.
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> list(nx.dfs_postorder_nodes(G, source=0))
(4)             [4, 3, 2, 1, 0]
(4)             >>> list(nx.dfs_postorder_nodes(G, source=0, depth_limit=2))
(4)             [1, 0]
(4)             Notes
(4)             -----
(4)             If a source is not specified then a source is chosen arbitrarily and
(4)             repeatedly until all components in the graph are searched.
(4)             The implementation of this function is adapted from David Eppstein's
(4)             depth-first search function in `PADS`_, with modifications
(4)             to allow depth limits based on the Wikipedia article
(4)             "`Depth-limited search`_".
(4)             .. _PADS: http://www.ics.uci.edu/~eppstein/PADS
(4)             .. _Depth-limited search: https://en.wikipedia.org/wiki/Depth-limited_search
(4)             See Also
(4)             --------
(4)             dfs_edges
(4)             dfs_preorder_nodes
(4)             dfs_labeled_edges
(4)             :func:`~networkx.algorithms.traversal.edgedfs.edge_dfs`
(4)             :func:`~networkx.algorithms.traversal.breadth_first_search.bfs_tree`
(4)             """
(4)             edges = nx.dfs_labeled_edges(
(8)                 G, source=source, depth_limit=depth_limit, sort_neighbors=sort_neighbors
(4)             )
(4)             return (v for u, v, d in edges if d == "reverse")
(0)         @nx._dispatchable
(0)         def dfs_preorder_nodes(G, source=None, depth_limit=None, *, sort_neighbors=None):
(4)             """Generate nodes in a depth-first-search pre-ordering starting at source.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX graph
(4)             source : node, optional
(7)                Specify starting node for depth-first search and return nodes in
(7)                the component reachable from source.
(4)             depth_limit : int, optional (default=len(G))
(7)                Specify the maximum search depth.
(4)             sort_neighbors : function (default=None)
(8)                 A function that takes an iterator over nodes as the input, and
(8)                 returns an iterable of the same nodes with a custom ordering.
(8)                 For example, `sorted` will sort the nodes in increasing order.
```

```
(4)                Returns
(4)                -------
(4)                nodes: generator
(7)                   A generator of nodes in a depth-first-search pre-ordering.
(4)                Examples
(4)                --------
(4)                >>> G = nx.path_graph(5)
(4)                >>> list(nx.dfs_preorder_nodes(G, source=0))
(4)                [0, 1, 2, 3, 4]
(4)                >>> list(nx.dfs_preorder_nodes(G, source=0, depth_limit=2))
(4)                [0, 1, 2]
(4)                Notes
(4)                -----
(4)                If a source is not specified then a source is chosen arbitrarily and
(4)                repeatedly until all components in the graph are searched.
(4)                The implementation of this function is adapted from David Eppstein's
(4)                depth-first search function in `PADS`_, with modifications
(4)                to allow depth limits based on the Wikipedia article
(4)                "`Depth-limited search`_".
(4)                .. _PADS: http://www.ics.uci.edu/~eppstein/PADS
(4)                .. _Depth-limited search: https://en.wikipedia.org/wiki/Depth-
limited_search
(4)                See Also
(4)                --------
(4)                dfs_edges
(4)                dfs_postorder_nodes
(4)                dfs_labeled_edges
(4)                :func:`~networkx.algorithms.traversal.breadth_first_search.bfs_edges`
(4)                """
(4)                edges = nx.dfs_labeled_edges(
(8)                    G, source=source, depth_limit=depth_limit,
sort_neighbors=sort_neighbors
(4)                )
(4)                return (v for u, v, d in edges if d == "forward")
(0)            @nx._dispatchable
(0)            def dfs_labeled_edges(G, source=None, depth_limit=None, *,
sort_neighbors=None):
(4)                """Iterate over edges in a depth-first-search (DFS) labeled by type.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX graph
(4)                source : node, optional
(7)                   Specify starting node for depth-first search and return edges in
(7)                   the component reachable from source.
(4)                depth_limit : int, optional (default=len(G))
(7)                   Specify the maximum search depth.
(4)                sort_neighbors : function (default=None)
(8)                    A function that takes an iterator over nodes as the input, and
(8)                    returns an iterable of the same nodes with a custom ordering.
(8)                    For example, `sorted` will sort the nodes in increasing order.
(4)                Returns
(4)                -------
(4)                edges: generator
(7)                   A generator of triples of the form (*u*, *v*, *d*), where (*u*,
(7)                   *v*) is the edge being explored in the depth-first search and *d*
(7)                   is one of the strings 'forward', 'nontree', 'reverse', or 'reverse-
depth_limit'.
(7)                   A 'forward' edge is one in which *u* has been visited but *v* has
(7)                   not. A 'nontree' edge is one in which both *u* and *v* have been
(7)                   visited but the edge is not in the DFS tree. A 'reverse' edge is
(7)                   one in which both *u* and *v* have been visited and the edge is in
(7)                   the DFS tree. When the `depth_limit` is reached via a 'forward' edge,
(7)                   a 'reverse' edge is immediately generated rather than the subtree
(7)                   being explored. To indicate this flavor of 'reverse' edge, the string
(7)                   yielded is 'reverse-depth_limit'.
(4)                Examples
(4)                --------
(4)                The labels reveal the complete transcript of the depth-first search
(4)                algorithm in more detail than, for example, :func:`dfs_edges`::
```

```
367: (8)                >>> from pprint import pprint
368: (8)                >>>
369: (8)                >>> G = nx.DiGraph([(0, 1), (1, 2), (2, 1)])
370: (8)                >>> pprint(list(nx.dfs_labeled_edges(G, source=0)))
371: (8)                [(0, 0, 'forward'),
372: (9)                 (0, 1, 'forward'),
373: (9)                 (1, 2, 'forward'),
374: (9)                 (2, 1, 'nontree'),
375: (9)                 (1, 2, 'reverse'),
376: (9)                 (0, 1, 'reverse'),
377: (9)                 (0, 0, 'reverse')]
378: (4)            Notes
379: (4)            -----
380: (4)            If a source is not specified then a source is chosen arbitrarily and
381: (4)            repeatedly until all components in the graph are searched.
382: (4)            The implementation of this function is adapted from David Eppstein's
383: (4)            depth-first search function in `PADS`_, with modifications
384: (4)            to allow depth limits based on the Wikipedia article
385: (4)            "`Depth-limited search`_".
386: (4)            .. _PADS: http://www.ics.uci.edu/~eppstein/PADS
387: (4)            .. _Depth-limited search: https://en.wikipedia.org/wiki/Depth-
limited_search
388: (4)            See Also
389: (4)            --------
390: (4)            dfs_edges
391: (4)            dfs_preorder_nodes
392: (4)            dfs_postorder_nodes
393: (4)            """
394: (4)            if source is None:
395: (8)                nodes = G
396: (4)            else:
397: (8)                nodes = [source]
398: (4)            if depth_limit is None:
399: (8)                depth_limit = len(G)
400: (4)            get_children = (
401: (8)                G.neighbors
402: (8)                if sort_neighbors is None
403: (8)                else lambda n: iter(sort_neighbors(G.neighbors(n)))
404: (4)            )
405: (4)            visited = set()
406: (4)            for start in nodes:
407: (8)                if start in visited:
408: (12)                   continue
409: (8)                yield start, start, "forward"
410: (8)                visited.add(start)
411: (8)                stack = [(start, get_children(start))]
412: (8)                depth_now = 1
413: (8)                while stack:
414: (12)                   parent, children = stack[-1]
415: (12)                   for child in children:
416: (16)                       if child in visited:
417: (20)                           yield parent, child, "nontree"
418: (16)                       else:
419: (20)                           yield parent, child, "forward"
420: (20)                           visited.add(child)
421: (20)                           if depth_now < depth_limit:
422: (24)                               stack.append((child, iter(get_children(child))))
423: (24)                               depth_now += 1
424: (24)                               break
425: (20)                           else:
426: (24)                               yield parent, child, "reverse-depth_limit"
427: (12)                   else:
428: (16)                       stack.pop()
429: (16)                       depth_now -= 1
430: (16)                       if stack:
431: (20)                           yield stack[-1][0], parent, "reverse"
432: (8)                yield start, start, "reverse"

----------------------------------------
```

File 366 - edgebfs.py:

```
1: (0)              """
2: (0)              ============================
3: (0)              Breadth First Search on Edges
4: (0)              ============================
5: (0)              Algorithms for a breadth-first traversal of edges in a graph.
6: (0)              """
7: (0)              from collections import deque
8: (0)              import networkx as nx
9: (0)              FORWARD = "forward"
10: (0)             REVERSE = "reverse"
11: (0)             __all__ = ["edge_bfs"]
12: (0)             @nx._dispatchable
13: (0)             def edge_bfs(G, source=None, orientation=None):
14: (4)                 """A directed, breadth-first-search of edges in `G`, beginning at
`source`.
15: (4)                 Yield the edges of G in a breadth-first-search order continuing until
16: (4)                 all edges are generated.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : graph
20: (8)                     A directed/undirected graph/multigraph.
21: (4)                 source : node, list of nodes
22: (8)                     The node from which the traversal begins. If None, then a source
23: (8)                     is chosen arbitrarily and repeatedly until all edges from each node in
24: (8)                     the graph are searched.
25: (4)                 orientation : None | 'original' | 'reverse' | 'ignore' (default: None)
26: (8)                     For directed graphs and directed multigraphs, edge traversals need not
27: (8)                     respect the original orientation of the edges.
28: (8)                     When set to 'reverse' every edge is traversed in the reverse
direction.
29: (8)                     When set to 'ignore', every edge is treated as undirected.
30: (8)                     When set to 'original', every edge is treated as directed.
31: (8)                     In all three cases, the yielded edge tuples add a last entry to
32: (8)                     indicate the direction in which that edge was traversed.
33: (8)                     If orientation is None, the yielded edge has no direction indicated.
34: (8)                     The direction is respected, but not reported.
35: (4)                 Yields
36: (4)                 ------
37: (4)                 edge : directed edge
38: (8)                     A directed edge indicating the path taken by the breadth-first-search.
39: (8)                     For graphs, `edge` is of the form `(u, v)` where `u` and `v`
40: (8)                     are the tail and head of the edge as determined by the traversal.
41: (8)                     For multigraphs, `edge` is of the form `(u, v, key)`, where `key` is
42: (8)                     the key of the edge. When the graph is directed, then `u` and `v`
43: (8)                     are always in the order of the actual directed edge.
44: (8)                     If orientation is not None then the edge tuple is extended to include
45: (8)                     the direction of traversal ('forward' or 'reverse') on that edge.
46: (4)                 Examples
47: (4)                 --------
48: (4)                 >>> nodes = [0, 1, 2, 3]
49: (4)                 >>> edges = [(0, 1), (1, 0), (1, 0), (2, 0), (2, 1), (3, 1)]
50: (4)                 >>> list(nx.edge_bfs(nx.Graph(edges), nodes))
51: (4)                 [(0, 1), (0, 2), (1, 2), (1, 3)]
52: (4)                 >>> list(nx.edge_bfs(nx.DiGraph(edges), nodes))
53: (4)                 [(0, 1), (1, 0), (2, 0), (2, 1), (3, 1)]
54: (4)                 >>> list(nx.edge_bfs(nx.MultiGraph(edges), nodes))
55: (4)                 [(0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 2, 0), (1, 2, 0), (1, 3, 0)]
56: (4)                 >>> list(nx.edge_bfs(nx.MultiDiGraph(edges), nodes))
57: (4)                 [(0, 1, 0), (1, 0, 0), (1, 0, 1), (2, 0, 0), (2, 1, 0), (3, 1, 0)]
58: (4)                 >>> list(nx.edge_bfs(nx.DiGraph(edges), nodes, orientation="ignore"))
59: (4)                 [(0, 1, 'forward'), (1, 0, 'reverse'), (2, 0, 'reverse'), (2, 1,
'reverse'), (3, 1, 'reverse')]
60: (4)                 >>> list(nx.edge_bfs(nx.MultiDiGraph(edges), nodes, orientation="ignore"))
61: (4)                 [(0, 1, 0, 'forward'), (1, 0, 0, 'reverse'), (1, 0, 1, 'reverse'), (2, 0,
0, 'reverse'), (2, 1, 0, 'reverse'), (3, 1, 0, 'reverse')]
62: (4)                 Notes
```

```
(4)     -----
(4)     The goal of this function is to visit edges. It differs from the more
(4)     familiar breadth-first-search of nodes, as provided by
(4)     :func:`networkx.algorithms.traversal.breadth_first_search.bfs_edges`, in
(4)     that it does not stop once every node has been visited. In a directed
graph
(4)     with edges [(0, 1), (1, 2), (2, 1)], the edge (2, 1) would not be visited
(4)     if not for the functionality provided by this function.
(4)     The naming of this function is very similar to bfs_edges. The difference
(4)     is that 'edge_bfs' yields edges even if they extend back to an already
(4)     explored node while 'bfs_edges' yields the edges of the tree that results
(4)     from a breadth-first-search (BFS) so no edges are reported if they extend
(4)     to already explored nodes. That means 'edge_bfs' reports all edges while
(4)     'bfs_edges' only report those traversed by a node-based BFS. Yet another
(4)     description is that 'bfs_edges' reports the edges traversed during BFS
(4)     while 'edge_bfs' reports all edges in the order they are explored.
(4)     See Also
(4)     --------
(4)     bfs_edges
(4)     bfs_tree
(4)     edge_dfs
(4)     """
(4)     nodes = list(G.nbunch_iter(source))
(4)     if not nodes:
(8)         return
(4)     directed = G.is_directed()
(4)     kwds = {"data": False}
(4)     if G.is_multigraph() is True:
(8)         kwds["keys"] = True
(4)     if orientation is None:
(8)         def edges_from(node):
(12)            return iter(G.edges(node, **kwds))
(4)     elif not directed or orientation == "original":
(8)         def edges_from(node):
(12)            for e in G.edges(node, **kwds):
(16)                yield e + (FORWARD,)
(4)     elif orientation == "reverse":
(8)         def edges_from(node):
(12)            for e in G.in_edges(node, **kwds):
(16)                yield e + (REVERSE,)
(4)     elif orientation == "ignore":
(8)         def edges_from(node):
(12)            for e in G.edges(node, **kwds):
(16)                yield e + (FORWARD,)
(12)            for e in G.in_edges(node, **kwds):
(16)                yield e + (REVERSE,)
(4)     else:
(8)         raise nx.NetworkXError("invalid orientation argument.")
(4)     if directed:
(8)         neighbors = G.successors
(8)         def edge_id(edge):
(12)            return edge[:-1] if orientation is not None else edge
(4)     else:
(8)         neighbors = G.neighbors
(8)         def edge_id(edge):
(12)            return (frozenset(edge[:2]),) + edge[2:]
(4)     check_reverse = directed and orientation in ("reverse", "ignore")
(4)     visited_nodes = set(nodes)
(4)     visited_edges = set()
(4)     queue = deque([(n, edges_from(n)) for n in nodes])
(4)     while queue:
(8)         parent, children_edges = queue.popleft()
(8)         for edge in children_edges:
(12)            if check_reverse and edge[-1] == REVERSE:
(16)                child = edge[0]
(12)            else:
(16)                child = edge[1]
(12)            if child not in visited_nodes:
(16)                visited_nodes.add(child)
```

```
131: (16)                           queue.append((child, edges_from(child)))
132: (12)                       edgeid = edge_id(edge)
133: (12)                       if edgeid not in visited_edges:
134: (16)                           visited_edges.add(edgeid)
135: (16)                           yield edge
```

File 367 - edgedfs.py:

```
1: (0)              """
2: (0)              ==========================
3: (0)              Depth First Search on Edges
4: (0)              ==========================
5: (0)              Algorithms for a depth-first traversal of edges in a graph.
6: (0)              """
7: (0)              import networkx as nx
8: (0)              FORWARD = "forward"
9: (0)              REVERSE = "reverse"
10: (0)             __all__ = ["edge_dfs"]
11: (0)             @nx._dispatchable
12: (0)             def edge_dfs(G, source=None, orientation=None):
13: (4)                 """A directed, depth-first-search of edges in `G`, beginning at `source`.
14: (4)                 Yield the edges of G in a depth-first-search order continuing until
15: (4)                 all edges are generated.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 G : graph
19: (8)                     A directed/undirected graph/multigraph.
20: (4)                 source : node, list of nodes
21: (8)                     The node from which the traversal begins. If None, then a source
22: (8)                     is chosen arbitrarily and repeatedly until all edges from each node in
23: (8)                     the graph are searched.
24: (4)                 orientation : None | 'original' | 'reverse' | 'ignore' (default: None)
25: (8)                     For directed graphs and directed multigraphs, edge traversals need not
26: (8)                     respect the original orientation of the edges.
27: (8)                     When set to 'reverse' every edge is traversed in the reverse
direction.
28: (8)                     When set to 'ignore', every edge is treated as undirected.
29: (8)                     When set to 'original', every edge is treated as directed.
30: (8)                     In all three cases, the yielded edge tuples add a last entry to
31: (8)                     indicate the direction in which that edge was traversed.
32: (8)                     If orientation is None, the yielded edge has no direction indicated.
33: (8)                     The direction is respected, but not reported.
34: (4)                 Yields
35: (4)                 ------
36: (4)                 edge : directed edge
37: (8)                     A directed edge indicating the path taken by the depth-first
traversal.
38: (8)                     For graphs, `edge` is of the form `(u, v)` where `u` and `v`
39: (8)                     are the tail and head of the edge as determined by the traversal.
40: (8)                     For multigraphs, `edge` is of the form `(u, v, key)`, where `key` is
41: (8)                     the key of the edge. When the graph is directed, then `u` and `v`
42: (8)                     are always in the order of the actual directed edge.
43: (8)                     If orientation is not None then the edge tuple is extended to include
44: (8)                     the direction of traversal ('forward' or 'reverse') on that edge.
45: (4)                 Examples
46: (4)                 --------
47: (4)                 >>> nodes = [0, 1, 2, 3]
48: (4)                 >>> edges = [(0, 1), (1, 0), (1, 0), (2, 1), (3, 1)]
49: (4)                 >>> list(nx.edge_dfs(nx.Graph(edges), nodes))
50: (4)                 [(0, 1), (1, 2), (1, 3)]
51: (4)                 >>> list(nx.edge_dfs(nx.DiGraph(edges), nodes))
52: (4)                 [(0, 1), (1, 0), (2, 1), (3, 1)]
53: (4)                 >>> list(nx.edge_dfs(nx.MultiGraph(edges), nodes))
54: (4)                 [(0, 1, 0), (1, 0, 1), (0, 1, 2), (1, 2, 0), (1, 3, 0)]
55: (4)                 >>> list(nx.edge_dfs(nx.MultiDiGraph(edges), nodes))
56: (4)                 [(0, 1, 0), (1, 0, 0), (1, 0, 1), (2, 1, 0), (3, 1, 0)]
57: (4)                 >>> list(nx.edge_dfs(nx.DiGraph(edges), nodes, orientation="ignore"))
```

```
58: (4)                 [(0, 1, 'forward'), (1, 0, 'forward'), (2, 1, 'reverse'), (3, 1,
'reverse')]
59: (4)                 >>> list(nx.edge_dfs(nx.MultiDiGraph(edges), nodes, orientation="ignore"))
60: (4)                 [(0, 1, 0, 'forward'), (1, 0, 0, 'forward'), (1, 0, 1, 'reverse'), (2, 1,
0, 'reverse'), (3, 1, 0, 'reverse')]
61: (4)                 Notes
62: (4)                 -----
63: (4)                 The goal of this function is to visit edges. It differs from the more
64: (4)                 familiar depth-first traversal of nodes, as provided by
65: (4)                 :func:`~networkx.algorithms.traversal.depth_first_search.dfs_edges`, in
66: (4)                 that it does not stop once every node has been visited. In a directed
graph
67: (4)                 with edges [(0, 1), (1, 2), (2, 1)], the edge (2, 1) would not be visited
68: (4)                 if not for the functionality provided by this function.
69: (4)                 See Also
70: (4)                 --------
71: (4)                 :func:`~networkx.algorithms.traversal.depth_first_search.dfs_edges`
72: (4)                 """
73: (4)                 nodes = list(G.nbunch_iter(source))
74: (4)                 if not nodes:
75: (8)                     return
76: (4)                 directed = G.is_directed()
77: (4)                 kwds = {"data": False}
78: (4)                 if G.is_multigraph() is True:
79: (8)                     kwds["keys"] = True
80: (4)                 if orientation is None:
81: (8)                     def edges_from(node):
82: (12)                        return iter(G.edges(node, **kwds))
83: (4)                 elif not directed or orientation == "original":
84: (8)                     def edges_from(node):
85: (12)                        for e in G.edges(node, **kwds):
86: (16)                            yield e + (FORWARD,)
87: (4)                 elif orientation == "reverse":
88: (8)                     def edges_from(node):
89: (12)                        for e in G.in_edges(node, **kwds):
90: (16)                            yield e + (REVERSE,)
91: (4)                 elif orientation == "ignore":
92: (8)                     def edges_from(node):
93: (12)                        for e in G.edges(node, **kwds):
94: (16)                            yield e + (FORWARD,)
95: (12)                        for e in G.in_edges(node, **kwds):
96: (16)                            yield e + (REVERSE,)
97: (4)                 else:
98: (8)                     raise nx.NetworkXError("invalid orientation argument.")
99: (4)                 if directed:
100: (8)                    def edge_id(edge):
101: (12)                       return edge[:-1] if orientation is not None else edge
102: (4)                else:
103: (8)                    def edge_id(edge):
104: (12)                       return (frozenset(edge[:2]),) + edge[2:]
105: (4)                check_reverse = directed and orientation in ("reverse", "ignore")
106: (4)                visited_edges = set()
107: (4)                visited_nodes = set()
108: (4)                edges = {}
109: (4)                for start_node in nodes:
110: (8)                    stack = [start_node]
111: (8)                    while stack:
112: (12)                       current_node = stack[-1]
113: (12)                       if current_node not in visited_nodes:
114: (16)                           edges[current_node] = edges_from(current_node)
115: (16)                           visited_nodes.add(current_node)
116: (12)                       try:
117: (16)                           edge = next(edges[current_node])
118: (12)                       except StopIteration:
119: (16)                           stack.pop()
120: (12)                       else:
121: (16)                           edgeid = edge_id(edge)
122: (16)                           if edgeid not in visited_edges:
123: (20)                               visited_edges.add(edgeid)
```

```
124: (20)                                if check_reverse and edge[-1] == REVERSE:
125: (24)                                    stack.append(edge[0])
126: (20)                                else:
127: (24)                                    stack.append(edge[1])
128: (20)                                yield edge
```

File 368 - __init__.py:

```
1: (0)
```

File 369 - test_beamsearch.py:

```
1: (0)              """Unit tests for the beam search functions."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              def test_narrow():
5: (4)                  """Tests that a narrow beam width may cause an incomplete search."""
6: (4)                  G = nx.cycle_graph(4)
7: (4)                  edges = nx.bfs_beam_edges(G, source=0, value=lambda n: n, width=1)
8: (4)                  assert list(edges) == [(0, 3), (3, 2)]
9: (0)              @pytest.mark.parametrize("width", (2, None))
10: (0)             def test_wide(width):
11: (4)                 """All nodes are searched when `width` is None or >= max degree"""
12: (4)                 G = nx.cycle_graph(4)
13: (4)                 edges = nx.bfs_beam_edges(G, source=0, value=lambda n: n, width=width)
14: (4)                 assert list(edges) == [(0, 3), (0, 1), (3, 2)]
```

File 370 - test_bfs.py:

```
1: (0)              from functools import partial
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestBFS:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      G = nx.Graph()
8: (8)                      G.add_edges_from([(0, 1), (1, 2), (1, 3), (2, 4), (3, 4)])
9: (8)                      cls.G = G
10: (4)                 def test_successor(self):
11: (8)                     assert dict(nx.bfs_successors(self.G, source=0)) == {0: [1], 1: [2,
3], 2: [4]}
12: (4)                 def test_predecessor(self):
13: (8)                     assert dict(nx.bfs_predecessors(self.G, source=0)) == {1: 0, 2: 1, 3:
1, 4: 2}
14: (4)                 def test_bfs_tree(self):
15: (8)                     T = nx.bfs_tree(self.G, source=0)
16: (8)                     assert sorted(T.nodes()) == sorted(self.G.nodes())
17: (8)                     assert sorted(T.edges()) == [(0, 1), (1, 2), (1, 3), (2, 4)]
18: (4)                 def test_bfs_edges(self):
19: (8)                     edges = nx.bfs_edges(self.G, source=0)
20: (8)                     assert list(edges) == [(0, 1), (1, 2), (1, 3), (2, 4)]
21: (4)                 def test_bfs_edges_reverse(self):
22: (8)                     D = nx.DiGraph()
23: (8)                     D.add_edges_from([(0, 1), (1, 2), (1, 3), (2, 4), (3, 4)])
24: (8)                     edges = nx.bfs_edges(D, source=4, reverse=True)
25: (8)                     assert list(edges) == [(4, 2), (4, 3), (2, 1), (1, 0)]
26: (4)                 def test_bfs_edges_sorting(self):
27: (8)                     D = nx.DiGraph()
28: (8)                     D.add_edges_from([(0, 1), (0, 2), (1, 4), (1, 3), (2, 5)])
29: (8)                     sort_desc = partial(sorted, reverse=True)
30: (8)                     edges_asc = nx.bfs_edges(D, source=0, sort_neighbors=sorted)
31: (8)                     edges_desc = nx.bfs_edges(D, source=0, sort_neighbors=sort_desc)
32: (8)                     assert list(edges_asc) == [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5)]
```

```
 33: (8)                     assert list(edges_desc) == [(0, 2), (0, 1), (2, 5), (1, 4), (1, 3)]
 34: (4)                 def test_bfs_tree_isolates(self):
 35: (8)                     G = nx.Graph()
 36: (8)                     G.add_node(1)
 37: (8)                     G.add_node(2)
 38: (8)                     T = nx.bfs_tree(G, source=1)
 39: (8)                     assert sorted(T.nodes()) == [1]
 40: (8)                     assert sorted(T.edges()) == []
 41: (4)                 def test_bfs_layers(self):
 42: (8)                     expected = {
 43: (12)                        0: [0],
 44: (12)                        1: [1],
 45: (12)                        2: [2, 3],
 46: (12)                        3: [4],
 47: (8)                     }
 48: (8)                     assert dict(enumerate(nx.bfs_layers(self.G, sources=[0]))) == expected
 49: (8)                     assert dict(enumerate(nx.bfs_layers(self.G, sources=0))) == expected
 50: (4)                 def test_bfs_layers_missing_source(self):
 51: (8)                     with pytest.raises(nx.NetworkXError):
 52: (12)                        next(nx.bfs_layers(self.G, sources="abc"))
 53: (8)                     with pytest.raises(nx.NetworkXError):
 54: (12)                        next(nx.bfs_layers(self.G, sources=["abc"]))
 55: (4)                 def test_descendants_at_distance(self):
 56: (8)                     for distance, descendants in enumerate([{0}, {1}, {2, 3}, {4}]):
 57: (12)                        assert nx.descendants_at_distance(self.G, 0, distance) ==
 descendants
 58: (4)                 def test_descendants_at_distance_missing_source(self):
 59: (8)                     with pytest.raises(nx.NetworkXError):
 60: (12)                        nx.descendants_at_distance(self.G, "abc", 0)
 61: (4)                 def test_bfs_labeled_edges_directed(self):
 62: (8)                     D = nx.cycle_graph(5, create_using=nx.DiGraph)
 63: (8)                     expected = [
 64: (12)                        (0, 1, "tree"),
 65: (12)                        (1, 2, "tree"),
 66: (12)                        (2, 3, "tree"),
 67: (12)                        (3, 4, "tree"),
 68: (12)                        (4, 0, "reverse"),
 69: (8)                     ]
 70: (8)                     answer = list(nx.bfs_labeled_edges(D, 0))
 71: (8)                     assert expected == answer
 72: (8)                     D.add_edge(4, 4)
 73: (8)                     expected.append((4, 4, "level"))
 74: (8)                     answer = list(nx.bfs_labeled_edges(D, 0))
 75: (8)                     assert expected == answer
 76: (8)                     D.add_edge(0, 2)
 77: (8)                     D.add_edge(1, 5)
 78: (8)                     D.add_edge(2, 5)
 79: (8)                     D.remove_edge(4, 4)
 80: (8)                     expected = [
 81: (12)                        (0, 1, "tree"),
 82: (12)                        (0, 2, "tree"),
 83: (12)                        (1, 2, "level"),
 84: (12)                        (1, 5, "tree"),
 85: (12)                        (2, 3, "tree"),
 86: (12)                        (2, 5, "forward"),
 87: (12)                        (3, 4, "tree"),
 88: (12)                        (4, 0, "reverse"),
 89: (8)                     ]
 90: (8)                     answer = list(nx.bfs_labeled_edges(D, 0))
 91: (8)                     assert expected == answer
 92: (8)                     G = D.to_undirected()
 93: (8)                     G.add_edge(4, 4)
 94: (8)                     expected = [
 95: (12)                        (0, 1, "tree"),
 96: (12)                        (0, 2, "tree"),
 97: (12)                        (0, 4, "tree"),
 98: (12)                        (1, 2, "level"),
 99: (12)                        (1, 5, "tree"),
 100: (12)                       (2, 3, "tree"),
```

```
101: (12)                        (2, 5, "forward"),
102: (12)                        (4, 3, "forward"),
103: (12)                        (4, 4, "level"),
104: (8)                     ]
105: (8)                     answer = list(nx.bfs_labeled_edges(G, 0))
106: (8)                     assert expected == answer
107: (0)             class TestBreadthLimitedSearch:
108: (4)                 @classmethod
109: (4)                 def setup_class(cls):
110: (8)                     G = nx.Graph()
111: (8)                     nx.add_path(G, [0, 1, 2, 3, 4, 5, 6])
112: (8)                     nx.add_path(G, [2, 7, 8, 9, 10])
113: (8)                     cls.G = G
114: (8)                     D = nx.Graph()
115: (8)                     D.add_edges_from([(0, 1), (2, 3)])
116: (8)                     nx.add_path(D, [2, 7, 8, 9, 10])
117: (8)                     cls.D = D
118: (4)                 def test_limited_bfs_successor(self):
119: (8)                     assert dict(nx.bfs_successors(self.G, source=1, depth_limit=3)) == {
120: (12)                        1: [0, 2],
121: (12)                        2: [3, 7],
122: (12)                        3: [4],
123: (12)                        7: [8],
124: (8)                     }
125: (8)                     result = {
126: (12)                        n: sorted(s) for n, s in nx.bfs_successors(self.D, source=7,
depth_limit=2)
127: (8)                     }
128: (8)                     assert result == {8: [9], 2: [3], 7: [2, 8]}
129: (4)                 def test_limited_bfs_predecessor(self):
130: (8)                     assert dict(nx.bfs_predecessors(self.G, source=1, depth_limit=3)) == {
131: (12)                        0: 1,
132: (12)                        2: 1,
133: (12)                        3: 2,
134: (12)                        4: 3,
135: (12)                        7: 2,
136: (12)                        8: 7,
137: (8)                     }
138: (8)                     assert dict(nx.bfs_predecessors(self.D, source=7, depth_limit=2)) == {
139: (12)                        2: 7,
140: (12)                        3: 2,
141: (12)                        8: 7,
142: (12)                        9: 8,
143: (8)                     }
144: (4)                 def test_limited_bfs_tree(self):
145: (8)                     T = nx.bfs_tree(self.G, source=3, depth_limit=1)
146: (8)                     assert sorted(T.edges()) == [(3, 2), (3, 4)]
147: (4)                 def test_limited_bfs_edges(self):
148: (8)                     edges = nx.bfs_edges(self.G, source=9, depth_limit=4)
149: (8)                     assert list(edges) == [(9, 8), (9, 10), (8, 7), (7, 2), (2, 1), (2,
3)]
150: (4)                 def test_limited_bfs_layers(self):
151: (8)                     assert dict(enumerate(nx.bfs_layers(self.G, sources=[0]))) == {
152: (12)                        0: [0],
153: (12)                        1: [1],
154: (12)                        2: [2],
155: (12)                        3: [3, 7],
156: (12)                        4: [4, 8],
157: (12)                        5: [5, 9],
158: (12)                        6: [6, 10],
159: (8)                     }
160: (8)                     assert dict(enumerate(nx.bfs_layers(self.D, sources=2))) == {
161: (12)                        0: [2],
162: (12)                        1: [3, 7],
163: (12)                        2: [8],
164: (12)                        3: [9],
165: (12)                        4: [10],
166: (8)                     }
167: (4)                 def test_limited_descendants_at_distance(self):
```

```
168: (8)                    for distance, descendants in enumerate(
169: (12)                       [{0}, {1}, {2}, {3, 7}, {4, 8}, {5, 9}, {6, 10}]
170: (8)                    ):
171: (12)                       assert nx.descendants_at_distance(self.G, 0, distance) ==
descendants
172: (8)                    for distance, descendants in enumerate([{2}, {3, 7}, {8}, {9}, {10}]):
173: (12)                       assert nx.descendants_at_distance(self.D, 2, distance) ==
descendants
174: (0)            def test_deprecations():
175: (4)                G = nx.Graph([(1, 2)])
176: (4)                generic_bfs = nx.breadth_first_search.generic_bfs_edges
177: (4)                with pytest.deprecated_call():
178: (8)                    list(generic_bfs(G, source=1, sort_neighbors=sorted))
179: (4)                with pytest.deprecated_call():
180: (8)                    list(generic_bfs(G, source=1, neighbors=G.neighbors,
sort_neighbors=sorted))
```

File 371 - test_dfs.py:

```
1: (0)              import networkx as nx
2: (0)              class TestDFS:
3: (4)                  @classmethod
4: (4)                  def setup_class(cls):
5: (8)                      G = nx.Graph()
6: (8)                      G.add_edges_from([(0, 1), (1, 2), (1, 3), (2, 4), (3, 0), (0, 4)])
7: (8)                      cls.G = G
8: (8)                      D = nx.Graph()
9: (8)                      D.add_edges_from([(0, 1), (2, 3)])
10: (8)                     cls.D = D
11: (4)                 def test_preorder_nodes(self):
12: (8)                     assert list(nx.dfs_preorder_nodes(self.G, source=0)) == [0, 1, 2, 4,
3]
13: (8)                     assert list(nx.dfs_preorder_nodes(self.D)) == [0, 1, 2, 3]
14: (8)                     assert list(nx.dfs_preorder_nodes(self.D, source=2)) == [2, 3]
15: (4)                 def test_postorder_nodes(self):
16: (8)                     assert list(nx.dfs_postorder_nodes(self.G, source=0)) == [4, 2, 3, 1,
0]
17: (8)                     assert list(nx.dfs_postorder_nodes(self.D)) == [1, 0, 3, 2]
18: (8)                     assert list(nx.dfs_postorder_nodes(self.D, source=0)) == [1, 0]
19: (4)                 def test_successor(self):
20: (8)                     assert nx.dfs_successors(self.G, source=0) == {0: [1], 1: [2, 3], 2:
[4]}
21: (8)                     assert nx.dfs_successors(self.G, source=1) == {0: [3, 4], 1: [0], 4:
[2]}
22: (8)                     assert nx.dfs_successors(self.D) == {0: [1], 2: [3]}
23: (8)                     assert nx.dfs_successors(self.D, source=1) == {1: [0]}
24: (4)                 def test_predecessor(self):
25: (8)                     assert nx.dfs_predecessors(self.G, source=0) == {1: 0, 2: 1, 3: 1, 4:
2}
26: (8)                     assert nx.dfs_predecessors(self.D) == {1: 0, 3: 2}
27: (4)                 def test_dfs_tree(self):
28: (8)                     exp_nodes = sorted(self.G.nodes())
29: (8)                     exp_edges = [(0, 1), (1, 2), (1, 3), (2, 4)]
30: (8)                     T = nx.dfs_tree(self.G, source=0)
31: (8)                     assert sorted(T.nodes()) == exp_nodes
32: (8)                     assert sorted(T.edges()) == exp_edges
33: (8)                     T = nx.dfs_tree(self.G, source=None)
34: (8)                     assert sorted(T.nodes()) == exp_nodes
35: (8)                     assert sorted(T.edges()) == exp_edges
36: (8)                     T = nx.dfs_tree(self.G)
37: (8)                     assert sorted(T.nodes()) == exp_nodes
38: (8)                     assert sorted(T.edges()) == exp_edges
39: (4)                 def test_dfs_edges(self):
40: (8)                     edges = nx.dfs_edges(self.G, source=0)
41: (8)                     assert list(edges) == [(0, 1), (1, 2), (2, 4), (1, 3)]
42: (8)                     edges = nx.dfs_edges(self.D)
43: (8)                     assert list(edges) == [(0, 1), (2, 3)]
```

```
    def test_dfs_edges_sorting(self):
        G = nx.Graph([(0, 1), (1, 2), (1, 3), (2, 4), (3, 0), (0, 4)])
        edges_asc = nx.dfs_edges(G, source=0, sort_neighbors=sorted)
        sorted_desc = lambda x: sorted(x, reverse=True)
        edges_desc = nx.dfs_edges(G, source=0, sort_neighbors=sorted_desc)
        assert list(edges_asc) == [(0, 1), (1, 2), (2, 4), (1, 3)]
        assert list(edges_desc) == [(0, 4), (4, 2), (2, 1), (1, 3)]
    def test_dfs_labeled_edges(self):
        edges = list(nx.dfs_labeled_edges(self.G, source=0))
        forward = [(u, v) for (u, v, d) in edges if d == "forward"]
        assert forward == [(0, 0), (0, 1), (1, 2), (2, 4), (1, 3)]
        assert edges == [
            (0, 0, "forward"),
            (0, 1, "forward"),
            (1, 0, "nontree"),
            (1, 2, "forward"),
            (2, 1, "nontree"),
            (2, 4, "forward"),
            (4, 2, "nontree"),
            (4, 0, "nontree"),
            (2, 4, "reverse"),
            (1, 2, "reverse"),
            (1, 3, "forward"),
            (3, 1, "nontree"),
            (3, 0, "nontree"),
            (1, 3, "reverse"),
            (0, 1, "reverse"),
            (0, 3, "nontree"),
            (0, 4, "nontree"),
            (0, 0, "reverse"),
        ]
    def test_dfs_labeled_edges_sorting(self):
        G = nx.Graph([(0, 1), (1, 2), (1, 3), (2, 4), (3, 0), (0, 4)])
        edges_asc = nx.dfs_labeled_edges(G, source=0, sort_neighbors=sorted)
        sorted_desc = lambda x: sorted(x, reverse=True)
        edges_desc = nx.dfs_labeled_edges(G, source=0, sort_neighbors=sorted_desc)
        assert list(edges_asc) == [
            (0, 0, "forward"),
            (0, 1, "forward"),
            (1, 0, "nontree"),
            (1, 2, "forward"),
            (2, 1, "nontree"),
            (2, 4, "forward"),
            (4, 0, "nontree"),
            (4, 2, "nontree"),
            (2, 4, "reverse"),
            (1, 2, "reverse"),
            (1, 3, "forward"),
            (3, 0, "nontree"),
            (3, 1, "nontree"),
            (1, 3, "reverse"),
            (0, 1, "reverse"),
            (0, 3, "nontree"),
            (0, 4, "nontree"),
            (0, 0, "reverse"),
        ]
        assert list(edges_desc) == [
            (0, 0, "forward"),
            (0, 4, "forward"),
            (4, 2, "forward"),
            (2, 4, "nontree"),
            (2, 1, "forward"),
            (1, 3, "forward"),
            (3, 1, "nontree"),
            (3, 0, "nontree"),
            (1, 3, "reverse"),
            (1, 2, "nontree"),
            (1, 0, "nontree"),
```

```
112: (12)                        (2, 1, "reverse"),
113: (12)                        (4, 2, "reverse"),
114: (12)                        (4, 0, "nontree"),
115: (12)                        (0, 4, "reverse"),
116: (12)                        (0, 3, "nontree"),
117: (12)                        (0, 1, "nontree"),
118: (12)                        (0, 0, "reverse"),
119: (8)                    ]
120: (4)                def test_dfs_labeled_disconnected_edges(self):
121: (8)                    edges = list(nx.dfs_labeled_edges(self.D))
122: (8)                    forward = [(u, v) for (u, v, d) in edges if d == "forward"]
123: (8)                    assert forward == [(0, 0), (0, 1), (2, 2), (2, 3)]
124: (8)                    assert edges == [
125: (12)                        (0, 0, "forward"),
126: (12)                        (0, 1, "forward"),
127: (12)                        (1, 0, "nontree"),
128: (12)                        (0, 1, "reverse"),
129: (12)                        (0, 0, "reverse"),
130: (12)                        (2, 2, "forward"),
131: (12)                        (2, 3, "forward"),
132: (12)                        (3, 2, "nontree"),
133: (12)                        (2, 3, "reverse"),
134: (12)                        (2, 2, "reverse"),
135: (8)                    ]
136: (4)                def test_dfs_tree_isolates(self):
137: (8)                    G = nx.Graph()
138: (8)                    G.add_node(1)
139: (8)                    G.add_node(2)
140: (8)                    T = nx.dfs_tree(G, source=1)
141: (8)                    assert sorted(T.nodes()) == [1]
142: (8)                    assert sorted(T.edges()) == []
143: (8)                    T = nx.dfs_tree(G, source=None)
144: (8)                    assert sorted(T.nodes()) == [1, 2]
145: (8)                    assert sorted(T.edges()) == []
146: (0)            class TestDepthLimitedSearch:
147: (4)                @classmethod
148: (4)                def setup_class(cls):
149: (8)                    G = nx.Graph()
150: (8)                    nx.add_path(G, [0, 1, 2, 3, 4, 5, 6])
151: (8)                    nx.add_path(G, [2, 7, 8, 9, 10])
152: (8)                    cls.G = G
153: (8)                    D = nx.Graph()
154: (8)                    D.add_edges_from([(0, 1), (2, 3)])
155: (8)                    nx.add_path(D, [2, 7, 8, 9, 10])
156: (8)                    cls.D = D
157: (4)                def test_dls_preorder_nodes(self):
158: (8)                    assert list(nx.dfs_preorder_nodes(self.G, source=0, depth_limit=2)) ==
[0, 1, 2]
159: (8)                    assert list(nx.dfs_preorder_nodes(self.D, source=1, depth_limit=2)) ==
([1, 0])
160: (4)                def test_dls_postorder_nodes(self):
161: (8)                    assert list(nx.dfs_postorder_nodes(self.G, source=3, depth_limit=3))
== [
162: (12)                        1,
163: (12)                        7,
164: (12)                        2,
165: (12)                        5,
166: (12)                        4,
167: (12)                        3,
168: (8)                    ]
169: (8)                    assert list(nx.dfs_postorder_nodes(self.D, source=2, depth_limit=2))
== (
170: (12)                        [3, 7, 2]
171: (8)                    )
172: (4)                def test_dls_successor(self):
173: (8)                    result = nx.dfs_successors(self.G, source=4, depth_limit=3)
174: (8)                    assert {n: set(v) for n, v in result.items()} == {
175: (12)                        2: {1, 7},
176: (12)                        3: {2},
```

```
(12)                        4: {3, 5},
(12)                        5: {6},
(8)                     }
(8)                     result = nx.dfs_successors(self.D, source=7, depth_limit=2)
(8)                     assert {n: set(v) for n, v in result.items()} == {8: {9}, 2: {3}, 7:
{8, 2}}
(4)                 def test_dls_predecessor(self):
(8)                     assert nx.dfs_predecessors(self.G, source=0, depth_limit=3) == {
(12)                        1: 0,
(12)                        2: 1,
(12)                        3: 2,
(12)                        7: 2,
(8)                     }
(8)                     assert nx.dfs_predecessors(self.D, source=2, depth_limit=3) == {
(12)                        8: 7,
(12)                        9: 8,
(12)                        3: 2,
(12)                        7: 2,
(8)                     }
(4)                 def test_dls_tree(self):
(8)                     T = nx.dfs_tree(self.G, source=3, depth_limit=1)
(8)                     assert sorted(T.edges()) == [(3, 2), (3, 4)]
(4)                 def test_dls_edges(self):
(8)                     edges = nx.dfs_edges(self.G, source=9, depth_limit=4)
(8)                     assert list(edges) == [(9, 8), (8, 7), (7, 2), (2, 1), (2, 3), (9,
10)]
(4)                 def test_dls_labeled_edges_depth_1(self):
(8)                     edges = list(nx.dfs_labeled_edges(self.G, source=5, depth_limit=1))
(8)                     forward = [(u, v) for (u, v, d) in edges if d == "forward"]
(8)                     assert forward == [(5, 5), (5, 4), (5, 6)]
(8)                     assert edges == [
(12)                        (5, 5, "forward"),
(12)                        (5, 4, "forward"),
(12)                        (5, 4, "reverse-depth_limit"),
(12)                        (5, 6, "forward"),
(12)                        (5, 6, "reverse-depth_limit"),
(12)                        (5, 5, "reverse"),
(8)                     ]
(4)                 def test_dls_labeled_edges_depth_2(self):
(8)                     edges = list(nx.dfs_labeled_edges(self.G, source=6, depth_limit=2))
(8)                     forward = [(u, v) for (u, v, d) in edges if d == "forward"]
(8)                     assert forward == [(6, 6), (6, 5), (5, 4)]
(8)                     assert edges == [
(12)                        (6, 6, "forward"),
(12)                        (6, 5, "forward"),
(12)                        (5, 4, "forward"),
(12)                        (5, 4, "reverse-depth_limit"),
(12)                        (5, 6, "nontree"),
(12)                        (6, 5, "reverse"),
(12)                        (6, 6, "reverse"),
(8)                     ]
(4)                 def test_dls_labeled_disconnected_edges(self):
(8)                     edges = list(nx.dfs_labeled_edges(self.D, depth_limit=1))
(8)                     assert edges == [
(12)                        (0, 0, "forward"),
(12)                        (0, 1, "forward"),
(12)                        (0, 1, "reverse-depth_limit"),
(12)                        (0, 0, "reverse"),
(12)                        (2, 2, "forward"),
(12)                        (2, 3, "forward"),
(12)                        (2, 3, "reverse-depth_limit"),
(12)                        (2, 7, "forward"),
(12)                        (2, 7, "reverse-depth_limit"),
(12)                        (2, 2, "reverse"),
(12)                        (8, 8, "forward"),
(12)                        (8, 7, "nontree"),
(12)                        (8, 9, "forward"),
(12)                        (8, 9, "reverse-depth_limit"),
(12)                        (8, 8, "reverse"),
```

```
244: (12)                        (10, 10, "forward"),
245: (12)                        (10, 9, "nontree"),
246: (12)                        (10, 10, "reverse"),
247: (8)                    ]
248: (8)                    edges = list(nx.dfs_labeled_edges(self.D, depth_limit=19))
249: (8)                    assert edges == [
250: (12)                        (0, 0, "forward"),
251: (12)                        (0, 1, "forward"),
252: (12)                        (1, 0, "nontree"),
253: (12)                        (0, 1, "reverse"),
254: (12)                        (0, 0, "reverse"),
255: (12)                        (2, 2, "forward"),
256: (12)                        (2, 3, "forward"),
257: (12)                        (3, 2, "nontree"),
258: (12)                        (2, 3, "reverse"),
259: (12)                        (2, 7, "forward"),
260: (12)                        (7, 2, "nontree"),
261: (12)                        (7, 8, "forward"),
262: (12)                        (8, 7, "nontree"),
263: (12)                        (8, 9, "forward"),
264: (12)                        (9, 8, "nontree"),
265: (12)                        (9, 10, "forward"),
266: (12)                        (10, 9, "nontree"),
267: (12)                        (9, 10, "reverse"),
268: (12)                        (8, 9, "reverse"),
269: (12)                        (7, 8, "reverse"),
270: (12)                        (2, 7, "reverse"),
271: (12)                        (2, 2, "reverse"),
272: (8)                    ]

----------------------------------------

File 372 - test_edgebfs.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.traversal.edgedfs import FORWARD, REVERSE
4: (0)              class TestEdgeBFS:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      cls.nodes = [0, 1, 2, 3]
8: (8)                      cls.edges = [(0, 1), (1, 0), (1, 0), (2, 0), (2, 1), (3, 1)]
9: (4)                  def test_empty(self):
10: (8)                     G = nx.Graph()
11: (8)                     edges = list(nx.edge_bfs(G))
12: (8)                     assert edges == []
13: (4)                 def test_graph_single_source(self):
14: (8)                     G = nx.Graph(self.edges)
15: (8)                     G.add_edge(4, 5)
16: (8)                     x = list(nx.edge_bfs(G, [0]))
17: (8)                     x_ = [(0, 1), (0, 2), (1, 2), (1, 3)]
18: (8)                     assert x == x_
19: (4)                 def test_graph(self):
20: (8)                     G = nx.Graph(self.edges)
21: (8)                     x = list(nx.edge_bfs(G, self.nodes))
22: (8)                     x_ = [(0, 1), (0, 2), (1, 2), (1, 3)]
23: (8)                     assert x == x_
24: (4)                 def test_digraph(self):
25: (8)                     G = nx.DiGraph(self.edges)
26: (8)                     x = list(nx.edge_bfs(G, self.nodes))
27: (8)                     x_ = [(0, 1), (1, 0), (2, 0), (2, 1), (3, 1)]
28: (8)                     assert x == x_
29: (4)                 def test_digraph_orientation_invalid(self):
30: (8)                     G = nx.DiGraph(self.edges)
31: (8)                     edge_iterator = nx.edge_bfs(G, self.nodes, orientation="hello")
32: (8)                     pytest.raises(nx.NetworkXError, list, edge_iterator)
33: (4)                 def test_digraph_orientation_none(self):
34: (8)                     G = nx.DiGraph(self.edges)
35: (8)                     x = list(nx.edge_bfs(G, self.nodes, orientation=None))
```

```
(8)                     x_ = [(0, 1), (1, 0), (2, 0), (2, 1), (3, 1)]
(8)                     assert x == x_
(4)                 def test_digraph_orientation_original(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.edge_bfs(G, self.nodes, orientation="original"))
(8)                     x_ = [
(12)                        (0, 1, FORWARD),
(12)                        (1, 0, FORWARD),
(12)                        (2, 0, FORWARD),
(12)                        (2, 1, FORWARD),
(12)                        (3, 1, FORWARD),
(8)                     ]
(8)                     assert x == x_
(4)                 def test_digraph2(self):
(8)                     G = nx.DiGraph()
(8)                     nx.add_path(G, range(4))
(8)                     x = list(nx.edge_bfs(G, [0]))
(8)                     x_ = [(0, 1), (1, 2), (2, 3)]
(8)                     assert x == x_
(4)                 def test_digraph_rev(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.edge_bfs(G, self.nodes, orientation="reverse"))
(8)                     x_ = [
(12)                        (1, 0, REVERSE),
(12)                        (2, 0, REVERSE),
(12)                        (0, 1, REVERSE),
(12)                        (2, 1, REVERSE),
(12)                        (3, 1, REVERSE),
(8)                     ]
(8)                     assert x == x_
(4)                 def test_digraph_rev2(self):
(8)                     G = nx.DiGraph()
(8)                     nx.add_path(G, range(4))
(8)                     x = list(nx.edge_bfs(G, [3], orientation="reverse"))
(8)                     x_ = [(2, 3, REVERSE), (1, 2, REVERSE), (0, 1, REVERSE)]
(8)                     assert x == x_
(4)                 def test_multigraph(self):
(8)                     G = nx.MultiGraph(self.edges)
(8)                     x = list(nx.edge_bfs(G, self.nodes))
(8)                     x_ = [(0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 2, 0), (1, 2, 0), (1, 3,
0)]
(8)                     assert x == x_
(4)                 def test_multidigraph(self):
(8)                     G = nx.MultiDiGraph(self.edges)
(8)                     x = list(nx.edge_bfs(G, self.nodes))
(8)                     x_ = [(0, 1, 0), (1, 0, 0), (1, 0, 1), (2, 0, 0), (2, 1, 0), (3, 1,
0)]
(8)                     assert x == x_
(4)                 def test_multidigraph_rev(self):
(8)                     G = nx.MultiDiGraph(self.edges)
(8)                     x = list(nx.edge_bfs(G, self.nodes, orientation="reverse"))
(8)                     x_ = [
(12)                        (1, 0, 0, REVERSE),
(12)                        (1, 0, 1, REVERSE),
(12)                        (2, 0, 0, REVERSE),
(12)                        (0, 1, 0, REVERSE),
(12)                        (2, 1, 0, REVERSE),
(12)                        (3, 1, 0, REVERSE),
(8)                     ]
(8)                     assert x == x_
(4)                 def test_digraph_ignore(self):
(8)                     G = nx.DiGraph(self.edges)
(8)                     x = list(nx.edge_bfs(G, self.nodes, orientation="ignore"))
(8)                     x_ = [
(12)                        (0, 1, FORWARD),
(12)                        (1, 0, REVERSE),
(12)                        (2, 0, REVERSE),
(12)                        (2, 1, REVERSE),
(12)                        (3, 1, REVERSE),
```

```
103: (8)                    ]
104: (8)                    assert x == x_
105: (4)                def test_digraph_ignore2(self):
106: (8)                    G = nx.DiGraph()
107: (8)                    nx.add_path(G, range(4))
108: (8)                    x = list(nx.edge_bfs(G, [0], orientation="ignore"))
109: (8)                    x_ = [(0, 1, FORWARD), (1, 2, FORWARD), (2, 3, FORWARD)]
110: (8)                    assert x == x_
111: (4)                def test_multidigraph_ignore(self):
112: (8)                    G = nx.MultiDiGraph(self.edges)
113: (8)                    x = list(nx.edge_bfs(G, self.nodes, orientation="ignore"))
114: (8)                    x_ = [
115: (12)                       (0, 1, 0, FORWARD),
116: (12)                       (1, 0, 0, REVERSE),
117: (12)                       (1, 0, 1, REVERSE),
118: (12)                       (2, 0, 0, REVERSE),
119: (12)                       (2, 1, 0, REVERSE),
120: (12)                       (3, 1, 0, REVERSE),
121: (8)                    ]
122: (8)                    assert x == x_
```

File 373 - test_edgedfs.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms import edge_dfs
4: (0)              from networkx.algorithms.traversal.edgedfs import FORWARD, REVERSE
5: (0)              class TestEdgeDFS:
6: (4)                  @classmethod
7: (4)                  def setup_class(cls):
8: (8)                      cls.nodes = [0, 1, 2, 3]
9: (8)                      cls.edges = [(0, 1), (1, 0), (1, 0), (2, 1), (3, 1)]
10: (4)                 def test_empty(self):
11: (8)                     G = nx.Graph()
12: (8)                     edges = list(edge_dfs(G))
13: (8)                     assert edges == []
14: (4)                 def test_graph(self):
15: (8)                     G = nx.Graph(self.edges)
16: (8)                     x = list(edge_dfs(G, self.nodes))
17: (8)                     x_ = [(0, 1), (1, 2), (1, 3)]
18: (8)                     assert x == x_
19: (4)                 def test_digraph(self):
20: (8)                     G = nx.DiGraph(self.edges)
21: (8)                     x = list(edge_dfs(G, self.nodes))
22: (8)                     x_ = [(0, 1), (1, 0), (2, 1), (3, 1)]
23: (8)                     assert x == x_
24: (4)                 def test_digraph_orientation_invalid(self):
25: (8)                     G = nx.DiGraph(self.edges)
26: (8)                     edge_iterator = edge_dfs(G, self.nodes, orientation="hello")
27: (8)                     pytest.raises(nx.NetworkXError, list, edge_iterator)
28: (4)                 def test_digraph_orientation_none(self):
29: (8)                     G = nx.DiGraph(self.edges)
30: (8)                     x = list(edge_dfs(G, self.nodes, orientation=None))
31: (8)                     x_ = [(0, 1), (1, 0), (2, 1), (3, 1)]
32: (8)                     assert x == x_
33: (4)                 def test_digraph_orientation_original(self):
34: (8)                     G = nx.DiGraph(self.edges)
35: (8)                     x = list(edge_dfs(G, self.nodes, orientation="original"))
36: (8)                     x_ = [(0, 1, FORWARD), (1, 0, FORWARD), (2, 1, FORWARD), (3, 1,
FORWARD)]
37: (8)                     assert x == x_
38: (4)                 def test_digraph2(self):
39: (8)                     G = nx.DiGraph()
40: (8)                     nx.add_path(G, range(4))
41: (8)                     x = list(edge_dfs(G, [0]))
42: (8)                     x_ = [(0, 1), (1, 2), (2, 3)]
43: (8)                     assert x == x_
```

```
44: (4)                 def test_digraph_rev(self):
45: (8)                     G = nx.DiGraph(self.edges)
46: (8)                     x = list(edge_dfs(G, self.nodes, orientation="reverse"))
47: (8)                     x_ = [(1, 0, REVERSE), (0, 1, REVERSE), (2, 1, REVERSE), (3, 1,
REVERSE)]
48: (8)                     assert x == x_
49: (4)                 def test_digraph_rev2(self):
50: (8)                     G = nx.DiGraph()
51: (8)                     nx.add_path(G, range(4))
52: (8)                     x = list(edge_dfs(G, [3], orientation="reverse"))
53: (8)                     x_ = [(2, 3, REVERSE), (1, 2, REVERSE), (0, 1, REVERSE)]
54: (8)                     assert x == x_
55: (4)                 def test_multigraph(self):
56: (8)                     G = nx.MultiGraph(self.edges)
57: (8)                     x = list(edge_dfs(G, self.nodes))
58: (8)                     x_ = [(0, 1, 0), (1, 0, 1), (0, 1, 2), (1, 2, 0), (1, 3, 0)]
59: (8)                     assert x == x_
60: (4)                 def test_multidigraph(self):
61: (8)                     G = nx.MultiDiGraph(self.edges)
62: (8)                     x = list(edge_dfs(G, self.nodes))
63: (8)                     x_ = [(0, 1, 0), (1, 0, 0), (1, 0, 1), (2, 1, 0), (3, 1, 0)]
64: (8)                     assert x == x_
65: (4)                 def test_multidigraph_rev(self):
66: (8)                     G = nx.MultiDiGraph(self.edges)
67: (8)                     x = list(edge_dfs(G, self.nodes, orientation="reverse"))
68: (8)                     x_ = [
69: (12)                        (1, 0, 0, REVERSE),
70: (12)                        (0, 1, 0, REVERSE),
71: (12)                        (1, 0, 1, REVERSE),
72: (12)                        (2, 1, 0, REVERSE),
73: (12)                        (3, 1, 0, REVERSE),
74: (8)                     ]
75: (8)                     assert x == x_
76: (4)                 def test_digraph_ignore(self):
77: (8)                     G = nx.DiGraph(self.edges)
78: (8)                     x = list(edge_dfs(G, self.nodes, orientation="ignore"))
79: (8)                     x_ = [(0, 1, FORWARD), (1, 0, FORWARD), (2, 1, REVERSE), (3, 1,
REVERSE)]
80: (8)                     assert x == x_
81: (4)                 def test_digraph_ignore2(self):
82: (8)                     G = nx.DiGraph()
83: (8)                     nx.add_path(G, range(4))
84: (8)                     x = list(edge_dfs(G, [0], orientation="ignore"))
85: (8)                     x_ = [(0, 1, FORWARD), (1, 2, FORWARD), (2, 3, FORWARD)]
86: (8)                     assert x == x_
87: (4)                 def test_multidigraph_ignore(self):
88: (8)                     G = nx.MultiDiGraph(self.edges)
89: (8)                     x = list(edge_dfs(G, self.nodes, orientation="ignore"))
90: (8)                     x_ = [
91: (12)                        (0, 1, 0, FORWARD),
92: (12)                        (1, 0, 0, FORWARD),
93: (12)                        (1, 0, 1, REVERSE),
94: (12)                        (2, 1, 0, REVERSE),
95: (12)                        (3, 1, 0, REVERSE),
96: (8)                     ]
97: (8)                     assert x == x_
```

File 374 - __init__.py:

```
1: (0)              from .branchings import *
2: (0)              from .coding import *
3: (0)              from .mst import *
4: (0)              from .recognition import *
5: (0)              from .operations import *
6: (0)              from .decomposition import *
```

File 375 - branchings.py:

```
1: (0)              """
2: (0)              Algorithms for finding optimum branchings and spanning arborescences.
3: (0)              This implementation is based on:
4: (4)                  J. Edmonds, Optimum branchings, J. Res. Natl. Bur. Standards 71B (1967),
5: (4)                  233–240. URL: http://archive.org/details/jresv71Bn4p233
6: (0)              """
7: (0)              import string
8: (0)              from dataclasses import dataclass, field
9: (0)              from operator import itemgetter
10: (0)             from queue import PriorityQueue
11: (0)             import networkx as nx
12: (0)             from networkx.utils import py_random_state
13: (0)             from .recognition import is_arborescence, is_branching
14: (0)             __all__ = [
15: (4)                 "branching_weight",
16: (4)                 "greedy_branching",
17: (4)                 "maximum_branching",
18: (4)                 "minimum_branching",
19: (4)                 "minimal_branching",
20: (4)                 "maximum_spanning_arborescence",
21: (4)                 "minimum_spanning_arborescence",
22: (4)                 "ArborescenceIterator",
23: (4)                 "Edmonds",
24: (0)             ]
25: (0)             KINDS = {"max", "min"}
26: (0)             STYLES = {
27: (4)                 "branching": "branching",
28: (4)                 "arborescence": "arborescence",
29: (4)                 "spanning arborescence": "arborescence",
30: (0)             }
31: (0)             INF = float("inf")
32: (0)             @py_random_state(1)
33: (0)             def random_string(L=15, seed=None):
34: (4)                 return "".join([seed.choice(string.ascii_letters) for n in range(L)])
35: (0)             def _min_weight(weight):
36: (4)                 return -weight
37: (0)             def _max_weight(weight):
38: (4)                 return weight
39: (0)             @nx._dispatchable(edge_attrs={"attr": "default"})
40: (0)             def branching_weight(G, attr="weight", default=1):
41: (4)                 """
42: (4)                 Returns the total weight of a branching.
43: (4)                 You must access this function through the networkx.algorithms.tree module.
44: (4)                 Parameters
45: (4)                 ----------
46: (4)                 G : DiGraph
47: (8)                     The directed graph.
48: (4)                 attr : str
49: (8)                     The attribute to use as weights. If None, then each edge will be
50: (8)                     treated equally with a weight of 1.
51: (4)                 default : float
52: (8)                     When `attr` is not None, then if an edge does not have that attribute,
53: (8)                     `default` specifies what value it should take.
54: (4)                 Returns
55: (4)                 -------
56: (4)                 weight: int or float
57: (8)                     The total weight of the branching.
58: (4)                 Examples
59: (4)                 --------
60: (4)                 >>> G = nx.DiGraph()
61: (4)                 >>> G.add_weighted_edges_from([(0, 1, 2), (1, 2, 4), (2, 3, 3), (3, 4,
2)])
62: (4)                 >>> nx.tree.branching_weight(G)
63: (4)                 11
64: (4)                 """
65: (4)                 return sum(edge[2].get(attr, default) for edge in G.edges(data=True))
```

```
66: (0)             @py_random_state(4)
67: (0)             @nx._dispatchable(edge_attrs={"attr": "default"}, returns_graph=True)
68: (0)             def greedy_branching(G, attr="weight", default=1, kind="max", seed=None):
69: (4)                 """
70: (4)                 Returns a branching obtained through a greedy algorithm.
71: (4)                 This algorithm is wrong, and cannot give a proper optimal branching.
72: (4)                 However, we include it for pedagogical reasons, as it can be helpful to
73: (4)                 see what its outputs are.
74: (4)                 The output is a branching, and possibly, a spanning arborescence. However,
75: (4)                 it is not guaranteed to be optimal in either case.
76: (4)                 Parameters
77: (4)                 ----------
78: (4)                 G : DiGraph
79: (8)                     The directed graph to scan.
80: (4)                 attr : str
81: (8)                     The attribute to use as weights. If None, then each edge will be
82: (8)                     treated equally with a weight of 1.
83: (4)                 default : float
84: (8)                     When `attr` is not None, then if an edge does not have that attribute,
85: (8)                     `default` specifies what value it should take.
86: (4)                 kind : str
87: (8)                     The type of optimum to search for: 'min' or 'max' greedy branching.
88: (4)                 seed : integer, random_state, or None (default)
89: (8)                     Indicator of random number generation state.
90: (8)                     See :ref:`Randomness<randomness>`.
91: (4)                 Returns
92: (4)                 -------
93: (4)                 B : directed graph
94: (8)                     The greedily obtained branching.
95: (4)                 """
96: (4)                 if kind not in KINDS:
97: (8)                     raise nx.NetworkXException("Unknown value for `kind`.")
98: (4)                 if kind == "min":
99: (8)                     reverse = False
100: (4)                else:
101: (8)                    reverse = True
102: (4)                if attr is None:
103: (8)                    attr = random_string(seed=seed)
104: (4)                edges = [(u, v, data.get(attr, default)) for (u, v, data) in
G.edges(data=True)]
105: (4)                try:
106: (8)                    edges.sort(key=itemgetter(2, 0, 1), reverse=reverse)
107: (4)                except TypeError:
108: (8)                    edges.sort(key=itemgetter(2), reverse=reverse)
109: (4)                B = nx.DiGraph()
110: (4)                B.add_nodes_from(G)
111: (4)                uf = nx.utils.UnionFind()
112: (4)                for i, (u, v, w) in enumerate(edges):
113: (8)                    if uf[u] == uf[v]:
114: (12)                       continue
115: (8)                    elif B.in_degree(v) == 1:
116: (12)                       continue
117: (8)                    else:
118: (12)                       data = {}
119: (12)                       if attr is not None:
120: (16)                           data[attr] = w
121: (12)                       B.add_edge(u, v, **data)
122: (12)                       uf.union(u, v)
123: (4)                return B
124: (0)            class MultiDiGraph_EdgeKey(nx.MultiDiGraph):
125: (4)                """
126: (4)                MultiDiGraph which assigns unique keys to every edge.
127: (4)                Adds a dictionary edge_index which maps edge keys to (u, v, data) tuples.
128: (4)                This is not a complete implementation. For Edmonds algorithm, we only use
129: (4)                add_node and add_edge, so that is all that is implemented here. During
130: (4)                additions, any specified keys are ignored---this means that you also
131: (4)                cannot update edge attributes through add_node and add_edge.
132: (4)                Why do we need this? Edmonds algorithm requires that we track edges, even
133: (4)                as we change the head and tail of an edge, and even changing the weight
```

```
134: (4)                of edges. We must reliably track edges across graph mutations.
135: (4)                """
136: (4)                def __init__(self, incoming_graph_data=None, **attr):
137: (8)                    cls = super()
138: (8)                    cls.__init__(incoming_graph_data=incoming_graph_data, **attr)
139: (8)                    self._cls = cls
140: (8)                    self.edge_index = {}
141: (8)                    import warnings
142: (8)                    msg = "MultiDiGraph_EdgeKey has been deprecated and will be removed in
NetworkX 3.4."
143: (8)                    warnings.warn(msg, DeprecationWarning)
144: (4)                def remove_node(self, n):
145: (8)                    keys = set()
146: (8)                    for keydict in self.pred[n].values():
147: (12)                       keys.update(keydict)
148: (8)                    for keydict in self.succ[n].values():
149: (12)                       keys.update(keydict)
150: (8)                    for key in keys:
151: (12)                       del self.edge_index[key]
152: (8)                    self._cls.remove_node(n)
153: (4)                def remove_nodes_from(self, nbunch):
154: (8)                    for n in nbunch:
155: (12)                       self.remove_node(n)
156: (4)                def add_edge(self, u_for_edge, v_for_edge, key_for_edge, **attr):
157: (8)                    """
158: (8)                    Key is now required.
159: (8)                    """
160: (8)                    u, v, key = u_for_edge, v_for_edge, key_for_edge
161: (8)                    if key in self.edge_index:
162: (12)                       uu, vv, _ = self.edge_index[key]
163: (12)                       if (u != uu) or (v != vv):
164: (16)                           raise Exception(f"Key {key!r} is already in use.")
165: (8)                    self._cls.add_edge(u, v, key, **attr)
166: (8)                    self.edge_index[key] = (u, v, self.succ[u][v][key])
167: (4)                def add_edges_from(self, ebunch_to_add, **attr):
168: (8)                    for u, v, k, d in ebunch_to_add:
169: (12)                       self.add_edge(u, v, k, **d)
170: (4)                def remove_edge_with_key(self, key):
171: (8)                    try:
172: (12)                       u, v, _ = self.edge_index[key]
173: (8)                    except KeyError as err:
174: (12)                       raise KeyError(f"Invalid edge key {key!r}") from err
175: (8)                    else:
176: (12)                       del self.edge_index[key]
177: (12)                       self._cls.remove_edge(u, v, key)
178: (4)                def remove_edges_from(self, ebunch):
179: (8)                    raise NotImplementedError
180: (0)            def get_path(G, u, v):
181: (4)                """
182: (4)                Returns the edge keys of the unique path between u and v.
183: (4)                This is not a generic function. G must be a branching and an instance of
184: (4)                MultiDiGraph_EdgeKey.
185: (4)                """
186: (4)                nodes = nx.shortest_path(G, u, v)
187: (4)                def first_key(i, vv):
188: (8)                    keys = G[nodes[i]][vv].keys()
189: (8)                    keys = list(keys)
190: (8)                    return keys[0]
191: (4)                edges = [first_key(i, vv) for i, vv in enumerate(nodes[1:])]
192: (4)                return nodes, edges
193: (0)            class Edmonds:
194: (4)                """
195: (4)                Edmonds algorithm [1]_ for finding optimal branchings and spanning
196: (4)                arborescences.
197: (4)                This algorithm can find both minimum and maximum spanning arborescences
and
198: (4)                branchings.
199: (4)                Notes
200: (4)                -----
```

```
    While this algorithm can find a minimum branching, since it isn't required
    to be spanning, the minimum branching is always from the set of negative
    weight edges which is most likely the empty set for most graphs.
    References
    ----------
    .. [1] J. Edmonds, Optimum Branchings, Journal of Research of the National
           Bureau of Standards, 1967, Vol. 71B, p.233-240,
           https://archive.org/details/jresv71Bn4p233
    """
    def __init__(self, G, seed=None):
        self.G_original = G
        self.store = True
        self.edges = []
        self.template = random_string(seed=seed) + "_{0}"
        import warnings
        msg = "Edmonds has been deprecated and will be removed in NetworkX 3.4. Please use the appropriate minimum or maximum branching or arborescence function directly."
        warnings.warn(msg, DeprecationWarning)
    def _init(self, attr, default, kind, style, preserve_attrs, seed, partition):
        """
        So we need the code in _init and find_optimum to successfully run edmonds algorithm.
        Responsibilities of the _init function:
        - Check that the kind argument is in {min, max} or raise a NetworkXException.
        - Transform the graph if we need a minimum arborescence/branching.
          - The current method is to map weight -> -weight. This is NOT a good approach since
            the algorithm can and does choose to ignore negative weights when creating a branching
            since that is always optimal when maximzing the weights. I think we should set the edge
            weights to be (max_weight + 1) - edge_weight.
        - Transform the graph into a MultiDiGraph, adding the partition information and potoentially
          other edge attributes if we set preserve_attrs = True.
        - Setup the buckets and union find data structures required for the algorithm.
        """
        if kind not in KINDS:
            raise nx.NetworkXException("Unknown value for `kind`.")
        self.attr = attr
        self.default = default
        self.kind = kind
        self.style = style
        if kind == "min":
            self.trans = trans = _min_weight
        else:
            self.trans = trans = _max_weight
        if attr is None:
            attr = random_string(seed=seed)
        self._attr = attr
        self.candidate_attr = "candidate_" + random_string(seed=seed)
        self.G = G = MultiDiGraph_EdgeKey()
        self.G.__networkx_cache__ = None  # Disable caching
        for key, (u, v, data) in enumerate(self.G_original.edges(data=True)):
            d = {attr: trans(data.get(attr, default))}
            if data.get(partition) is not None:
                d[partition] = data.get(partition)
            if preserve_attrs:
                for d_k, d_v in data.items():
                    if d_k != attr:
                        d[d_k] = d_v
            G.add_edge(u, v, key, **d)
        self.level = 0
        self.B = MultiDiGraph_EdgeKey()
        self.B.edge_index = {}
        self.graphs = []  # G^i
```

```
        self.branchings = []  # B^i
        self.uf = nx.utils.UnionFind()
        self.circuits = []
        self.minedge_circuit = []
    def find_optimum(
        self,
        attr="weight",
        default=1,
        kind="max",
        style="branching",
        preserve_attrs=False,
        partition=None,
        seed=None,
    ):
        """
        Returns a branching from G.
        Parameters
        ----------
        attr : str
            The edge attribute used to in determining optimality.
        default : float
            The value of the edge attribute used if an edge does not have
            the attribute `attr`.
        kind : {'min', 'max'}
            The type of optimum to search for, either 'min' or 'max'.
        style : {'branching', 'arborescence'}
            If 'branching', then an optimal branching is found. If `style` is
            'arborescence', then a branching is found, such that if the
            branching is also an arborescence, then the branching is an
            optimal spanning arborescences. A given graph G need not have
            an optimal spanning arborescence.
        preserve_attrs : bool
            If True, preserve the other edge attributes of the original
            graph (that are not the one passed to `attr`)
        partition : str
            The edge attribute holding edge partition data. Used in the
            spanning arborescence iterator.
        seed : integer, random_state, or None (default)
            Indicator of random number generation state.
            See :ref:`Randomness<randomness>`.
        Returns
        -------
        H : (multi)digraph
            The branching.
        """
        self._init(attr, default, kind, style, preserve_attrs, seed, partition)
        uf = self.uf
        G, B = self.G, self.B
        D = set()
        nodes = iter(list(G.nodes()))
        attr = self._attr
        def desired_edge(v):
            """
            Find the edge directed toward v with maximal weight.
            If an edge partition exists in this graph, return the included edge

            if it exists and no not return any excluded edges. There can only
            be one included edge for each vertex otherwise the edge partition is

            empty.
            """
            edge = None
            weight = -INF
            for u, _, key, data in G.in_edges(v, data=True, keys=True):
                if data.get(partition) == nx.EdgePartition.EXCLUDED:
                    continue
                new_weight = data[attr]
                if data.get(partition) == nx.EdgePartition.INCLUDED:
```

```
327: (20)                                weight = new_weight
328: (20)                                edge = (u, v, key, new_weight, data)
329: (20)                                return edge, weight
330: (16)                            if new_weight > weight:
331: (20)                                weight = new_weight
332: (20)                                edge = (u, v, key, new_weight, data)
333: (12)                        return edge, weight
334: (8)                     while True:
335: (12)                        try:
336: (16)                            v = next(nodes)
337: (12)                        except StopIteration:
338: (16)                            assert len(G) == len(B)
339: (16)                            if len(B):
340: (20)                                assert is_branching(B)
341: (16)                            if self.store:
342: (20)                                self.graphs.append(G.copy())
343: (20)                                self.branchings.append(B.copy())
344: (20)                                self.circuits.append([])
345: (20)                                self.minedge_circuit.append(None)
346: (16)                            break
347: (12)                        else:
348: (16)                            if v in D:
349: (20)                                continue
350: (12)                        D.add(v)
351: (12)                        B.add_node(v)
352: (12)                        edge, weight = desired_edge(v)
353: (12)                        if edge is None:
354: (16)                            continue
355: (12)                        else:
356: (16)                            u = edge[0]
357: (16)                            if uf[u] == uf[v]:
358: (20)                                Q_nodes, Q_edges = get_path(B, v, u)
359: (20)                                Q_edges.append(edge[2])  # Edge key
360: (16)                            else:
361: (20)                                Q_nodes, Q_edges = None, None
362: (16)                            if self.style == "branching" and weight <= 0:
363: (20)                                acceptable = False
364: (16)                            else:
365: (20)                                acceptable = True
366: (16)                            if acceptable:
367: (20)                                dd = {attr: weight}
368: (20)                                if edge[4].get(partition) is not None:
369: (24)                                    dd[partition] = edge[4].get(partition)
370: (20)                                B.add_edge(u, v, edge[2], **dd)
371: (20)                                G[u][v][edge[2]][self.candidate_attr] = True
372: (20)                                uf.union(u, v)
373: (20)                                if Q_edges is not None:
374: (24)                                    minweight = INF
375: (24)                                    minedge = None
376: (24)                                    Q_incoming_weight = {}
377: (24)                                    for edge_key in Q_edges:
378: (28)                                        u, v, data = B.edge_index[edge_key]
379: (28)                                        w = data[attr]
380: (28)                                        Q_incoming_weight[v] = w
381: (28)                                        if data.get(partition) ==
nx.EdgePartition.INCLUDED:
382: (32)                                            continue
383: (28)                                        if w < minweight:
384: (32)                                            minweight = w
385: (32)                                            minedge = edge_key
386: (24)                                    self.circuits.append(Q_edges)
387: (24)                                    self.minedge_circuit.append(minedge)
388: (24)                                    if self.store:
389: (28)                                        self.graphs.append(G.copy())
390: (24)                                    self.branchings.append(B.copy())
391: (24)                                    new_node = self.template.format(self.level)
392: (24)                                    G.add_node(new_node)
393: (24)                                    new_edges = []
394: (24)                                    for u, v, key, data in G.edges(data=True, keys=True):
```

```
395: (28)                                        if u in Q_incoming_weight:
396: (32)                                            if v in Q_incoming_weight:
397: (36)                                                continue
398: (32)                                            else:
399: (36)                                                dd = data.copy()
400: (36)                                                new_edges.append((new_node, v, key, dd))
401: (28)                                        else:
402: (32)                                            if v in Q_incoming_weight:
403: (36)                                                w = data[attr]
404: (36)                                                w += minweight - Q_incoming_weight[v]
405: (36)                                                dd = data.copy()
406: (36)                                                dd[attr] = w
407: (36)                                                new_edges.append((u, new_node, key, dd))
408: (32)                                            else:
409: (36)                                                continue
410: (24)                                    G.remove_nodes_from(Q_nodes)
411: (24)                                    B.remove_nodes_from(Q_nodes)
412: (24)                                    D.difference_update(set(Q_nodes))
413: (24)                                    for u, v, key, data in new_edges:
414: (28)                                        G.add_edge(u, v, key, **data)
415: (28)                                        if self.candidate_attr in data:
416: (32)                                            del data[self.candidate_attr]
417: (32)                                            B.add_edge(u, v, key, **data)
418: (32)                                            uf.union(u, v)
419: (24)                                    nodes = iter(list(G.nodes()))
420: (24)                                    self.level += 1
421: (8)                    H = self.G_original.__class__()
422: (8)                    def is_root(G, u, edgekeys):
423: (12)                       """
424: (12)                       Returns True if `u` is a root node in G.
425: (12)                       Node `u` will be a root node if its in-degree, restricted to the
426: (12)                       specified edges, is equal to 0.
427: (12)                       """
428: (12)                       if u not in G:
429: (16)                           raise Exception(f"{u!r} not in G")
430: (12)                       for v in G.pred[u]:
431: (16)                           for edgekey in G.pred[u][v]:
432: (20)                               if edgekey in edgekeys:
433: (24)                                   return False, edgekey
434: (12)                       else:
435: (16)                           return True, None
436: (8)                    edges = set(self.branchings[self.level].edge_index)
437: (8)                    while self.level > 0:
438: (12)                       self.level -= 1
439: (12)                       merged_node = self.template.format(self.level)
440: (12)                       circuit = self.circuits[self.level]
441: (12)                       isroot, edgekey = is_root(self.graphs[self.level + 1],
merged_node, edges)
442: (12)                       edges.update(circuit)
443: (12)                       if isroot:
444: (16)                           minedge = self.minedge_circuit[self.level]
445: (16)                           if minedge is None:
446: (20)                               raise Exception
447: (16)                           edges.remove(minedge)
448: (12)                       else:
449: (16)                           G = self.graphs[self.level]
450: (16)                           target = G.edge_index[edgekey][1]
451: (16)                           for edgekey in circuit:
452: (20)                               u, v, data = G.edge_index[edgekey]
453: (20)                               if v == target:
454: (24)                                   break
455: (16)                           else:
456: (20)                               raise Exception("Couldn't find edge incoming to merged
node.")
457: (16)                           edges.remove(edgekey)
458: (8)                    self.edges = edges
459: (8)                    H.add_nodes_from(self.G_original)
460: (8)                    for edgekey in edges:
461: (12)                       u, v, d = self.graphs[0].edge_index[edgekey]
```

```
462: (12)                        dd = {self.attr: self.trans(d[self.attr])}
463: (12)                        if preserve_attrs:
464: (16)                            for key, value in d.items():
465: (20)                                if key not in [self.attr, self.candidate_attr]:
466: (24)                                    dd[key] = value
467: (12)                        H.add_edge(u, v, **dd)
468: (8)                    return H
469: (0)            @nx._dispatchable(preserve_edge_attrs=True, returns_graph=True)
470: (0)            def maximum_branching(
471: (4)                G,
472: (4)                attr="weight",
473: (4)                default=1,
474: (4)                preserve_attrs=False,
475: (4)                partition=None,
476: (0)            ):
477: (4)                def edmonds_add_edge(G, edge_index, u, v, key, **d):
478: (8)                    """
479: (8)                    Adds an edge to `G` while also updating the edge index.
480: (8)                    This algorithm requires the use of an external dictionary to track
481: (8)                    the edge keys since it is possible that the source or destination
482: (8)                    node of an edge will be changed and the default key-handling
483: (8)                    capabilities of the MultiDiGraph class do not account for this.
484: (8)                    Parameters
485: (8)                    ----------
486: (8)                    G : MultiDiGraph
487: (12)                       The graph to insert an edge into.
488: (8)                    edge_index : dict
489: (12)                       A mapping from integers to the edges of the graph.
490: (8)                    u : node
491: (12)                       The source node of the new edge.
492: (8)                    v : node
493: (12)                       The destination node of the new edge.
494: (8)                    key : int
495: (12)                       The key to use from `edge_index`.
496: (8)                    d : keyword arguments, optional
497: (12)                       Other attributes to store on the new edge.
498: (8)                    """
499: (8)                    if key in edge_index:
500: (12)                       uu, vv, _ = edge_index[key]
501: (12)                       if (u != uu) or (v != vv):
502: (16)                           raise Exception(f"Key {key!r} is already in use.")
503: (8)                    G.add_edge(u, v, key, **d)
504: (8)                    edge_index[key] = (u, v, G.succ[u][v][key])
505: (4)                def edmonds_remove_node(G, edge_index, n):
506: (8)                    """
507: (8)                    Remove a node from the graph, updating the edge index to match.
508: (8)                    Parameters
509: (8)                    ----------
510: (8)                    G : MultiDiGraph
511: (12)                       The graph to remove an edge from.
512: (8)                    edge_index : dict
513: (12)                       A mapping from integers to the edges of the graph.
514: (8)                    n : node
515: (12)                       The node to remove from `G`.
516: (8)                    """
517: (8)                    keys = set()
518: (8)                    for keydict in G.pred[n].values():
519: (12)                       keys.update(keydict)
520: (8)                    for keydict in G.succ[n].values():
521: (12)                       keys.update(keydict)
522: (8)                    for key in keys:
523: (12)                       del edge_index[key]
524: (8)                    G.remove_node(n)
525: (4)                candidate_attr = "edmonds' secret candidate attribute"
526: (4)                new_node_base_name = "edmonds new node base name "
527: (4)                G_original = G
528: (4)                G = nx.MultiDiGraph()
529: (4)                G.__networkx_cache__ = None  # Disable caching
530: (4)                G_edge_index = {}
```

```
    for key, (u, v, data) in enumerate(G_original.edges(data=True)):
        d = {attr: data.get(attr, default)}
        if data.get(partition) is not None:
            d[partition] = data.get(partition)
        if preserve_attrs:
            for d_k, d_v in data.items():
                if d_k != attr:
                    d[d_k] = d_v
        edmonds_add_edge(G, G_edge_index, u, v, key, **d)
    level = 0  # Stores the number of contracted nodes
    B = nx.MultiDiGraph()
    B_edge_index = {}
    graphs = []  # G^i list
    branchings = []  # B^i list
    selected_nodes = set()  # D^i bucket
    uf = nx.utils.UnionFind()
    circuits = []
    minedge_circuit = []
    def edmonds_find_desired_edge(v):
        """
        Find the edge directed towards v with maximal weight.
        If an edge partition exists in this graph, return the included
        edge if it exists and never return any excluded edge.
        Note: There can only be one included edge for each vertex otherwise
        the edge partition is empty.
        Parameters
        ----------
        v : node
            The node to search for the maximal weight incoming edge.
        """
        edge = None
        max_weight = -INF
        for u, _, key, data in G.in_edges(v, data=True, keys=True):
            if data.get(partition) == nx.EdgePartition.EXCLUDED:
                continue
            new_weight = data[attr]
            if data.get(partition) == nx.EdgePartition.INCLUDED:
                max_weight = new_weight
                edge = (u, v, key, new_weight, data)
                break
            if new_weight > max_weight:
                max_weight = new_weight
                edge = (u, v, key, new_weight, data)
        return edge, max_weight
    def edmonds_step_I2(v, desired_edge, level):
        """
        Perform step I2 from Edmonds' paper
        First, check if the last step I1 created a cycle. If it did not, do nothing.
        If it did, store the cycle for later reference and contract it.
        Parameters
        ----------
        v : node
            The current node to consider
        desired_edge : edge
            The minimum desired edge to remove from the cycle.
        level : int
            The current level, i.e. the number of cycles that have already been removed.
        """
        u = desired_edge[0]
        Q_nodes = nx.shortest_path(B, v, u)
        Q_edges = [
            list(B[Q_nodes[i]][vv].keys())[0] for i, vv in enumerate(Q_nodes[1:])
        ]
        Q_edges.append(desired_edge[2])  # Add the new edge key to complete the circuit
        minweight = INF
```

```
(8)                     minedge = None
(8)                     Q_incoming_weight = {}
(8)                     for edge_key in Q_edges:
(12)                        u, v, data = B_edge_index[edge_key]
(12)                        w = data[attr]
(12)                        Q_incoming_weight[v] = w
(12)                        if data.get(partition) == nx.EdgePartition.INCLUDED:
(16)                            continue
(12)                        if w < minweight:
(16)                            minweight = w
(16)                            minedge = edge_key
(8)                     circuits.append(Q_edges)
(8)                     minedge_circuit.append(minedge)
(8)                     graphs.append((G.copy(), G_edge_index.copy()))
(8)                     branchings.append((B.copy(), B_edge_index.copy()))
(8)                     new_node = new_node_base_name + str(level)
(8)                     G.add_node(new_node)
(8)                     new_edges = []
(8)                     for u, v, key, data in G.edges(data=True, keys=True):
(12)                        if u in Q_incoming_weight:
(16)                            if v in Q_incoming_weight:
(20)                                continue
(16)                            else:
(20)                                dd = data.copy()
(20)                                new_edges.append((new_node, v, key, dd))
(12)                        else:
(16)                            if v in Q_incoming_weight:
(20)                                w = data[attr]
(20)                                w += minweight - Q_incoming_weight[v]
(20)                                dd = data.copy()
(20)                                dd[attr] = w
(20)                                new_edges.append((u, new_node, key, dd))
(16)                            else:
(20)                                continue
(8)                     for node in Q_nodes:
(12)                        edmonds_remove_node(G, G_edge_index, node)
(12)                        edmonds_remove_node(B, B_edge_index, node)
(8)                     selected_nodes.difference_update(set(Q_nodes))
(8)                     for u, v, key, data in new_edges:
(12)                        edmonds_add_edge(G, G_edge_index, u, v, key, **data)
(12)                        if candidate_attr in data:
(16)                            del data[candidate_attr]
(16)                            edmonds_add_edge(B, B_edge_index, u, v, key, **data)
(16)                            uf.union(u, v)
(4)                 def is_root(G, u, edgekeys):
(8)                     """
(8)                     Returns True if `u` is a root node in G.
(8)                     Node `u` is a root node if its in-degree over the specified edges is
zero.
(8)                     Parameters
(8)                     ----------
(8)                     G : Graph
(12)                        The current graph.
(8)                     u : node
(12)                        The node in `G` to check if it is a root.
(8)                     edgekeys : iterable of edges
(12)                        The edges for which to check if `u` is a root of.
(8)                     """
(8)                     if u not in G:
(12)                        raise Exception(f"{u!r} not in G")
(8)                     for v in G.pred[u]:
(12)                        for edgekey in G.pred[u][v]:
(16)                            if edgekey in edgekeys:
(20)                                return False, edgekey
(8)                     else:
(12)                        return True, None
(4)                 nodes = iter(list(G.nodes))
(4)                 while True:
(8)                     try:
```

```
(12)                        v = next(nodes)
(8)                     except StopIteration:
(12)                        assert len(G) == len(B)
(12)                        if len(B):
(16)                            assert is_branching(B)
(12)                        graphs.append((G.copy(), G_edge_index.copy()))
(12)                        branchings.append((B.copy(), B_edge_index.copy()))
(12)                        circuits.append([])
(12)                        minedge_circuit.append(None)
(12)                        break
(8)                     else:
(12)                        if v in selected_nodes:
(16)                            continue
(8)                     selected_nodes.add(v)
(8)                     B.add_node(v)
(8)                     desired_edge, desired_edge_weight = edmonds_find_desired_edge(v)
(8)                     if desired_edge is not None and desired_edge_weight > 0:
(12)                        u = desired_edge[0]
(12)                        circuit = uf[u] == uf[v]
(12)                        dd = {attr: desired_edge_weight}
(12)                        if desired_edge[4].get(partition) is not None:
(16)                            dd[partition] = desired_edge[4].get(partition)
(12)                        edmonds_add_edge(B, B_edge_index, u, v, desired_edge[2], **dd)
(12)                        G[u][v][desired_edge[2]][candidate_attr] = True
(12)                        uf.union(u, v)
(12)                        if circuit:
(16)                            edmonds_step_I2(v, desired_edge, level)
(16)                            nodes = iter(list(G.nodes()))
(16)                            level += 1
(4)                 H = G_original.__class__()
(4)                 edges = set(branchings[level][1])
(4)                 while level > 0:
(8)                     level -= 1
(8)                     merged_node = new_node_base_name + str(level)
(8)                     circuit = circuits[level]
(8)                     isroot, edgekey = is_root(graphs[level + 1][0], merged_node, edges)
(8)                     edges.update(circuit)
(8)                     if isroot:
(12)                        minedge = minedge_circuit[level]
(12)                        if minedge is None:
(16)                            raise Exception
(12)                        edges.remove(minedge)
(8)                     else:
(12)                        G, G_edge_index = graphs[level]
(12)                        target = G_edge_index[edgekey][1]
(12)                        for edgekey in circuit:
(16)                            u, v, data = G_edge_index[edgekey]
(16)                            if v == target:
(20)                                break
(12)                        else:
(16)                            raise Exception("Couldn't find edge incoming to merged node.")
(12)                        edges.remove(edgekey)
(4)                 H.add_nodes_from(G_original)
(4)                 for edgekey in edges:
(8)                     u, v, d = graphs[0][1][edgekey]
(8)                     dd = {attr: d[attr]}
(8)                     if preserve_attrs:
(12)                        for key, value in d.items():
(16)                            if key not in [attr, candidate_attr]:
(20)                                dd[key] = value
(8)                     H.add_edge(u, v, **dd)
(4)                 return H
(0)             @nx._dispatchable(preserve_edge_attrs=True, mutates_input=True,
returns_graph=True)
(0)             def minimum_branching(
(4)                 G, attr="weight", default=1, preserve_attrs=False, partition=None
(0)             ):
(4)                 for _, _, d in G.edges(data=True):
(8)                     d[attr] = -d.get(attr, default)
```

```
732: (4)                nx._clear_cache(G)
733: (4)                B = maximum_branching(G, attr, default, preserve_attrs, partition)
734: (4)                for _, _, d in G.edges(data=True):
735: (8)                    d[attr] = -d.get(attr, default)
736: (4)                nx._clear_cache(G)
737: (4)                for _, _, d in B.edges(data=True):
738: (8)                    d[attr] = -d.get(attr, default)
739: (4)                nx._clear_cache(B)
740: (4)                return B
741: (0)            @nx._dispatchable(preserve_edge_attrs=True, mutates_input=True,
returns_graph=True)
742: (0)            def minimal_branching(
743: (4)                G, /, *, attr="weight", default=1, preserve_attrs=False, partition=None
744: (0)            ):
745: (4)                """
746: (4)                Returns a minimal branching from `G`.
747: (4)                A minimal branching is a branching similar to a minimal arborescence but
748: (4)                without the requirement that the result is actually a spanning
arborescence.
749: (4)                This allows minimal branchinges to be computed over graphs which may not
750: (4)                have arborescence (such as multiple components).
751: (4)                Parameters
752: (4)                ----------
753: (4)                G : (multi)digraph-like
754: (8)                    The graph to be searched.
755: (4)                attr : str
756: (8)                    The edge attribute used in determining optimality.
757: (4)                default : float
758: (8)                    The value of the edge attribute used if an edge does not have
759: (8)                    the attribute `attr`.
760: (4)                preserve_attrs : bool
761: (8)                    If True, preserve the other attributes of the original graph (that are
not
762: (8)                    passed to `attr`)
763: (4)                partition : str
764: (8)                    The key for the edge attribute containing the partition
765: (8)                    data on the graph. Edges can be included, excluded or open using the
766: (8)                    `EdgePartition` enum.
767: (4)                Returns
768: (4)                -------
769: (4)                B : (multi)digraph-like
770: (8)                    A minimal branching.
771: (4)                """
772: (4)                max_weight = -INF
773: (4)                min_weight = INF
774: (4)                for _, _, w in G.edges(data=attr, default=default):
775: (8)                    if w > max_weight:
776: (12)                       max_weight = w
777: (8)                    if w < min_weight:
778: (12)                       min_weight = w
779: (4)                for _, _, d in G.edges(data=True):
780: (8)                    d[attr] = max_weight + 1 + (max_weight - min_weight) - d.get(attr,
default)
781: (4)                nx._clear_cache(G)
782: (4)                B = maximum_branching(G, attr, default, preserve_attrs, partition)
783: (4)                for _, _, d in G.edges(data=True):
784: (8)                    d[attr] = max_weight + 1 + (max_weight - min_weight) - d.get(attr,
default)
785: (4)                nx._clear_cache(G)
786: (4)                for _, _, d in B.edges(data=True):
787: (8)                    d[attr] = max_weight + 1 + (max_weight - min_weight) - d.get(attr,
default)
788: (4)                nx._clear_cache(B)
789: (4)                return B
790: (0)            @nx._dispatchable(preserve_edge_attrs=True, mutates_input=True,
returns_graph=True)
791: (0)            def maximum_spanning_arborescence(
792: (4)                G, attr="weight", default=1, preserve_attrs=False, partition=None
793: (0)            ):
```

```
794: (4)                min_weight = INF
795: (4)                max_weight = -INF
796: (4)                for _, _, w in G.edges(data=attr, default=default):
797: (8)                    if w < min_weight:
798: (12)                       min_weight = w
799: (8)                    if w > max_weight:
800: (12)                       max_weight = w
801: (4)                for _, _, d in G.edges(data=True):
802: (8)                    d[attr] = d.get(attr, default) - min_weight + 1 - (min_weight -
max_weight)
803: (4)                nx._clear_cache(G)
804: (4)                B = maximum_branching(G, attr, default, preserve_attrs, partition)
805: (4)                for _, _, d in G.edges(data=True):
806: (8)                    d[attr] = d.get(attr, default) + min_weight - 1 + (min_weight -
max_weight)
807: (4)                nx._clear_cache(G)
808: (4)                for _, _, d in B.edges(data=True):
809: (8)                    d[attr] = d.get(attr, default) + min_weight - 1 + (min_weight -
max_weight)
810: (4)                nx._clear_cache(B)
811: (4)                if not is_arborescence(B):
812: (8)                    raise nx.exception.NetworkXException("No maximum spanning arborescence
in G.")
813: (4)                return B
814: (0)            @nx._dispatchable(preserve_edge_attrs=True, mutates_input=True,
returns_graph=True)
815: (0)            def minimum_spanning_arborescence(
816: (4)                G, attr="weight", default=1, preserve_attrs=False, partition=None
817: (0)            ):
818: (4)                B = minimal_branching(
819: (8)                    G,
820: (8)                    attr=attr,
821: (8)                    default=default,
822: (8)                    preserve_attrs=preserve_attrs,
823: (8)                    partition=partition,
824: (4)                )
825: (4)                if not is_arborescence(B):
826: (8)                    raise nx.exception.NetworkXException("No minimum spanning arborescence
in G.")
827: (4)                return B
828: (0)            docstring_branching = """
829: (0)            Returns a {kind} {style} from G.
830: (0)            Parameters
831: (0)            ----------
832: (0)            G : (multi)digraph-like
833: (4)                The graph to be searched.
834: (0)            attr : str
835: (4)                The edge attribute used to in determining optimality.
836: (0)            default : float
837: (4)                The value of the edge attribute used if an edge does not have
838: (4)                the attribute `attr`.
839: (0)            preserve_attrs : bool
840: (4)                If True, preserve the other attributes of the original graph (that are not
841: (4)                passed to `attr`)
842: (0)            partition : str
843: (4)                The key for the edge attribute containing the partition
844: (4)                data on the graph. Edges can be included, excluded or open using the
845: (4)                `EdgePartition` enum.
846: (0)            Returns
847: (0)            -------
848: (0)            B : (multi)digraph-like
849: (4)                A {kind} {style}.
850: (0)            """
851: (0)            docstring_arborescence = (
852: (4)                docstring_branching
853: (4)                + """
854: (0)            Raises
855: (0)            ------
856: (0)            NetworkXException
```

```
    If the graph does not contain a {kind} {style}.
"""
)
maximum_branching.__doc__ = docstring_branching.format(
    kind="maximum", style="branching"
)
minimum_branching.__doc__ = (
    docstring_branching.format(kind="minimum", style="branching")
    + """
See Also
--------
    minimal_branching
"""
)
maximum_spanning_arborescence.__doc__ = docstring_arborescence.format(
    kind="maximum", style="spanning arborescence"
)
minimum_spanning_arborescence.__doc__ = docstring_arborescence.format(
    kind="minimum", style="spanning arborescence"
)
class ArborescenceIterator:
    """
    Iterate over all spanning arborescences of a graph in either increasing or
    decreasing cost.
    Notes
    -----
    This iterator uses the partition scheme from [1]_ (included edges,
    excluded edges and open edges). It generates minimum spanning
    arborescences using a modified Edmonds' Algorithm which respects the
    partition of edges. For arborescences with the same weight, ties are
    broken arbitrarily.
    References
    ----------
    .. [1] G.K. Janssens, K. Sörensen, An algorithm to generate all spanning
           trees in order of increasing cost, Pesquisa Operacional, 2005-08,
           Vol. 25 (2), p. 219-229,
           https://www.scielo.br/j/pope/a/XHswBwRwJyrfL88dmMwYNWp/?lang=en
    """
    @dataclass(order=True)
    class Partition:
        """
        This dataclass represents a partition and stores a dict with the edge
        data and the weight of the minimum spanning arborescence of the
        partition dict.
        """
        mst_weight: float
        partition_dict: dict = field(compare=False)
        def __copy__(self):
            return ArborescenceIterator.Partition(
                self.mst_weight, self.partition_dict.copy()
            )
    def __init__(self, G, weight="weight", minimum=True, init_partition=None):
        """
        Initialize the iterator
        Parameters
        ----------
        G : nx.DiGraph
            The directed graph which we need to iterate trees over
        weight : String, default = "weight"
            The edge attribute used to store the weight of the edge
        minimum : bool, default = True
            Return the trees in increasing order while true and decreasing order

            while false.
        init_partition : tuple, default = None
            In the case that certain edges have to be included or excluded from

            the arborescences, `init_partition` should be in the form
            `(included_edges, excluded_edges)` where each edges is a
```

```
924: (12)                        `(u, v)`-tuple inside an iterable such as a list or set.
925: (8)                    """
926: (8)                    self.G = G.copy()
927: (8)                    self.weight = weight
928: (8)                    self.minimum = minimum
929: (8)                    self.method = (
930: (12)                        minimum_spanning_arborescence if minimum else
maximum_spanning_arborescence
931: (8)                    )
932: (8)                    self.partition_key = (
933: (12)                        "ArborescenceIterators super secret partition attribute name"
934: (8)                    )
935: (8)                    if init_partition is not None:
936: (12)                        partition_dict = {}
937: (12)                        for e in init_partition[0]:
938: (16)                            partition_dict[e] = nx.EdgePartition.INCLUDED
939: (12)                        for e in init_partition[1]:
940: (16)                            partition_dict[e] = nx.EdgePartition.EXCLUDED
941: (12)                        self.init_partition = ArborescenceIterator.Partition(0,
partition_dict)
942: (8)                    else:
943: (12)                        self.init_partition = None
944: (4)                def __iter__(self):
945: (8)                    """
946: (8)                    Returns
947: (8)                    -------
948: (8)                    ArborescenceIterator
949: (12)                        The iterator object for this graph
950: (8)                    """
951: (8)                    self.partition_queue = PriorityQueue()
952: (8)                    self._clear_partition(self.G)
953: (8)                    if self.init_partition is not None:
954: (12)                        self._write_partition(self.init_partition)
955: (8)                    mst_weight = self.method(
956: (12)                        self.G,
957: (12)                        self.weight,
958: (12)                        partition=self.partition_key,
959: (12)                        preserve_attrs=True,
960: (8)                    ).size(weight=self.weight)
961: (8)                    self.partition_queue.put(
962: (12)                        self.Partition(
963: (16)                            mst_weight if self.minimum else -mst_weight,
964: (16)                            {}
965: (16)                            if self.init_partition is None
966: (16)                            else self.init_partition.partition_dict,
967: (12)                        )
968: (8)                    )
969: (8)                    return self
970: (4)                def __next__(self):
971: (8)                    """
972: (8)                    Returns
973: (8)                    -------
974: (8)                    (multi)Graph
975: (12)                        The spanning tree of next greatest weight, which ties broken
976: (12)                        arbitrarily.
977: (8)                    """
978: (8)                    if self.partition_queue.empty():
979: (12)                        del self.G, self.partition_queue
980: (12)                        raise StopIteration
981: (8)                    partition = self.partition_queue.get()
982: (8)                    self._write_partition(partition)
983: (8)                    next_arborescence = self.method(
984: (12)                        self.G,
985: (12)                        self.weight,
986: (12)                        partition=self.partition_key,
987: (12)                        preserve_attrs=True,
988: (8)                    )
989: (8)                    self._partition(partition, next_arborescence)
990: (8)                    self._clear_partition(next_arborescence)
```

```
        return next_arborescence
    def _partition(self, partition, partition_arborescence):
        """
        Create new partitions based of the minimum spanning tree of the
        current minimum partition.
        Parameters
        ----------
        partition : Partition
            The Partition instance used to generate the current minimum spanning
            tree.
        partition_arborescence : nx.Graph
            The minimum spanning arborescence of the input partition.
        """
        p1 = self.Partition(0, partition.partition_dict.copy())
        p2 = self.Partition(0, partition.partition_dict.copy())
        for e in partition_arborescence.edges:
            if e not in partition.partition_dict:
                p1.partition_dict[e] = nx.EdgePartition.EXCLUDED
                p2.partition_dict[e] = nx.EdgePartition.INCLUDED
                self._write_partition(p1)
                try:
                    p1_mst = self.method(
                        self.G,
                        self.weight,
                        partition=self.partition_key,
                        preserve_attrs=True,
                    )
                    p1_mst_weight = p1_mst.size(weight=self.weight)
                    p1.mst_weight = p1_mst_weight if self.minimum else -p1_mst_weight
                    self.partition_queue.put(p1.__copy__())
                except nx.NetworkXException:
                    pass
                p1.partition_dict = p2.partition_dict.copy()
    def _write_partition(self, partition):
        """
        Writes the desired partition into the graph to calculate the minimum
        spanning tree. Also, if one incoming edge is included, mark all others
        as excluded so that if that vertex is merged during Edmonds' algorithm
        we cannot still pick another of that vertex's included edges.
        Parameters
        ----------
        partition : Partition
            A Partition dataclass describing a partition on the edges of the
            graph.
        """
        for u, v, d in self.G.edges(data=True):
            if (u, v) in partition.partition_dict:
                d[self.partition_key] = partition.partition_dict[(u, v)]
            else:
                d[self.partition_key] = nx.EdgePartition.OPEN
        nx._clear_cache(self.G)
        for n in self.G:
            included_count = 0
            excluded_count = 0
            for u, v, d in self.G.in_edges(nbunch=n, data=True):
                if d.get(self.partition_key) == nx.EdgePartition.INCLUDED:
                    included_count += 1
                elif d.get(self.partition_key) == nx.EdgePartition.EXCLUDED:
                    excluded_count += 1
            if included_count == 1 and excluded_count != self.G.in_degree(n) - 1:
                for u, v, d in self.G.in_edges(nbunch=n, data=True):
                    if d.get(self.partition_key) != nx.EdgePartition.INCLUDED:
                        d[self.partition_key] = nx.EdgePartition.EXCLUDED
    def _clear_partition(self, G):
        """
        Removes partition data from the graph
```

```
1057: (8)                    """
1058: (8)                    for u, v, d in G.edges(data=True):
1059: (12)                       if self.partition_key in d:
1060: (16)                           del d[self.partition_key]
1061: (8)                    nx._clear_cache(self.G)
```

File 376 - coding.py:

```
1: (0)              """Functions for encoding and decoding trees.
2: (0)              Since a tree is a highly restricted form of graph, it can be represented
3: (0)              concisely in several ways. This module includes functions for encoding
4: (0)              and decoding trees in the form of nested tuples and Prüfer
5: (0)              sequences. The former requires a rooted tree, whereas the latter can be
6: (0)              applied to unrooted trees. Furthermore, there is a bijection from Prüfer
7: (0)              sequences to labeled trees.
8: (0)              """
9: (0)              from collections import Counter
10: (0)             from itertools import chain
11: (0)             import networkx as nx
12: (0)             from networkx.utils import not_implemented_for
13: (0)             __all__ = [
14: (4)                 "from_nested_tuple",
15: (4)                 "from_prufer_sequence",
16: (4)                 "NotATree",
17: (4)                 "to_nested_tuple",
18: (4)                 "to_prufer_sequence",
19: (0)             ]
20: (0)             class NotATree(nx.NetworkXException):
21: (4)                 """Raised when a function expects a tree (that is, a connected
22: (4)                 undirected graph with no cycles) but gets a non-tree graph as input
23: (4)                 instead.
24: (4)                 """
25: (0)             @not_implemented_for("directed")
26: (0)             @nx._dispatchable(graphs="T")
27: (0)             def to_nested_tuple(T, root, canonical_form=False):
28: (4)                 """Returns a nested tuple representation of the given tree.
29: (4)                 The nested tuple representation of a tree is defined
30: (4)                 recursively. The tree with one node and no edges is represented by
31: (4)                 the empty tuple, ``()``. A tree with ``k`` subtrees is represented
32: (4)                 by a tuple of length ``k`` in which each element is the nested tuple
33: (4)                 representation of a subtree.
34: (4)                 Parameters
35: (4)                 ----------
36: (4)                 T : NetworkX graph
37: (8)                     An undirected graph object representing a tree.
38: (4)                 root : node
39: (8)                     The node in ``T`` to interpret as the root of the tree.
40: (4)                 canonical_form : bool
41: (8)                     If ``True``, each tuple is sorted so that the function returns
42: (8)                     a canonical form for rooted trees. This means "lighter" subtrees
43: (8)                     will appear as nested tuples before "heavier" subtrees. In this
44: (8)                     way, each isomorphic rooted tree has the same nested tuple
45: (8)                     representation.
46: (4)                 Returns
47: (4)                 -------
48: (4)                 tuple
49: (8)                     A nested tuple representation of the tree.
50: (4)                 Notes
51: (4)                 -----
52: (4)                 This function is *not* the inverse of :func:`from_nested_tuple`; the
53: (4)                 only guarantee is that the rooted trees are isomorphic.
54: (4)                 See also
55: (4)                 --------
56: (4)                 from_nested_tuple
57: (4)                 to_prufer_sequence
58: (4)                 Examples
59: (4)                 --------
```

```
(4)                 The tree need not be a balanced binary tree::
(8)                     >>> T = nx.Graph()
(8)                     >>> T.add_edges_from([(0, 1), (0, 2), (0, 3)])
(8)                     >>> T.add_edges_from([(1, 4), (1, 5)])
(8)                     >>> T.add_edges_from([(3, 6), (3, 7)])
(8)                     >>> root = 0
(8)                     >>> nx.to_nested_tuple(T, root)
(8)                     ((((), ()), (), ((), ()))
(4)                 Continuing the above example, if ``canonical_form`` is ``True``, the
(4)                 nested tuples will be sorted::
(8)                     >>> nx.to_nested_tuple(T, root, canonical_form=True)
(8)                     ((), ((), ()), ((), ()))
(4)                 Even the path graph can be interpreted as a tree::
(8)                     >>> T = nx.path_graph(4)
(8)                     >>> root = 0
(8)                     >>> nx.to_nested_tuple(T, root)
(8)                     ((((),),),)
(4)                 """
(4)                 def _make_tuple(T, root, _parent):
(8)                     """Recursively compute the nested tuple representation of the
(8)                     given rooted tree.
(8)                     ``_parent`` is the parent node of ``root`` in the supertree in
(8)                     which ``T`` is a subtree, or ``None`` if ``root`` is the root of
(8)                     the supertree. This argument is used to determine which
(8)                     neighbors of ``root`` are children and which is the parent.
(8)                     """
(8)                     children = set(T[root]) - {_parent}
(8)                     if len(children) == 0:
(12)                        return ()
(8)                     nested = (_make_tuple(T, v, root) for v in children)
(8)                     if canonical_form:
(12)                        nested = sorted(nested)
(8)                     return tuple(nested)
(4)                 if not nx.is_tree(T):
(8)                     raise nx.NotATree("provided graph is not a tree")
(4)                 if root not in T:
(8)                     raise nx.NodeNotFound(f"Graph {T} contains no node {root}")
(4)                 return _make_tuple(T, root, None)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def from_nested_tuple(sequence, sensible_relabeling=False):
(4)                 """Returns the rooted tree corresponding to the given nested tuple.
(4)                 The nested tuple representation of a tree is defined
(4)                 recursively. The tree with one node and no edges is represented by
(4)                 the empty tuple, ``()``. A tree with ``k`` subtrees is represented
(4)                 by a tuple of length ``k`` in which each element is the nested tuple
(4)                 representation of a subtree.
(4)                 Parameters
(4)                 ----------
(4)                 sequence : tuple
(8)                     A nested tuple representing a rooted tree.
(4)                 sensible_relabeling : bool
(8)                     Whether to relabel the nodes of the tree so that nodes are
(8)                     labeled in increasing order according to their breadth-first
(8)                     search order from the root node.
(4)                 Returns
(4)                 -------
(4)                 NetworkX graph
(8)                     The tree corresponding to the given nested tuple, whose root
(8)                     node is node 0. If ``sensible_labeling`` is ``True``, nodes will
(8)                     be labeled in breadth-first search order starting from the root
(8)                     node.
(4)                 Notes
(4)                 -----
(4)                 This function is *not* the inverse of :func:`to_nested_tuple`; the
(4)                 only guarantee is that the rooted trees are isomorphic.
(4)                 See also
(4)                 --------
(4)                 to_nested_tuple
(4)                 from_prufer_sequence
```

```
    Examples
    --------
    Sensible relabeling ensures that the nodes are labeled from the root
    starting at 0::
        >>> balanced = ((((), ()), ((), ()))
        >>> T = nx.from_nested_tuple(balanced, sensible_relabeling=True)
        >>> edges = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
        >>> all((u, v) in T.edges() or (v, u) in T.edges() for (u, v) in
edges)
        True
    """
    def _make_tree(sequence):
        """Recursively creates a tree from the given sequence of nested
        tuples.
        This function employs the :func:`~networkx.tree.join` function
        to recursively join subtrees into a larger tree.
        """
        if len(sequence) == 0:
            return nx.empty_graph(1)
        return nx.tree.join_trees([(_make_tree(child), 0) for child in
sequence])
    T = _make_tree(sequence)
    if sensible_relabeling:
        bfs_nodes = chain([0], (v for u, v in nx.bfs_edges(T, 0)))
        labels = {v: i for i, v in enumerate(bfs_nodes)}
        T = nx.relabel_nodes(T, labels)
    return T
@not_implemented_for("directed")
@nx._dispatchable(graphs="T")
def to_prufer_sequence(T):
    r"""Returns the Prüfer sequence of the given tree.
    A *Prüfer sequence* is a list of *n* - 2 numbers between 0 and
    *n* - 1, inclusive. The tree corresponding to a given Prüfer
    sequence can be recovered by repeatedly joining a node in the
    sequence with a node with the smallest potential degree according to
    the sequence.
    Parameters
    ----------
    T : NetworkX graph
        An undirected graph object representing a tree.
    Returns
    -------
    list
        The Prüfer sequence of the given tree.
    Raises
    ------
    NetworkXPointlessConcept
        If the number of nodes in `T` is less than two.
    NotATree
        If `T` is not a tree.
    KeyError
        If the set of nodes in `T` is not {0, …, *n* - 1}.
    Notes
    -----
    There is a bijection from labeled trees to Prüfer sequences. This
    function is the inverse of the :func:`from_prufer_sequence`
    function.
    Sometimes Prüfer sequences use nodes labeled from 1 to *n* instead
    of from 0 to *n* - 1. This function requires nodes to be labeled in
    the latter form. You can use :func:`~networkx.relabel_nodes` to
    relabel the nodes of your tree to the appropriate format.
    This implementation is from [1]_ and has a running time of
    $O(n)$.
    See also
    --------
    to_nested_tuple
    from_prufer_sequence
    References
    ----------
```

```
(4)                 .. [1] Wang, Xiaodong, Lei Wang, and Yingjie Wu.
(11)                       "An optimal algorithm for Prufer codes."
(11)                       *Journal of Software Engineering and Applications* 2.02 (2009):
111.
(11)                       <https://doi.org/10.4236/jsea.2009.22016>
(4)                 Examples
(4)                 --------
(4)                 There is a bijection between Prüfer sequences and labeled trees, so
(4)                 this function is the inverse of the :func:`from_prufer_sequence`
(4)                 function:
(4)                 >>> edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
(4)                 >>> tree = nx.Graph(edges)
(4)                 >>> sequence = nx.to_prufer_sequence(tree)
(4)                 >>> sequence
(4)                 [3, 3, 3, 4]
(4)                 >>> tree2 = nx.from_prufer_sequence(sequence)
(4)                 >>> list(tree2.edges()) == edges
(4)                 True
(4)                 """
(4)                 n = len(T)
(4)                 if n < 2:
(8)                     msg = "Prüfer sequence undefined for trees with fewer than two nodes"
(8)                     raise nx.NetworkXPointlessConcept(msg)
(4)                 if not nx.is_tree(T):
(8)                     raise nx.NotATree("provided graph is not a tree")
(4)                 if set(T) != set(range(n)):
(8)                     raise KeyError("tree must have node labels {0, ..., n - 1}")
(4)                 degree = dict(T.degree())
(4)                 def parents(u):
(8)                     return next(v for v in T[u] if degree[v] > 1)
(4)                 index = u = next(k for k in range(n) if degree[k] == 1)
(4)                 result = []
(4)                 for i in range(n - 2):
(8)                     v = parents(u)
(8)                     result.append(v)
(8)                     degree[v] -= 1
(8)                     if v < index and degree[v] == 1:
(12)                        u = v
(8)                     else:
(12)                        index = u = next(k for k in range(index + 1, n) if degree[k] == 1)
(4)                 return result
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def from_prufer_sequence(sequence):
(4)                 r"""Returns the tree corresponding to the given Prüfer sequence.
(4)                 A *Prüfer sequence* is a list of *n* - 2 numbers between 0 and
(4)                 *n* - 1, inclusive. The tree corresponding to a given Prüfer
(4)                 sequence can be recovered by repeatedly joining a node in the
(4)                 sequence with a node with the smallest potential degree according to
(4)                 the sequence.
(4)                 Parameters
(4)                 ----------
(4)                 sequence : list
(8)                     A Prüfer sequence, which is a list of *n* - 2 integers between
(8)                     zero and *n* - 1, inclusive.
(4)                 Returns
(4)                 -------
(4)                 NetworkX graph
(8)                     The tree corresponding to the given Prüfer sequence.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If the Prüfer sequence is not valid.
(4)                 Notes
(4)                 -----
(4)                 There is a bijection from labeled trees to Prüfer sequences. This
(4)                 function is the inverse of the :func:`from_prufer_sequence` function.
(4)                 Sometimes Prüfer sequences use nodes labeled from 1 to *n* instead
(4)                 of from 0 to *n* - 1. This function requires nodes to be labeled in
(4)                 the latter form. You can use :func:`networkx.relabel_nodes` to
```

```
264: (4)                relabel the nodes of your tree to the appropriate format.
265: (4)                This implementation is from [1]_ and has a running time of
266: (4)                $O(n)$.
267: (4)                References
268: (4)                ----------
269: (4)                .. [1] Wang, Xiaodong, Lei Wang, and Yingjie Wu.
270: (11)                      "An optimal algorithm for Prufer codes."
271: (11)                      *Journal of Software Engineering and Applications* 2.02 (2009):
111.
272: (11)                      <https://doi.org/10.4236/jsea.2009.22016>
273: (4)                See also
274: (4)                --------
275: (4)                from_nested_tuple
276: (4)                to_prufer_sequence
277: (4)                Examples
278: (4)                --------
279: (4)                There is a bijection between Prüfer sequences and labeled trees, so
280: (4)                this function is the inverse of the :func:`to_prufer_sequence`
281: (4)                function:
282: (4)                >>> edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
283: (4)                >>> tree = nx.Graph(edges)
284: (4)                >>> sequence = nx.to_prufer_sequence(tree)
285: (4)                >>> sequence
286: (4)                [3, 3, 3, 4]
287: (4)                >>> tree2 = nx.from_prufer_sequence(sequence)
288: (4)                >>> list(tree2.edges()) == edges
289: (4)                True
290: (4)                """
291: (4)                n = len(sequence) + 2
292: (4)                degree = Counter(chain(sequence, range(n)))
293: (4)                T = nx.empty_graph(n)
294: (4)                not_orphaned = set()
295: (4)                index = u = next(k for k in range(n) if degree[k] == 1)
296: (4)                for v in sequence:
297: (8)                    if v < 0 or v > n - 1:
298: (12)                       raise nx.NetworkXError(
299: (16)                           f"Invalid Prufer sequence: Values must be between 0 and {n-1},
got {v}"
300: (12)                       )
301: (8)                    T.add_edge(u, v)
302: (8)                    not_orphaned.add(u)
303: (8)                    degree[v] -= 1
304: (8)                    if v < index and degree[v] == 1:
305: (12)                       u = v
306: (8)                    else:
307: (12)                       index = u = next(k for k in range(index + 1, n) if degree[k] == 1)
308: (4)                orphans = set(T) - not_orphaned
309: (4)                u, v = orphans
310: (4)                T.add_edge(u, v)
311: (4)                return T

----------------------------------------

File 377 - decomposition.py:

1: (0)              r"""Function for computing a junction tree of a graph."""
2: (0)              from itertools import combinations
3: (0)              import networkx as nx
4: (0)              from networkx.algorithms import chordal_graph_cliques,
complete_to_chordal_graph, moral
5: (0)              from networkx.utils import not_implemented_for
6: (0)              __all__ = ["junction_tree"]
7: (0)              @not_implemented_for("multigraph")
8: (0)              @nx._dispatchable(returns_graph=True)
9: (0)              def junction_tree(G):
10: (4)                 r"""Returns a junction tree of a given graph.
11: (4)                 A junction tree (or clique tree) is constructed from a (un)directed graph
G.
12: (4)                 The tree is constructed based on a moralized and triangulated version of
```

```
G.
13: (4)                 The tree's nodes consist of maximal cliques and sepsets of the revised
graph.
14: (4)                 The sepset of two cliques is the intersection of the nodes of these
cliques,
15: (4)                 e.g. the sepset of (A,B,C) and (A,C,E,F) is (A,C). These nodes are often
called
16: (4)                 "variables" in this literature. The tree is bipartite with each sepset
17: (4)                 connected to its two cliques.
18: (4)                 Junction Trees are not unique as the order of clique consideration
determines
19: (4)                 which sepsets are included.
20: (4)                 The junction tree algorithm consists of five steps [1]_:
21: (4)                 1. Moralize the graph
22: (4)                 2. Triangulate the graph
23: (4)                 3. Find maximal cliques
24: (4)                 4. Build the tree from cliques, connecting cliques with shared
25: (7)                    nodes, set edge-weight to number of shared variables
26: (4)                 5. Find maximum spanning tree
27: (4)                 Parameters
28: (4)                 ----------
29: (4)                 G : networkx.Graph
30: (8)                     Directed or undirected graph.
31: (4)                 Returns
32: (4)                 -------
33: (4)                 junction_tree : networkx.Graph
34: (8)                     The corresponding junction tree of `G`.
35: (4)                 Raises
36: (4)                 ------
37: (4)                 NetworkXNotImplemented
38: (8)                     Raised if `G` is an instance of `MultiGraph` or `MultiDiGraph`.
39: (4)                 References
40: (4)                 ----------
41: (4)                 .. [1] Junction tree algorithm:
42: (7)                    https://en.wikipedia.org/wiki/Junction_tree_algorithm
43: (4)                 .. [2] Finn V. Jensen and Frank Jensen. 1994. Optimal
44: (7)                    junction trees. In Proceedings of the Tenth international
45: (7)                    conference on Uncertainty in artificial intelligence (UAI’94).
46: (7)                    Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 360–366.
47: (4)                 """
48: (4)                 clique_graph = nx.Graph()
49: (4)                 if G.is_directed():
50: (8)                     G = moral.moral_graph(G)
51: (4)                 chordal_graph, _ = complete_to_chordal_graph(G)
52: (4)                 cliques = [tuple(sorted(i)) for i in chordal_graph_cliques(chordal_graph)]
53: (4)                 clique_graph.add_nodes_from(cliques, type="clique")
54: (4)                 for edge in combinations(cliques, 2):
55: (8)                     set_edge_0 = set(edge[0])
56: (8)                     set_edge_1 = set(edge[1])
57: (8)                     if not set_edge_0.isdisjoint(set_edge_1):
58: (12)                        sepset = tuple(sorted(set_edge_0.intersection(set_edge_1)))
59: (12)                        clique_graph.add_edge(edge[0], edge[1], weight=len(sepset),
sepset=sepset)
60: (4)                 junction_tree = nx.maximum_spanning_tree(clique_graph)
61: (4)                 for edge in list(junction_tree.edges(data=True)):
62: (8)                     junction_tree.add_node(edge[2]["sepset"], type="sepset")
63: (8)                     junction_tree.add_edge(edge[0], edge[2]["sepset"])
64: (8)                     junction_tree.add_edge(edge[1], edge[2]["sepset"])
65: (8)                     junction_tree.remove_edge(edge[0], edge[1])
66: (4)                 return junction_tree

----------------------------------------

File 378 - mst.py:

1: (0)              """
2: (0)              Algorithms for calculating min/max spanning trees/forests.
3: (0)              """
4: (0)              from dataclasses import dataclass, field
```

```
5: (0)              from enum import Enum
6: (0)              from heapq import heappop, heappush
7: (0)              from itertools import count
8: (0)              from math import isnan
9: (0)              from operator import itemgetter
10: (0)             from queue import PriorityQueue
11: (0)             import networkx as nx
12: (0)             from networkx.utils import UnionFind, not_implemented_for, py_random_state
13: (0)             __all__ = [
14: (4)                 "minimum_spanning_edges",
15: (4)                 "maximum_spanning_edges",
16: (4)                 "minimum_spanning_tree",
17: (4)                 "maximum_spanning_tree",
18: (4)                 "number_of_spanning_trees",
19: (4)                 "random_spanning_tree",
20: (4)                 "partition_spanning_tree",
21: (4)                 "EdgePartition",
22: (4)                 "SpanningTreeIterator",
23: (0)             ]
24: (0)             class EdgePartition(Enum):
25: (4)                 """
26: (4)                 An enum to store the state of an edge partition. The enum is written to
the
27: (4)                 edges of a graph before being pasted to `kruskal_mst_edges`. Options are:
28: (4)                 - EdgePartition.OPEN
29: (4)                 - EdgePartition.INCLUDED
30: (4)                 - EdgePartition.EXCLUDED
31: (4)                 """
32: (4)                 OPEN = 0
33: (4)                 INCLUDED = 1
34: (4)                 EXCLUDED = 2
35: (0)             @not_implemented_for("multigraph")
36: (0)             @nx._dispatchable(edge_attrs="weight", preserve_edge_attrs="data")
37: (0)             def boruvka_mst_edges(
38: (4)                 G, minimum=True, weight="weight", keys=False, data=True, ignore_nan=False
39: (0)             ):
40: (4)                 """Iterate over edges of a Borůvka's algorithm min/max spanning tree.
41: (4)                 Parameters
42: (4)                 ----------
43: (4)                 G : NetworkX Graph
44: (8)                     The edges of `G` must have distinct weights,
45: (8)                     otherwise the edges may not form a tree.
46: (4)                 minimum : bool (default: True)
47: (8)                     Find the minimum (True) or maximum (False) spanning tree.
48: (4)                 weight : string (default: 'weight')
49: (8)                     The name of the edge attribute holding the edge weights.
50: (4)                 keys : bool (default: True)
51: (8)                     This argument is ignored since this function is not
52: (8)                     implemented for multigraphs; it exists only for consistency
53: (8)                     with the other minimum spanning tree functions.
54: (4)                 data : bool (default: True)
55: (8)                     Flag for whether to yield edge attribute dicts.
56: (8)                     If True, yield edges `(u, v, d)`, where `d` is the attribute dict.
57: (8)                     If False, yield edges `(u, v)`.
58: (4)                 ignore_nan : bool (default: False)
59: (8)                     If a NaN is found as an edge weight normally an exception is raised.
60: (8)                     If `ignore_nan is True` then that edge is ignored instead.
61: (4)                 """
62: (4)                 forest = UnionFind(G)
63: (4)                 def best_edge(component):
64: (8)                     """Returns the optimum (minimum or maximum) edge on the edge
65: (8)                     boundary of the given set of nodes.
66: (8)                     A return value of ``None`` indicates an empty boundary.
67: (8)                     """
68: (8)                     sign = 1 if minimum else -1
69: (8)                     minwt = float("inf")
70: (8)                     boundary = None
71: (8)                     for e in nx.edge_boundary(G, component, data=True):
72: (12)                        wt = e[-1].get(weight, 1) * sign
```

```
 73: (12)                        if isnan(wt):
 74: (16)                            if ignore_nan:
 75: (20)                                continue
 76: (16)                            msg = f"NaN found as an edge weight. Edge {e}"
 77: (16)                            raise ValueError(msg)
 78: (12)                        if wt < minwt:
 79: (16)                            minwt = wt
 80: (16)                            boundary = e
 81: (8)                     return boundary
 82: (4)                 best_edges = (best_edge(component) for component in forest.to_sets())
 83: (4)                 best_edges = [edge for edge in best_edges if edge is not None]
 84: (4)                 while best_edges:
 85: (8)                     best_edges = (best_edge(component) for component in forest.to_sets())
 86: (8)                     best_edges = [edge for edge in best_edges if edge is not None]
 87: (8)                     for u, v, d in best_edges:
 88: (12)                        if forest[u] != forest[v]:
 89: (16)                            if data:
 90: (20)                                yield u, v, d
 91: (16)                            else:
 92: (20)                                yield u, v
 93: (16)                            forest.union(u, v)
 94: (0)             @nx._dispatchable(
 95: (4)                 edge_attrs={"weight": None, "partition": None}, preserve_edge_attrs="data"
 96: (0)             )
 97: (0)             def kruskal_mst_edges(
 98: (4)                 G, minimum, weight="weight", keys=True, data=True, ignore_nan=False,
partition=None
 99: (0)             ):
100: (4)                 """
101: (4)                 Iterate over edge of a Kruskal's algorithm min/max spanning tree.
102: (4)                 Parameters
103: (4)                 ----------
104: (4)                 G : NetworkX Graph
105: (8)                     The graph holding the tree of interest.
106: (4)                 minimum : bool (default: True)
107: (8)                     Find the minimum (True) or maximum (False) spanning tree.
108: (4)                 weight : string (default: 'weight')
109: (8)                     The name of the edge attribute holding the edge weights.
110: (4)                 keys : bool (default: True)
111: (8)                     If `G` is a multigraph, `keys` controls whether edge keys ar yielded.
112: (8)                     Otherwise `keys` is ignored.
113: (4)                 data : bool (default: True)
114: (8)                     Flag for whether to yield edge attribute dicts.
115: (8)                     If True, yield edges `(u, v, d)`, where `d` is the attribute dict.
116: (8)                     If False, yield edges `(u, v)`.
117: (4)                 ignore_nan : bool (default: False)
118: (8)                     If a NaN is found as an edge weight normally an exception is raised.
119: (8)                     If `ignore_nan is True` then that edge is ignored instead.
120: (4)                 partition : string (default: None)
121: (8)                     The name of the edge attribute holding the partition data, if it
exists.
122: (8)                     Partition data is written to the edges using the `EdgePartition` enum.
123: (8)                     If a partition exists, all included edges and none of the excluded
edges
124: (8)                     will appear in the final tree. Open edges may or may not be used.
125: (4)                 Yields
126: (4)                 ------
127: (4)                 edge tuple
128: (8)                     The edges as discovered by Kruskal's method. Each edge can
129: (8)                     take the following forms: `(u, v)`, `(u, v, d)` or `(u, v, k, d)`
130: (8)                     depending on the `key` and `data` parameters
131: (4)                 """
132: (4)                 subtrees = UnionFind()
133: (4)                 if G.is_multigraph():
134: (8)                     edges = G.edges(keys=True, data=True)
135: (4)                 else:
136: (8)                     edges = G.edges(data=True)
137: (4)                 """
138: (4)                 Sort the edges of the graph with respect to the partition data.
```

```
    Edges are returned in the following order:
    * Included edges
    * Open edges from smallest to largest weight
    * Excluded edges
    """
    included_edges = []
    open_edges = []
    for e in edges:
        d = e[-1]
        wt = d.get(weight, 1)
        if isnan(wt):
            if ignore_nan:
                continue
            raise ValueError(f"NaN found as an edge weight. Edge {e}")
        edge = (wt,) + e
        if d.get(partition) == EdgePartition.INCLUDED:
            included_edges.append(edge)
        elif d.get(partition) == EdgePartition.EXCLUDED:
            continue
        else:
            open_edges.append(edge)
    if minimum:
        sorted_open_edges = sorted(open_edges, key=itemgetter(0))
    else:
        sorted_open_edges = sorted(open_edges, key=itemgetter(0), reverse=True)
    included_edges.extend(sorted_open_edges)
    sorted_edges = included_edges
    del open_edges, sorted_open_edges, included_edges
    if G.is_multigraph():
        for wt, u, v, k, d in sorted_edges:
            if subtrees[u] != subtrees[v]:
                if keys:
                    if data:
                        yield u, v, k, d
                    else:
                        yield u, v, k
                else:
                    if data:
                        yield u, v, d
                    else:
                        yield u, v
                subtrees.union(u, v)
    else:
        for wt, u, v, d in sorted_edges:
            if subtrees[u] != subtrees[v]:
                if data:
                    yield u, v, d
                else:
                    yield u, v
                subtrees.union(u, v)
@nx._dispatchable(edge_attrs="weight", preserve_edge_attrs="data")
def prim_mst_edges(G, minimum, weight="weight", keys=True, data=True, ignore_nan=False):
    """Iterate over edges of Prim's algorithm min/max spanning tree.
    Parameters
    ----------
    G : NetworkX Graph
        The graph holding the tree of interest.
    minimum : bool (default: True)
        Find the minimum (True) or maximum (False) spanning tree.
    weight : string (default: 'weight')
        The name of the edge attribute holding the edge weights.
    keys : bool (default: True)
        If `G` is a multigraph, `keys` controls whether edge keys ar yielded.
        Otherwise `keys` is ignored.
    data : bool (default: True)
        Flag for whether to yield edge attribute dicts.
        If True, yield edges `(u, v, d)`, where `d` is the attribute dict.
```

```
(8)                 If False, yield edges `(u, v)`.
(4)             ignore_nan : bool (default: False)
(8)                 If a NaN is found as an edge weight normally an exception is raised.
(8)                 If `ignore_nan is True` then that edge is ignored instead.
(4)             """
(4)             is_multigraph = G.is_multigraph()
(4)             push = heappush
(4)             pop = heappop
(4)             nodes = set(G)
(4)             c = count()
(4)             sign = 1 if minimum else -1
(4)             while nodes:
(8)                 u = nodes.pop()
(8)                 frontier = []
(8)                 visited = {u}
(8)                 if is_multigraph:
(12)                    for v, keydict in G.adj[u].items():
(16)                        for k, d in keydict.items():
(20)                            wt = d.get(weight, 1) * sign
(20)                            if isnan(wt):
(24)                                if ignore_nan:
(28)                                    continue
(24)                                msg = f"NaN found as an edge weight. Edge {(u, v, k, d)}"
(24)                                raise ValueError(msg)
(20)                            push(frontier, (wt, next(c), u, v, k, d))
(8)                 else:
(12)                    for v, d in G.adj[u].items():
(16)                        wt = d.get(weight, 1) * sign
(16)                        if isnan(wt):
(20)                            if ignore_nan:
(24)                                continue
(20)                            msg = f"NaN found as an edge weight. Edge {(u, v, d)}"
(20)                            raise ValueError(msg)
(16)                        push(frontier, (wt, next(c), u, v, d))
(8)                 while nodes and frontier:
(12)                    if is_multigraph:
(16)                        W, _, u, v, k, d = pop(frontier)
(12)                    else:
(16)                        W, _, u, v, d = pop(frontier)
(12)                    if v in visited or v not in nodes:
(16)                        continue
(12)                    if is_multigraph and keys:
(16)                        if data:
(20)                            yield u, v, k, d
(16)                        else:
(20)                            yield u, v, k
(12)                    else:
(16)                        if data:
(20)                            yield u, v, d
(16)                        else:
(20)                            yield u, v
(12)                    visited.add(v)
(12)                    nodes.discard(v)
(12)                    if is_multigraph:
(16)                        for w, keydict in G.adj[v].items():
(20)                            if w in visited:
(24)                                continue
(20)                            for k2, d2 in keydict.items():
(24)                                new_weight = d2.get(weight, 1) * sign
(24)                                if isnan(new_weight):
(28)                                    if ignore_nan:
(32)                                        continue
(28)                                    msg = f"NaN found as an edge weight. Edge {(v, w, k2, d2)}"
(28)                                    raise ValueError(msg)
(24)                                push(frontier, (new_weight, next(c), v, w, k2, d2))
(12)                    else:
(16)                        for w, d2 in G.adj[v].items():
```

```
(20)                                if w in visited:
(24)                                    continue
(20)                                new_weight = d2.get(weight, 1) * sign
(20)                                if isnan(new_weight):
(24)                                    if ignore_nan:
(28)                                        continue
(24)                                    msg = f"NaN found as an edge weight. Edge {(v, w, d2)}"
(24)                                    raise ValueError(msg)
(20)                                push(frontier, (new_weight, next(c), v, w, d2))
(0)             ALGORITHMS = {
(4)                 "boruvka": boruvka_mst_edges,
(4)                 "borůvka": boruvka_mst_edges,
(4)                 "kruskal": kruskal_mst_edges,
(4)                 "prim": prim_mst_edges,
(0)             }
(0)             @not_implemented_for("directed")
(0)             @nx._dispatchable(edge_attrs="weight", preserve_edge_attrs="data")
(0)             def minimum_spanning_edges(
(4)                 G, algorithm="kruskal", weight="weight", keys=True, data=True, ignore_nan=False
(0)             ):
(4)                 """Generate edges in a minimum spanning forest of an undirected
(4)                 weighted graph.
(4)                 A minimum spanning tree is a subgraph of the graph (a tree)
(4)                 with the minimum sum of edge weights.  A spanning forest is a
(4)                 union of the spanning trees for each connected component of the graph.
(4)                 Parameters
(4)                 ----------
(4)                 G : undirected Graph
(7)                    An undirected graph. If `G` is connected, then the algorithm finds a
(7)                    spanning tree. Otherwise, a spanning forest is found.
(4)                 algorithm : string
(7)                    The algorithm to use when finding a minimum spanning tree. Valid
(7)                    choices are 'kruskal', 'prim', or 'boruvka'. The default is 'kruskal'.
(4)                 weight : string
(7)                    Edge data key to use for weight (default 'weight').
(4)                 keys : bool
(7)                    Whether to yield edge key in multigraphs in addition to the edge.
(7)                    If `G` is not a multigraph, this is ignored.
(4)                 data : bool, optional
(7)                    If True yield the edge data along with the edge.
(4)                 ignore_nan : bool (default: False)
(8)                     If a NaN is found as an edge weight normally an exception is raised.
(8)                     If `ignore_nan is True` then that edge is ignored instead.
(4)                 Returns
(4)                 -------
(4)                 edges : iterator
(7)                    An iterator over edges in a maximum spanning tree of `G`.
(7)                    Edges connecting nodes `u` and `v` are represented as tuples:
(7)                    `(u, v, k, d)` or `(u, v, k)` or `(u, v, d)` or `(u, v)`
(7)                    If `G` is a multigraph, `keys` indicates whether the edge key `k` will
(7)                    be reported in the third position in the edge tuple. `data` indicates
(7)                    whether the edge datadict `d` will appear at the end of the edge tuple.
(7)                    If `G` is not a multigraph, the tuples are `(u, v, d)` if `data` is True
(7)                    or `(u, v)` if `data` is False.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.algorithms import tree
(4)                 Find minimum spanning edges by Kruskal's algorithm
(4)                 >>> G = nx.cycle_graph(4)
(4)                 >>> G.add_edge(0, 3, weight=2)
(4)                 >>> mst = tree.minimum_spanning_edges(G, algorithm="kruskal", data=False)
(4)                 >>> edgelist = list(mst)
(4)                 >>> sorted(sorted(e) for e in edgelist)
(4)                 [[0, 1], [1, 2], [2, 3]]
(4)                 Find minimum spanning edges by Prim's algorithm
(4)                 >>> G = nx.cycle_graph(4)
```

```
339: (4)                >>> G.add_edge(0, 3, weight=2)
340: (4)                >>> mst = tree.minimum_spanning_edges(G, algorithm="prim", data=False)
341: (4)                >>> edgelist = list(mst)
342: (4)                >>> sorted(sorted(e) for e in edgelist)
343: (4)                [[0, 1], [1, 2], [2, 3]]
344: (4)                Notes
345: (4)                -----
346: (4)                For Borůvka's algorithm, each edge must have a weight attribute, and
347: (4)                each edge weight must be distinct.
348: (4)                For the other algorithms, if the graph edges do not have a weight
349: (4)                attribute a default weight of 1 will be used.
350: (4)                Modified code from David Eppstein, April 2006
351: (4)                http://www.ics.uci.edu/~eppstein/PADS/
352: (4)                """
353: (4)                try:
354: (8)                    algo = ALGORITHMS[algorithm]
355: (4)                except KeyError as err:
356: (8)                    msg = f"{algorithm} is not a valid choice for an algorithm."
357: (8)                    raise ValueError(msg) from err
358: (4)                return algo(
359: (8)                    G, minimum=True, weight=weight, keys=keys, data=data,
ignore_nan=ignore_nan
360: (4)                )
361: (0)            @not_implemented_for("directed")
362: (0)            @nx._dispatchable(edge_attrs="weight", preserve_edge_attrs="data")
363: (0)            def maximum_spanning_edges(
364: (4)                G, algorithm="kruskal", weight="weight", keys=True, data=True,
ignore_nan=False
365: (0)            ):
366: (4)                """Generate edges in a maximum spanning forest of an undirected
367: (4)                weighted graph.
368: (4)                A maximum spanning tree is a subgraph of the graph (a tree)
369: (4)                with the maximum possible sum of edge weights.  A spanning forest is a
370: (4)                union of the spanning trees for each connected component of the graph.
371: (4)                Parameters
372: (4)                ----------
373: (4)                G : undirected Graph
374: (7)                   An undirected graph. If `G` is connected, then the algorithm finds a
375: (7)                   spanning tree. Otherwise, a spanning forest is found.
376: (4)                algorithm : string
377: (7)                   The algorithm to use when finding a maximum spanning tree. Valid
378: (7)                   choices are 'kruskal', 'prim', or 'boruvka'. The default is 'kruskal'.
379: (4)                weight : string
380: (7)                   Edge data key to use for weight (default 'weight').
381: (4)                keys : bool
382: (7)                   Whether to yield edge key in multigraphs in addition to the edge.
383: (7)                   If `G` is not a multigraph, this is ignored.
384: (4)                data : bool, optional
385: (7)                   If True yield the edge data along with the edge.
386: (4)                ignore_nan : bool (default: False)
387: (8)                    If a NaN is found as an edge weight normally an exception is raised.
388: (8)                    If `ignore_nan is True` then that edge is ignored instead.
389: (4)                Returns
390: (4)                -------
391: (4)                edges : iterator
392: (7)                   An iterator over edges in a maximum spanning tree of `G`.
393: (7)                   Edges connecting nodes `u` and `v` are represented as tuples:
394: (7)                   `(u, v, k, d)` or `(u, v, k)` or `(u, v, d)` or `(u, v)`
395: (7)                   If `G` is a multigraph, `keys` indicates whether the edge key `k` will
396: (7)                   be reported in the third position in the edge tuple. `data` indicates
397: (7)                   whether the edge datadict `d` will appear at the end of the edge tuple.
398: (7)                   If `G` is not a multigraph, the tuples are `(u, v, d)` if `data` is
True
399: (7)                   or `(u, v)` if `data` is False.
400: (4)                Examples
401: (4)                --------
402: (4)                >>> from networkx.algorithms import tree
403: (4)                Find maximum spanning edges by Kruskal's algorithm
404: (4)                >>> G = nx.cycle_graph(4)
```

```
(4)                 >>> G.add_edge(0, 3, weight=2)
(4)                 >>> mst = tree.maximum_spanning_edges(G, algorithm="kruskal", data=False)
(4)                 >>> edgelist = list(mst)
(4)                 >>> sorted(sorted(e) for e in edgelist)
(4)                 [[0, 1], [0, 3], [1, 2]]
(4)                 Find maximum spanning edges by Prim's algorithm
(4)                 >>> G = nx.cycle_graph(4)
(4)                 >>> G.add_edge(0, 3, weight=2)  # assign weight 2 to edge 0-3
(4)                 >>> mst = tree.maximum_spanning_edges(G, algorithm="prim", data=False)
(4)                 >>> edgelist = list(mst)
(4)                 >>> sorted(sorted(e) for e in edgelist)
(4)                 [[0, 1], [0, 3], [2, 3]]
(4)                 Notes
(4)                 -----
(4)                 For Borůvka's algorithm, each edge must have a weight attribute, and
(4)                 each edge weight must be distinct.
(4)                 For the other algorithms, if the graph edges do not have a weight
(4)                 attribute a default weight of 1 will be used.
(4)                 Modified code from David Eppstein, April 2006
(4)                 http://www.ics.uci.edu/~eppstein/PADS/
(4)                 """
(4)                 try:
(8)                     algo = ALGORITHMS[algorithm]
(4)                 except KeyError as err:
(8)                     msg = f"{algorithm} is not a valid choice for an algorithm."
(8)                     raise ValueError(msg) from err
(4)                 return algo(
(8)                     G, minimum=False, weight=weight, keys=keys, data=data,
ignore_nan=ignore_nan
(4)                 )
(0)             @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
(0)             def minimum_spanning_tree(G, weight="weight", algorithm="kruskal",
ignore_nan=False):
(4)                 """Returns a minimum spanning tree or forest on an undirected graph `G`.
(4)                 Parameters
(4)                 ----------
(4)                 G : undirected graph
(8)                     An undirected graph. If `G` is connected, then the algorithm finds a
(8)                     spanning tree. Otherwise, a spanning forest is found.
(4)                 weight : str
(7)                    Data key to use for edge weights.
(4)                 algorithm : string
(7)                    The algorithm to use when finding a minimum spanning tree. Valid
(7)                    choices are 'kruskal', 'prim', or 'boruvka'. The default is
(7)                    'kruskal'.
(4)                 ignore_nan : bool (default: False)
(8)                     If a NaN is found as an edge weight normally an exception is raised.
(8)                     If `ignore_nan is True` then that edge is ignored instead.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX Graph
(7)                    A minimum spanning tree or forest.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.cycle_graph(4)
(4)                 >>> G.add_edge(0, 3, weight=2)
(4)                 >>> T = nx.minimum_spanning_tree(G)
(4)                 >>> sorted(T.edges(data=True))
(4)                 [(0, 1, {}), (1, 2, {}), (2, 3, {})]
(4)                 Notes
(4)                 -----
(4)                 For Borůvka's algorithm, each edge must have a weight attribute, and
(4)                 each edge weight must be distinct.
(4)                 For the other algorithms, if the graph edges do not have a weight
(4)                 attribute a default weight of 1 will be used.
(4)                 There may be more than one tree with the same minimum or maximum weight.
(4)                 See :mod:`networkx.tree.recognition` for more detailed definitions.
(4)                 Isolated nodes with self-loops are in the tree as edgeless isolated nodes.
(4)                 """
```

```
    edges = minimum_spanning_edges(
        G, algorithm, weight, keys=True, data=True, ignore_nan=ignore_nan
    )
    T = G.__class__()  # Same graph class as G
    T.graph.update(G.graph)
    T.add_nodes_from(G.nodes.items())
    T.add_edges_from(edges)
    return T
@nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
def partition_spanning_tree(
    G, minimum=True, weight="weight", partition="partition", ignore_nan=False
):
    """
    Find a spanning tree while respecting a partition of edges.
    Edges can be flagged as either `INCLUDED` which are required to be in the
    returned tree, `EXCLUDED`, which cannot be in the returned tree and `OPEN`.
    This is used in the SpanningTreeIterator to create new partitions following
    the algorithm of Sörensen and Janssens [1]_.
    Parameters
    ----------
    G : undirected graph
        An undirected graph.
    minimum : bool (default: True)
        Determines whether the returned tree is the minimum spanning tree of
        the partition of the maximum one.
    weight : str
        Data key to use for edge weights.
    partition : str
        The key for the edge attribute containing the partition
        data on the graph. Edges can be included, excluded or open using the
        `EdgePartition` enum.
    ignore_nan : bool (default: False)
        If a NaN is found as an edge weight normally an exception is raised.
        If `ignore_nan is True` then that edge is ignored instead.
    Returns
    -------
    G : NetworkX Graph
        A minimum spanning tree using all of the included edges in the graph and
        none of the excluded edges.
    References
    ----------
    .. [1] G.K. Janssens, K. Sörensen, An algorithm to generate all spanning
           trees in order of increasing cost, Pesquisa Operacional, 2005-08,
           Vol. 25 (2), p. 219-229,
           https://www.scielo.br/j/pope/a/XHswBwRwJyrfL88dmMwYNWp/?lang=en
    """
    edges = kruskal_mst_edges(
        G,
        minimum,
        weight,
        keys=True,
        data=True,
        ignore_nan=ignore_nan,
        partition=partition,
    )
    T = G.__class__()  # Same graph class as G
    T.graph.update(G.graph)
    T.add_nodes_from(G.nodes.items())
    T.add_edges_from(edges)
    return T
@nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
def maximum_spanning_tree(G, weight="weight", algorithm="kruskal", ignore_nan=False):
    """Returns a maximum spanning tree or forest on an undirected graph `G`.
    Parameters
    ----------
```

```
    G : undirected graph
        An undirected graph. If `G` is connected, then the algorithm finds a
        spanning tree. Otherwise, a spanning forest is found.
    weight : str
       Data key to use for edge weights.
    algorithm : string
       The algorithm to use when finding a maximum spanning tree. Valid
       choices are 'kruskal', 'prim', or 'boruvka'. The default is
       'kruskal'.
    ignore_nan : bool (default: False)
        If a NaN is found as an edge weight normally an exception is raised.
        If `ignore_nan is True` then that edge is ignored instead.
    Returns
    -------
    G : NetworkX Graph
       A maximum spanning tree or forest.
    Examples
    --------
    >>> G = nx.cycle_graph(4)
    >>> G.add_edge(0, 3, weight=2)
    >>> T = nx.maximum_spanning_tree(G)
    >>> sorted(T.edges(data=True))
    [(0, 1, {}), (0, 3, {'weight': 2}), (1, 2, {})]
    Notes
    -----
    For Borůvka's algorithm, each edge must have a weight attribute, and
    each edge weight must be distinct.
    For the other algorithms, if the graph edges do not have a weight
    attribute a default weight of 1 will be used.
    There may be more than one tree with the same minimum or maximum weight.
    See :mod:`networkx.tree.recognition` for more detailed definitions.
    Isolated nodes with self-loops are in the tree as edgeless isolated nodes.
    """
    edges = maximum_spanning_edges(
        G, algorithm, weight, keys=True, data=True, ignore_nan=ignore_nan
    )
    edges = list(edges)
    T = G.__class__()  # Same graph class as G
    T.graph.update(G.graph)
    T.add_nodes_from(G.nodes.items())
    T.add_edges_from(edges)
    return T
@py_random_state(3)
@nx._dispatchable(preserve_edge_attrs=True, returns_graph=True)
def random_spanning_tree(G, weight=None, *, multiplicative=True, seed=None):
    """
    Sample a random spanning tree using the edges weights of `G`.
    This function supports two different methods for determining the
    probability of the graph. If ``multiplicative=True``, the probability
    is based on the product of edge weights, and if ``multiplicative=False``
    it is based on the sum of the edge weight. However, since it is
    easier to determine the total weight of all spanning trees for the
    multiplicative version, that is significantly faster and should be used if
    possible. Additionally, setting `weight` to `None` will cause a spanning tree

    to be selected with uniform probability.
    The function uses algorithm A8 in [1]_ .
    Parameters
    ----------
    G : nx.Graph
        An undirected version of the original graph.
    weight : string
        The edge key for the edge attribute holding edge weight.
    multiplicative : bool, default=True
        If `True`, the probability of each tree is the product of its edge weight

        over the sum of the product of all the spanning trees in the graph. If
        `False`, the probability is the sum of its edge weight over the sum of
        the sum of weights for all spanning trees in the graph.
```

```
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    Returns
    -------
    nx.Graph
        A spanning tree using the distribution defined by the weight of the tree.
    References
    ----------
    .. [1] V. Kulkarni, Generating random combinatorial objects, Journal of
       Algorithms, 11 (1990), pp. 185–207
    """
    def find_node(merged_nodes, node):
        """
        We can think of clusters of contracted nodes as having one
        representative in the graph. Each node which is not in merged_nodes
        is still its own representative. Since a representative can be later
        contracted, we need to recursively search though the dict to find
        the final representative, but once we know it we can use path
        compression to speed up the access of the representative for next time.
        This cannot be replaced by the standard NetworkX union_find since that
        data structure will merge nodes with less representing nodes into the
        one with more representing nodes but this function requires we merge
        them using the order that contract_edges contracts using.
        Parameters
        ----------
        merged_nodes : dict
            The dict storing the mapping from node to representative
        node
            The node whose representative we seek
        Returns
        -------
        The representative of the `node`
        """
        if node not in merged_nodes:
            return node
        else:
            rep = find_node(merged_nodes, merged_nodes[node])
            merged_nodes[node] = rep
            return rep
    def prepare_graph():
        """
        For the graph `G`, remove all edges not in the set `V` and then
        contract all edges in the set `U`.
        Returns
        -------
        A copy of `G` which has had all edges not in `V` removed and all edges
        in `U` contracted.
        """
        result = nx.MultiGraph(incoming_graph_data=G)
        edges_to_remove = set(result.edges()).difference(V)
        result.remove_edges_from(edges_to_remove)
        merged_nodes = {}
        for u, v in U:
            u_rep = find_node(merged_nodes, u)
            v_rep = find_node(merged_nodes, v)
            if u_rep == v_rep:
                continue
            nx.contracted_nodes(result, u_rep, v_rep, self_loops=False, copy=False)
            merged_nodes[v_rep] = u_rep
        return merged_nodes, result
    def spanning_tree_total_weight(G, weight):
        """
        Find the sum of weights of the spanning trees of `G` using the
        appropriate `method`.
        This is easy if the chosen method is 'multiplicative', since we can
```

```
(8)                     use Kirchhoff's Tree Matrix Theorem directly. However, with the
(8)                     'additive' method, this process is slightly more complex and less
(8)                     computationally efficient as we have to find the number of spanning
(8)                     trees which contain each possible edge in the graph.
(8)                     Parameters
(8)                     ----------
(8)                     G : NetworkX Graph
(12)                        The graph to find the total weight of all spanning trees on.
(8)                     weight : string
(12)                        The key for the weight edge attribute of the graph.
(8)                     Returns
(8)                     -------
(8)                     float
(12)                        The sum of either the multiplicative or additive weight for all
(12)                        spanning trees in the graph.
(8)                     """
(8)                     if multiplicative:
(12)                        return nx.total_spanning_tree_weight(G, weight)
(8)                     else:
(12)                        if G.number_of_edges() == 1:
(16)                            return G.edges(data=weight).__iter__().__next__()[2]
(12)                        else:
(16)                            total = 0
(16)                            for u, v, w in G.edges(data=weight):
(20)                                total += w * nx.total_spanning_tree_weight(
(24)                                    nx.contracted_edge(G, edge=(u, v), self_loops=False),
None
(20)                                )
(16)                            return total
(4)                 if G.number_of_nodes() < 2:
(8)                     return nx.empty_graph(G.nodes)
(4)                 U = set()
(4)                 st_cached_value = 0
(4)                 V = set(G.edges())
(4)                 shuffled_edges = list(G.edges())
(4)                 seed.shuffle(shuffled_edges)
(4)                 for u, v in shuffled_edges:
(8)                     e_weight = G[u][v][weight] if weight is not None else 1
(8)                     node_map, prepared_G = prepare_graph()
(8)                     G_total_tree_weight = spanning_tree_total_weight(prepared_G, weight)
(8)                     rep_edge = (find_node(node_map, u), find_node(node_map, v))
(8)                     if rep_edge in prepared_G.edges:
(12)                        prepared_G_e = nx.contracted_edge(
(16)                            prepared_G, edge=rep_edge, self_loops=False
(12)                        )
(12)                        G_e_total_tree_weight = spanning_tree_total_weight(prepared_G_e,
weight)
(12)                        if multiplicative:
(16)                            threshold = e_weight * G_e_total_tree_weight /
G_total_tree_weight
(12)                        else:
(16)                            numerator = (
(20)                                st_cached_value + e_weight
(16)                            ) * nx.total_spanning_tree_weight(prepared_G_e) +
G_e_total_tree_weight
(16)                            denominator = (
(20)                                st_cached_value *
nx.total_spanning_tree_weight(prepared_G)
(20)                                + G_total_tree_weight
(16)                            )
(16)                            threshold = numerator / denominator
(8)                     else:
(12)                        threshold = 0.0
(8)                     z = seed.uniform(0.0, 1.0)
(8)                     if z > threshold:
(12)                        V.remove((u, v))
(8)                     else:
(12)                        st_cached_value += e_weight
(12)                        U.add((u, v))
```

```
734: (8)                    if len(U) == G.number_of_nodes() - 1:
735: (12)                       spanning_tree = nx.Graph()
736: (12)                       spanning_tree.add_edges_from(U)
737: (12)                       return spanning_tree
738: (4)                raise Exception(f"Something went wrong! Only {len(U)} edges in the
spanning tree!")
739: (0)            class SpanningTreeIterator:
740: (4)                """
741: (4)                Iterate over all spanning trees of a graph in either increasing or
742: (4)                decreasing cost.
743: (4)                Notes
744: (4)                -----
745: (4)                This iterator uses the partition scheme from [1]_ (included edges,
746: (4)                excluded edges and open edges) as well as a modified Kruskal's Algorithm
747: (4)                to generate minimum spanning trees which respect the partition of edges.
748: (4)                For spanning trees with the same weight, ties are broken arbitrarily.
749: (4)                References
750: (4)                ----------
751: (4)                .. [1] G.K. Janssens, K. Sörensen, An algorithm to generate all spanning
752: (11)                      trees in order of increasing cost, Pesquisa Operacional, 2005-08,
753: (11)                      Vol. 25 (2), p. 219-229,
754: (11)                      https://www.scielo.br/j/pope/a/XHswBwRwJyrfL88dmMwYNWp/?lang=en
755: (4)                """
756: (4)                @dataclass(order=True)
757: (4)                class Partition:
758: (8)                    """
759: (8)                    This dataclass represents a partition and stores a dict with the edge
760: (8)                    data and the weight of the minimum spanning tree of the partition
dict.
761: (8)                    """
762: (8)                    mst_weight: float
763: (8)                    partition_dict: dict = field(compare=False)
764: (8)                    def __copy__(self):
765: (12)                       return SpanningTreeIterator.Partition(
766: (16)                           self.mst_weight, self.partition_dict.copy()
767: (12)                       )
768: (4)                def __init__(self, G, weight="weight", minimum=True, ignore_nan=False):
769: (8)                    """
770: (8)                    Initialize the iterator
771: (8)                    Parameters
772: (8)                    ----------
773: (8)                    G : nx.Graph
774: (12)                       The directed graph which we need to iterate trees over
775: (8)                    weight : String, default = "weight"
776: (12)                       The edge attribute used to store the weight of the edge
777: (8)                    minimum : bool, default = True
778: (12)                       Return the trees in increasing order while true and decreasing
order
779: (12)                       while false.
780: (8)                    ignore_nan : bool, default = False
781: (12)                       If a NaN is found as an edge weight normally an exception is
raised.
782: (12)                       If `ignore_nan is True` then that edge is ignored instead.
783: (8)                    """
784: (8)                    self.G = G.copy()
785: (8)                    self.G.__networkx_cache__ = None  # Disable caching
786: (8)                    self.weight = weight
787: (8)                    self.minimum = minimum
788: (8)                    self.ignore_nan = ignore_nan
789: (8)                    self.partition_key = (
790: (12)                       "SpanningTreeIterators super secret partition attribute name"
791: (8)                    )
792: (4)                def __iter__(self):
793: (8)                    """
794: (8)                    Returns
795: (8)                    -------
796: (8)                    SpanningTreeIterator
797: (12)                       The iterator object for this graph
798: (8)                    """
```

```
799: (8)                    self.partition_queue = PriorityQueue()
800: (8)                    self._clear_partition(self.G)
801: (8)                    mst_weight = partition_spanning_tree(
802: (12)                       self.G, self.minimum, self.weight, self.partition_key,
self.ignore_nan
803: (8)                    ).size(weight=self.weight)
804: (8)                    self.partition_queue.put(
805: (12)                       self.Partition(mst_weight if self.minimum else -mst_weight, {})
806: (8)                    )
807: (8)                    return self
808: (4)                def __next__(self):
809: (8)                    """
810: (8)                    Returns
811: (8)                    -------
812: (8)                    (multi)Graph
813: (12)                       The spanning tree of next greatest weight, which ties broken
814: (12)                       arbitrarily.
815: (8)                    """
816: (8)                    if self.partition_queue.empty():
817: (12)                       del self.G, self.partition_queue
818: (12)                       raise StopIteration
819: (8)                    partition = self.partition_queue.get()
820: (8)                    self._write_partition(partition)
821: (8)                    next_tree = partition_spanning_tree(
822: (12)                       self.G, self.minimum, self.weight, self.partition_key,
self.ignore_nan
823: (8)                    )
824: (8)                    self._partition(partition, next_tree)
825: (8)                    self._clear_partition(next_tree)
826: (8)                    return next_tree
827: (4)                def _partition(self, partition, partition_tree):
828: (8)                    """
829: (8)                    Create new partitions based of the minimum spanning tree of the
830: (8)                    current minimum partition.
831: (8)                    Parameters
832: (8)                    ----------
833: (8)                    partition : Partition
834: (12)                       The Partition instance used to generate the current minimum
spanning
835: (12)                       tree.
836: (8)                    partition_tree : nx.Graph
837: (12)                       The minimum spanning tree of the input partition.
838: (8)                    """
839: (8)                    p1 = self.Partition(0, partition.partition_dict.copy())
840: (8)                    p2 = self.Partition(0, partition.partition_dict.copy())
841: (8)                    for e in partition_tree.edges:
842: (12)                       if e not in partition.partition_dict:
843: (16)                           p1.partition_dict[e] = EdgePartition.EXCLUDED
844: (16)                           p2.partition_dict[e] = EdgePartition.INCLUDED
845: (16)                           self._write_partition(p1)
846: (16)                           p1_mst = partition_spanning_tree(
847: (20)                               self.G,
848: (20)                               self.minimum,
849: (20)                               self.weight,
850: (20)                               self.partition_key,
851: (20)                               self.ignore_nan,
852: (16)                           )
853: (16)                           p1_mst_weight = p1_mst.size(weight=self.weight)
854: (16)                           if nx.is_connected(p1_mst):
855: (20)                               p1.mst_weight = p1_mst_weight if self.minimum else -
p1_mst_weight
856: (20)                               self.partition_queue.put(p1.__copy__())
857: (16)                           p1.partition_dict = p2.partition_dict.copy()
858: (4)                def _write_partition(self, partition):
859: (8)                    """
860: (8)                    Writes the desired partition into the graph to calculate the minimum
861: (8)                    spanning tree.
862: (8)                    Parameters
863: (8)                    ----------
```

```
(8)                     partition : Partition
(12)                        A Partition dataclass describing a partition on the edges of the
(12)                        graph.
(8)                     """
(8)                     for u, v, d in self.G.edges(data=True):
(12)                        if (u, v) in partition.partition_dict:
(16)                            d[self.partition_key] = partition.partition_dict[(u, v)]
(12)                        else:
(16)                            d[self.partition_key] = EdgePartition.OPEN
(4)                 def _clear_partition(self, G):
(8)                     """
(8)                     Removes partition data from the graph
(8)                     """
(8)                     for u, v, d in G.edges(data=True):
(12)                        if self.partition_key in d:
(16)                            del d[self.partition_key]
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def number_of_spanning_trees(G, *, root=None, weight=None):
(4)                 """Returns the number of spanning trees in `G`.
(4)                 A spanning tree for an undirected graph is a tree that connects
(4)                 all nodes in the graph. For a directed graph, the analog of a
(4)                 spanning tree is called a (spanning) arborescence. The arborescence
(4)                 includes a unique directed path from the `root` node to each other node.
(4)                 The graph must be weakly connected, and the root must be a node
(4)                 that includes all nodes as successors [3]_. Note that to avoid
(4)                 discussing sink-roots and reverse-arborescences, we have reversed
(4)                 the edge orientation from [3]_ and use the in-degree laplacian.
(4)                 This function (when `weight` is `None`) returns the number of
(4)                 spanning trees for an undirected graph and the number of
(4)                 arborescences from a single root node for a directed graph.
(4)                 When `weight` is the name of an edge attribute which holds the
(4)                 weight value of each edge, the function returns the sum over
(4)                 all trees of the multiplicative weight of each tree. That is,
(4)                 the weight of the tree is the product of its edge weights.
(4)                 Kirchoff's Tree Matrix Theorem states that any cofactor of the
(4)                 Laplacian matrix of a graph is the number of spanning trees in the
(4)                 graph. (Here we use cofactors for a diagonal entry so that the
(4)                 cofactor becomes the determinant of the matrix with one row
(4)                 and its matching column removed.) For a weighted Laplacian matrix,
(4)                 the cofactor is the sum across all spanning trees of the
(4)                 multiplicative weight of each tree. That is, the weight of each
(4)                 tree is the product of its edge weights. The theorem is also
(4)                 known as Kirchhoff's theorem [1]_ and the Matrix-Tree theorem [2]_.
(4)                 For directed graphs, a similar theorem (Tutte's Theorem) holds with
(4)                 the cofactor chosen to be the one with row and column removed that
(4)                 correspond to the root. The cofactor is the number of arborescences
(4)                 with the specified node as root. And the weighted version gives the
(4)                 sum of the arborescence weights with root `root`. The arborescence
(4)                 weight is the product of its edge weights.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 root : node
(7)                    A node in the directed graph `G` that has all nodes as descendants.
(7)                    (This is ignored for undirected graphs.)
(4)                 weight : string or None, optional (default=None)
(8)                     The name of the edge attribute holding the edge weight.
(8)                     If `None`, then each edge is assumed to have a weight of 1.
(4)                 Returns
(4)                 -------
(4)                 Number
(8)                     Undirected graphs:
(12)                        The number of spanning trees of the graph `G`.
(12)                        Or the sum of all spanning tree weights of the graph `G`
(12)                        where the weight of a tree is the product of its edge weights.
(8)                     Directed graphs:
(12)                        The number of arborescences of `G` rooted at node `root`.
(12)                        Or the sum of all arborescence weights of the graph `G` with
(12)                        specified root where the weight of an arborescence is the product
```

```
933: (12)                        of its edge weights.
934: (4)                Raises
935: (4)                ------
936: (4)                NetworkXPointlessConcept
937: (8)                    If `G` does not contain any nodes.
938: (4)                NetworkXError
939: (8)                    If the graph `G` is directed and the root node
940: (8)                    is not specified or is not in G.
941: (4)                Examples
942: (4)                --------
943: (4)                >>> G = nx.complete_graph(5)
944: (4)                >>> round(nx.number_of_spanning_trees(G))
945: (4)                125
946: (4)                >>> G = nx.Graph()
947: (4)                >>> G.add_edge(1, 2, weight=2)
948: (4)                >>> G.add_edge(1, 3, weight=1)
949: (4)                >>> G.add_edge(2, 3, weight=1)
950: (4)                >>> round(nx.number_of_spanning_trees(G, weight="weight"))
951: (4)                5
952: (4)                Notes
953: (4)                -----
954: (4)                Self-loops are excluded. Multi-edges are contracted in one edge
955: (4)                equal to the sum of the weights.
956: (4)                References
957: (4)                ----------
958: (4)                .. [1] Wikipedia
959: (7)                   "Kirchhoff's theorem."
960: (7)                   https://en.wikipedia.org/wiki/Kirchhoff%27s_theorem
961: (4)                .. [2] Kirchhoff, G. R.
962: (8)                    Über die Auflösung der Gleichungen, auf welche man
963: (8)                    bei der Untersuchung der linearen Vertheilung
964: (8)                    Galvanischer Ströme geführt wird
965: (8)                    Annalen der Physik und Chemie, vol. 72, pp. 497-508, 1847.
966: (4)                .. [3] Margoliash, J.
967: (8)                    "Matrix-Tree Theorem for Directed Graphs"
968: (8)
https://www.math.uchicago.edu/~may/VIGRE/VIGRE2010/REUPapers/Margoliash.pdf
969: (4)                """
970: (4)                import numpy as np
971: (4)                if len(G) == 0:
972: (8)                    raise nx.NetworkXPointlessConcept("Graph G must contain at least one
node.")
973: (4)                if not nx.is_directed(G):
974: (8)                    if not nx.is_connected(G):
975: (12)                       return 0
976: (8)                    G_laplacian = nx.laplacian_matrix(G, weight=weight).toarray()
977: (8)                    return float(np.linalg.det(G_laplacian[1:, 1:]))
978: (4)                if root is None:
979: (8)                    raise nx.NetworkXError("Input `root` must be provided when G is
directed")
980: (4)                if root not in G:
981: (8)                    raise nx.NetworkXError("The node root is not in the graph G.")
982: (4)                if not nx.is_weakly_connected(G):
983: (8)                    return 0
984: (4)                nodelist = [root] + [n for n in G if n != root]
985: (4)                A = nx.adjacency_matrix(G, nodelist=nodelist, weight=weight)
986: (4)                D = np.diag(A.sum(axis=0))
987: (4)                G_laplacian = D - A
988: (4)                return float(np.linalg.det(G_laplacian[1:, 1:]))

----------------------------------------

File 379 - operations.py:

1: (0)              """Operations on trees."""
2: (0)              from functools import partial
3: (0)              from itertools import accumulate, chain
4: (0)              import networkx as nx
5: (0)              __all__ = ["join", "join_trees"]
```

```
6: (0)              def join(rooted_trees, label_attribute=None):
7: (4)                  """A deprecated name for `join_trees`
8: (4)                  Returns a new rooted tree with a root node joined with the roots
9: (4)                  of each of the given rooted trees.
10: (4)                 .. deprecated:: 3.2
11: (7)                    `join` is deprecated in NetworkX v3.2 and will be removed in v3.4.
12: (7)                    It has been renamed join_trees with the same syntax/interface.
13: (4)                 """
14: (4)                 import warnings
15: (4)                 warnings.warn(
16: (8)                     "The function `join` is deprecated and is renamed `join_trees`.\n"
17: (8)                     "The ``join`` function itself will be removed in v3.4",
18: (8)                     DeprecationWarning,
19: (8)                     stacklevel=2,
20: (4)                 )
21: (4)                 return join_trees(rooted_trees, label_attribute=label_attribute)
22: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
23: (0)             def join_trees(rooted_trees, *, label_attribute=None, first_label=0):
24: (4)                 """Returns a new rooted tree made by joining `rooted_trees`
25: (4)                 Constructs a new tree by joining each tree in `rooted_trees`.
26: (4)                 A new root node is added and connected to each of the roots
27: (4)                 of the input trees. While copying the nodes from the trees,
28: (4)                 relabeling to integers occurs. If the `label_attribute` is provided,
29: (4)                 the old node labels will be stored in the new tree under this attribute.
30: (4)                 Parameters
31: (4)                 ----------
32: (4)                 rooted_trees : list
33: (8)                     A list of pairs in which each left element is a NetworkX graph
34: (8)                     object representing a tree and each right element is the root
35: (8)                     node of that tree. The nodes of these trees will be relabeled to
36: (8)                     integers.
37: (4)                 label_attribute : str
38: (8)                     If provided, the old node labels will be stored in the new tree
39: (8)                     under this node attribute. If not provided, the original labels
40: (8)                     of the nodes in the input trees are not stored.
41: (4)                 first_label : int, optional (default=0)
42: (8)                     Specifies the label for the new root node. If provided, the root node
of the joined tree
43: (8)                     will have this label. If not provided, the root node will default to a
label of 0.
44: (4)                 Returns
45: (4)                 -------
46: (4)                 NetworkX graph
47: (8)                     The rooted tree resulting from joining the provided `rooted_trees`.
The new tree has a root node
48: (8)                     labeled as specified by `first_label` (defaulting to 0 if not
provided). Subtrees from the input
49: (8)                     `rooted_trees` are attached to this new root node. Each non-root node,
if the `label_attribute`
50: (8)                     is provided, has an attribute that indicates the original label of the
node in the input tree.
51: (4)                 Notes
52: (4)                 -----
53: (4)                 Trees are stored in NetworkX as NetworkX Graphs. There is no specific
54: (4)                 enforcement of the fact that these are trees. Testing for each tree
55: (4)                 can be done using :func:`networkx.is_tree`.
56: (4)                 Graph, edge, and node attributes are propagated from the given
57: (4)                 rooted trees to the created tree. If there are any overlapping graph
58: (4)                 attributes, those from later trees will overwrite those from earlier
59: (4)                 trees in the tuple of positional arguments.
60: (4)                 Examples
61: (4)                 --------
62: (4)                 Join two full balanced binary trees of height *h* to get a full
63: (4)                 balanced binary tree of depth *h* + 1::
64: (8)                     >>> h = 4
65: (8)                     >>> left = nx.balanced_tree(2, h)
66: (8)                     >>> right = nx.balanced_tree(2, h)
67: (8)                     >>> joined_tree = nx.join([(left, 0), (right, 0)])
68: (8)                     >>> nx.is_isomorphic(joined_tree, nx.balanced_tree(2, h + 1))
```

```
69: (8)                     True
70: (4)                 """
71: (4)                 if not rooted_trees:
72: (8)                     return nx.empty_graph(1)
73: (4)                 trees, roots = zip(*rooted_trees)
74: (4)                 R = type(trees[0])()
75: (4)                 lengths = (len(tree) for tree in trees[:-1])
76: (4)                 first_labels = list(accumulate(lengths, initial=first_label + 1))
77: (4)                 new_roots = []
78: (4)                 for tree, root, first_node in zip(trees, roots, first_labels):
79: (8)                     new_root = first_node + list(tree.nodes()).index(root)
80: (8)                     new_roots.append(new_root)
81: (4)                 relabel = partial(
82: (8)                     nx.convert_node_labels_to_integers, label_attribute=label_attribute
83: (4)                 )
84: (4)                 new_trees = [
85: (8)                     relabel(tree, first_label=first_label)
86: (8)                     for tree, first_label in zip(trees, first_labels)
87: (4)                 ]
88: (4)                 for tree in new_trees:
89: (8)                     R.update(tree)
90: (4)                 R.add_node(first_label)
91: (4)                 R.add_edges_from((first_label, root) for root in new_roots)
92: (4)                 return R
```

File 380 - __init__.py:

```
1: (0)
```

File 381 - recognition.py:

```
1: (0)              """
2: (0)              Recognition Tests
3: (0)              =================
4: (0)              A *forest* is an acyclic, undirected graph, and a *tree* is a connected
forest.
5: (0)              Depending on the subfield, there are various conventions for generalizing
these
6: (0)              definitions to directed graphs.
7: (0)              In one convention, directed variants of forest and tree are defined in an
8: (0)              identical manner, except that the direction of the edges is ignored. In
effect,
9: (0)              each directed edge is treated as a single undirected edge. Then, additional
10: (0)             restrictions are imposed to define *branchings* and *arborescences*.
11: (0)             In another convention, directed variants of forest and tree correspond to
12: (0)             the previous convention's branchings and arborescences, respectively. Then two
13: (0)             new terms, *polyforest* and *polytree*, are defined to correspond to the other
14: (0)             convention's forest and tree.
15: (0)             Summarizing::
16: (3)                +-----------------------------+
17: (3)                | Convention A | Convention B |
18: (3)                +=============================+
19: (3)                | forest       | polyforest   |
20: (3)                | tree         | polytree     |
21: (3)                | branching    | forest       |
22: (3)                | arborescence | tree         |
23: (3)                +-----------------------------+
24: (0)             Each convention has its reasons. The first convention emphasizes definitional
25: (0)             similarity in that directed forests and trees are only concerned with
26: (0)             acyclicity and do not have an in-degree constraint, just as their undirected
27: (0)             counterparts do not. The second convention emphasizes functional similarity
28: (0)             in the sense that the directed analog of a spanning tree is a spanning
29: (0)             arborescence. That is, take any spanning tree and choose one node as the root.
30: (0)             Then every edge is assigned a direction such there is a directed path from the
31: (0)             root to every other node. The result is a spanning arborescence.
```

```
 32: (0)             NetworkX follows convention "A". Explicitly, these are:
 33: (0)             undirected forest
 34: (3)                An undirected graph with no undirected cycles.
 35: (0)             undirected tree
 36: (3)                A connected, undirected forest.
 37: (0)             directed forest
 38: (3)                A directed graph with no undirected cycles. Equivalently, the underlying
 39: (3)                graph structure (which ignores edge orientations) is an undirected forest.
 40: (3)                In convention B, this is known as a polyforest.
 41: (0)             directed tree
 42: (3)                A weakly connected, directed forest. Equivalently, the underlying graph
 43: (3)                structure (which ignores edge orientations) is an undirected tree. In
 44: (3)                convention B, this is known as a polytree.
 45: (0)             branching
 46: (3)                A directed forest with each node having, at most, one parent. So the
maximum
 47: (3)                in-degree is equal to 1. In convention B, this is known as a forest.
 48: (0)             arborescence
 49: (3)                A directed tree with each node having, at most, one parent. So the maximum
 50: (3)                in-degree is equal to 1. In convention B, this is known as a tree.
 51: (0)             For trees and arborescences, the adjective "spanning" may be added to
designate
 52: (0)             that the graph, when considered as a forest/branching, consists of a single
 53: (0)             tree/arborescence that includes all nodes in the graph. It is true, by
 54: (0)             definition, that every tree/arborescence is spanning with respect to the nodes
 55: (0)             that define the tree/arborescence and so, it might seem redundant to introduce
 56: (0)             the notion of "spanning". However, the nodes may represent a subset of
 57: (0)             nodes from a larger graph, and it is in this context that the term "spanning"
 58: (0)             becomes a useful notion.
 59: (0)             """
 60: (0)             import networkx as nx
 61: (0)             __all__ = ["is_arborescence", "is_branching", "is_forest", "is_tree"]
 62: (0)             @nx.utils.not_implemented_for("undirected")
 63: (0)             @nx._dispatchable
 64: (0)             def is_arborescence(G):
 65: (4)                 """
 66: (4)                 Returns True if `G` is an arborescence.
 67: (4)                 An arborescence is a directed tree with maximum in-degree equal to 1.
 68: (4)                 Parameters
 69: (4)                 ----------
 70: (4)                 G : graph
 71: (8)                     The graph to test.
 72: (4)                 Returns
 73: (4)                 -------
 74: (4)                 b : bool
 75: (8)                     A boolean that is True if `G` is an arborescence.
 76: (4)                 Examples
 77: (4)                 --------
 78: (4)                 >>> G = nx.DiGraph([(0, 1), (0, 2), (2, 3), (3, 4)])
 79: (4)                 >>> nx.is_arborescence(G)
 80: (4)                 True
 81: (4)                 >>> G.remove_edge(0, 1)
 82: (4)                 >>> G.add_edge(1, 2)  # maximum in-degree is 2
 83: (4)                 >>> nx.is_arborescence(G)
 84: (4)                 False
 85: (4)                 Notes
 86: (4)                 -----
 87: (4)                 In another convention, an arborescence is known as a *tree*.
 88: (4)                 See Also
 89: (4)                 --------
 90: (4)                 is_tree
 91: (4)                 """
 92: (4)                 return is_tree(G) and max(d for n, d in G.in_degree()) <= 1
 93: (0)             @nx.utils.not_implemented_for("undirected")
 94: (0)             @nx._dispatchable
 95: (0)             def is_branching(G):
 96: (4)                 """
 97: (4)                 Returns True if `G` is a branching.
 98: (4)                 A branching is a directed forest with maximum in-degree equal to 1.
```

```
99: (4)                 Parameters
100: (4)                ----------
101: (4)                G : directed graph
102: (8)                    The directed graph to test.
103: (4)                Returns
104: (4)                -------
105: (4)                b : bool
106: (8)                    A boolean that is True if `G` is a branching.
107: (4)                Examples
108: (4)                --------
109: (4)                >>> G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 4)])
110: (4)                >>> nx.is_branching(G)
111: (4)                True
112: (4)                >>> G.remove_edge(2, 3)
113: (4)                >>> G.add_edge(3, 1)  # maximum in-degree is 2
114: (4)                >>> nx.is_branching(G)
115: (4)                False
116: (4)                Notes
117: (4)                -----
118: (4)                In another convention, a branching is also known as a *forest*.
119: (4)                See Also
120: (4)                --------
121: (4)                is_forest
122: (4)                """
123: (4)                return is_forest(G) and max(d for n, d in G.in_degree()) <= 1
124: (0)            @nx._dispatchable
125: (0)            def is_forest(G):
126: (4)                """
127: (4)                Returns True if `G` is a forest.
128: (4)                A forest is a graph with no undirected cycles.
129: (4)                For directed graphs, `G` is a forest if the underlying graph is a forest.
130: (4)                The underlying graph is obtained by treating each directed edge as a
single
131: (4)                undirected edge in a multigraph.
132: (4)                Parameters
133: (4)                ----------
134: (4)                G : graph
135: (8)                    The graph to test.
136: (4)                Returns
137: (4)                -------
138: (4)                b : bool
139: (8)                    A boolean that is True if `G` is a forest.
140: (4)                Raises
141: (4)                ------
142: (4)                NetworkXPointlessConcept
143: (8)                    If `G` is empty.
144: (4)                Examples
145: (4)                --------
146: (4)                >>> G = nx.Graph()
147: (4)                >>> G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 5)])
148: (4)                >>> nx.is_forest(G)
149: (4)                True
150: (4)                >>> G.add_edge(4, 1)
151: (4)                >>> nx.is_forest(G)
152: (4)                False
153: (4)                Notes
154: (4)                -----
155: (4)                In another convention, a directed forest is known as a *polyforest* and
156: (4)                then *forest* corresponds to a *branching*.
157: (4)                See Also
158: (4)                --------
159: (4)                is_branching
160: (4)                """
161: (4)                if len(G) == 0:
162: (8)                    raise nx.exception.NetworkXPointlessConcept("G has no nodes.")
163: (4)                if G.is_directed():
164: (8)                    components = (G.subgraph(c) for c in
nx.weakly_connected_components(G))
165: (4)                else:
```

```
166: (8)                    components = (G.subgraph(c) for c in nx.connected_components(G))
167: (4)                return all(len(c) - 1 == c.number_of_edges() for c in components)
168: (0)            @nx._dispatchable
169: (0)            def is_tree(G):
170: (4)                """
171: (4)                Returns True if `G` is a tree.
172: (4)                A tree is a connected graph with no undirected cycles.
173: (4)                For directed graphs, `G` is a tree if the underlying graph is a tree. The
174: (4)                underlying graph is obtained by treating each directed edge as a single
175: (4)                undirected edge in a multigraph.
176: (4)                Parameters
177: (4)                ----------
178: (4)                G : graph
179: (8)                    The graph to test.
180: (4)                Returns
181: (4)                -------
182: (4)                b : bool
183: (8)                    A boolean that is True if `G` is a tree.
184: (4)                Raises
185: (4)                ------
186: (4)                NetworkXPointlessConcept
187: (8)                    If `G` is empty.
188: (4)                Examples
189: (4)                --------
190: (4)                >>> G = nx.Graph()
191: (4)                >>> G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 5)])
192: (4)                >>> nx.is_tree(G)  # n-1 edges
193: (4)                True
194: (4)                >>> G.add_edge(3, 4)
195: (4)                >>> nx.is_tree(G)  # n edges
196: (4)                False
197: (4)                Notes
198: (4)                -----
199: (4)                In another convention, a directed tree is known as a *polytree* and then
200: (4)                *tree* corresponds to an *arborescence*.
201: (4)                See Also
202: (4)                --------
203: (4)                is_arborescence
204: (4)                """
205: (4)                if len(G) == 0:
206: (8)                    raise nx.exception.NetworkXPointlessConcept("G has no nodes.")
207: (4)                if G.is_directed():
208: (8)                    is_connected = nx.is_weakly_connected
209: (4)                else:
210: (8)                    is_connected = nx.is_connected
211: (4)                return len(G) - 1 == G.number_of_edges() and is_connected(G)

----------------------------------------

File 382 - test_coding.py:

1: (0)              """Unit tests for the :mod:`~networkx.algorithms.tree.coding` module."""
2: (0)              from itertools import product
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.utils import edges_equal, nodes_equal
6: (0)              class TestPruferSequence:
7: (4)                  """Unit tests for the Prüfer sequence encoding and decoding
8: (4)                  functions.
9: (4)                  """
10: (4)                 def test_nontree(self):
11: (8)                     with pytest.raises(nx.NotATree):
12: (12)                        G = nx.cycle_graph(3)
13: (12)                        nx.to_prufer_sequence(G)
14: (4)                 def test_null_graph(self):
15: (8)                     with pytest.raises(nx.NetworkXPointlessConcept):
16: (12)                        nx.to_prufer_sequence(nx.null_graph())
17: (4)                 def test_trivial_graph(self):
18: (8)                     with pytest.raises(nx.NetworkXPointlessConcept):
```

```
(12)                        nx.to_prufer_sequence(nx.trivial_graph())
(4)                 def test_bad_integer_labels(self):
(8)                     with pytest.raises(KeyError):
(12)                        T = nx.Graph(nx.utils.pairwise("abc"))
(12)                        nx.to_prufer_sequence(T)
(4)                 def test_encoding(self):
(8)                     """Tests for encoding a tree as a Prüfer sequence using the
(8)                     iterative strategy.
(8)                     """
(8)                     tree = nx.Graph([(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)])
(8)                     sequence = nx.to_prufer_sequence(tree)
(8)                     assert sequence == [3, 3, 3, 4]
(4)                 def test_decoding(self):
(8)                     """Tests for decoding a tree from a Prüfer sequence."""
(8)                     sequence = [3, 3, 3, 4]
(8)                     tree = nx.from_prufer_sequence(sequence)
(8)                     assert nodes_equal(list(tree), list(range(6)))
(8)                     edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
(8)                     assert edges_equal(list(tree.edges()), edges)
(4)                 def test_decoding2(self):
(8)                     sequence = [2, 4, 0, 1, 3, 3]
(8)                     tree = nx.from_prufer_sequence(sequence)
(8)                     assert nodes_equal(list(tree), list(range(8)))
(8)                     edges = [(0, 1), (0, 4), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)]
(8)                     assert edges_equal(list(tree.edges()), edges)
(4)                 def test_inverse(self):
(8)                     """Tests that the encoding and decoding functions are inverses."""
(8)                     for T in nx.nonisomorphic_trees(4):
(12)                        T2 = nx.from_prufer_sequence(nx.to_prufer_sequence(T))
(12)                        assert nodes_equal(list(T), list(T2))
(12)                        assert edges_equal(list(T.edges()), list(T2.edges()))
(8)                     for seq in product(range(4), repeat=2):
(12)                        seq2 = nx.to_prufer_sequence(nx.from_prufer_sequence(seq))
(12)                        assert list(seq) == seq2
(0)             class TestNestedTuple:
(4)                 """Unit tests for the nested tuple encoding and decoding functions."""
(4)                 def test_nontree(self):
(8)                     with pytest.raises(nx.NotATree):
(12)                        G = nx.cycle_graph(3)
(12)                        nx.to_nested_tuple(G, 0)
(4)                 def test_unknown_root(self):
(8)                     with pytest.raises(nx.NodeNotFound):
(12)                        G = nx.path_graph(2)
(12)                        nx.to_nested_tuple(G, "bogus")
(4)                 def test_encoding(self):
(8)                     T = nx.full_rary_tree(2, 2**3 - 1)
(8)                     expected = ((((), ()), ((), ()))
(8)                     actual = nx.to_nested_tuple(T, 0)
(8)                     assert nodes_equal(expected, actual)
(4)                 def test_canonical_form(self):
(8)                     T = nx.Graph()
(8)                     T.add_edges_from([(0, 1), (0, 2), (0, 3)])
(8)                     T.add_edges_from([(1, 4), (1, 5)])
(8)                     T.add_edges_from([(3, 6), (3, 7)])
(8)                     root = 0
(8)                     actual = nx.to_nested_tuple(T, root, canonical_form=True)
(8)                     expected = ((), ((), ()), ((), ()))
(8)                     assert actual == expected
(4)                 def test_decoding(self):
(8)                     balanced = ((((), ()), ((), ()))
(8)                     expected = nx.full_rary_tree(2, 2**3 - 1)
(8)                     actual = nx.from_nested_tuple(balanced)
(8)                     assert nx.is_isomorphic(expected, actual)
(4)                 def test_sensible_relabeling(self):
(8)                     balanced = ((((), ()), ((), ()))
(8)                     T = nx.from_nested_tuple(balanced, sensible_relabeling=True)
(8)                     edges = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)]
(8)                     assert nodes_equal(list(T), list(range(2**3 - 1)))
(8)                     assert edges_equal(list(T.edges()), edges)
```

```
----------------------------------------

File 383 - test_branchings.py:

1: (0)              import math
2: (0)              from operator import itemgetter
3: (0)              import pytest
4: (0)              np = pytest.importorskip("numpy")
5: (0)              import networkx as nx
6: (0)              from networkx.algorithms.tree import branchings, recognition
7: (0)              G_array = np.array([
8: (4)                  [0, 0, 12, 0, 12, 0, 0, 0, 0],  # 0
9: (4)                  [4, 0, 0, 0, 0, 13, 0, 0, 0],  # 1
10: (4)                 [0, 17, 0, 21, 0, 12, 0, 0, 0],  # 2
11: (4)                 [5, 0, 0, 0, 17, 0, 18, 0, 0],  # 3
12: (4)                 [0, 0, 0, 0, 0, 0, 0, 12, 0],  # 4
13: (4)                 [0, 0, 0, 0, 0, 0, 14, 0, 12],  # 5
14: (4)                 [0, 0, 21, 0, 0, 0, 0, 0, 15],  # 6
15: (4)                 [0, 0, 0, 19, 0, 0, 15, 0, 0],  # 7
16: (4)                 [0, 0, 0, 0, 0, 0, 0, 18, 0],  # 8
17: (0)             ], dtype=int)
18: (0)             G_big_array = np.zeros(np.array(G_array.shape) * 2, dtype=int)
19: (0)             G_big_array[:G_array.shape[0], :G_array.shape[1]] = G_array
20: (0)             G_big_array[G_array.shape[0]:, G_array.shape[1]:] = G_array
21: (0)             def G1():
22: (4)                 G = nx.from_numpy_array(G_array, create_using=nx.MultiDiGraph)
23: (4)                 return G
24: (0)             def G2():
25: (4)                 Garr = G_array.copy()
26: (4)                 Garr[np.nonzero(Garr)] -= 10
27: (4)                 G = nx.from_numpy_array(Garr, create_using=nx.MultiDiGraph)
28: (4)                 return G
29: (0)             optimal_arborescence_1 = [
30: (4)                 (0, 2, 12),
31: (4)                 (2, 1, 17),
32: (4)                 (2, 3, 21),
33: (4)                 (1, 5, 13),
34: (4)                 (3, 4, 17),
35: (4)                 (3, 6, 18),
36: (4)                 (6, 8, 15),
37: (4)                 (8, 7, 18),
38: (0)             ]
39: (0)             optimal_branching_2a = [
40: (4)                 (5, 6, 4),
41: (4)                 (6, 2, 11),
42: (4)                 (6, 8, 5),
43: (4)                 (8, 7, 8),
44: (4)                 (2, 1, 7),
45: (4)                 (2, 3, 11),
46: (4)                 (3, 4, 7),
47: (0)             ]
48: (0)             optimal_branching_2b = [
49: (4)                 (8, 7, 8),
50: (4)                 (7, 3, 9),
51: (4)                 (3, 4, 7),
52: (4)                 (3, 6, 8),
53: (4)                 (6, 2, 11),
54: (4)                 (2, 1, 7),
55: (4)                 (1, 5, 3),
56: (0)             ]
57: (0)             optimal_arborescence_2 = [
58: (4)                 (0, 2, 2),
59: (4)                 (2, 1, 7),
60: (4)                 (2, 3, 11),
61: (4)                 (1, 5, 3),
62: (4)                 (3, 4, 7),
63: (4)                 (3, 6, 8),
64: (4)                 (6, 8, 5),
```

```
65: (4)                 (8, 7, 8),
66: (0)             ]
67: (0)             greedy_subopt_branching_1a = [
68: (4)                 (5, 6, 14),
69: (4)                 (6, 2, 21),
70: (4)                 (6, 8, 15),
71: (4)                 (8, 7, 18),
72: (4)                 (2, 1, 17),
73: (4)                 (2, 3, 21),
74: (4)                 (3, 0, 5),
75: (4)                 (3, 4, 17),
76: (0)             ]
77: (0)             greedy_subopt_branching_1b = [
78: (4)                 (8, 7, 18),
79: (4)                 (7, 6, 15),
80: (4)                 (6, 2, 21),
81: (4)                 (2, 1, 17),
82: (4)                 (2, 3, 21),
83: (4)                 (1, 5, 13),
84: (4)                 (3, 0, 5),
85: (4)                 (3, 4, 17),
86: (0)             ]
87: (0)             def build_branching(edges, double=False):
88: (4)                 G = nx.DiGraph()
89: (4)                 for u, v, weight in edges:
90: (8)                     G.add_edge(u, v, weight=weight)
91: (8)                     if double:
92: (12)                        G.add_edge(u + 9, v + 9, weight=weight)
93: (4)                 return G
94: (0)             def sorted_edges(G, attr="weight", default=1):
95: (4)                 edges = [(u, v, data.get(attr, default)) for (u, v, data) in
G.edges(data=True)]
96: (4)                 edges = sorted(edges, key=lambda x: (x[2], x[1], x[0]))
97: (4)                 return edges
98: (0)             def assert_equal_branchings(G1, G2, attr="weight", default=1):
99: (4)                 edges1 = list(G1.edges(data=True))
100: (4)                edges2 = list(G2.edges(data=True))
101: (4)                assert len(edges1) == len(edges2)
102: (4)                e1 = sorted_edges(G1, attr, default)
103: (4)                e2 = sorted_edges(G2, attr, default)
104: (4)                for a, b in zip(e1, e2):
105: (8)                    assert a[:2] == b[:2]
106: (8)                    np.testing.assert_almost_equal(a[2], b[2])
107: (0)            def test_optimal_branching1():
108: (4)                G = build_branching(optimal_arborescence_1)
109: (4)                assert recognition.is_arborescence(G), True
110: (4)                assert branchings.branching_weight(G) == 131
111: (0)            def test_optimal_branching2a():
112: (4)                G = build_branching(optimal_branching_2a)
113: (4)                assert recognition.is_arborescence(G), True
114: (4)                assert branchings.branching_weight(G) == 53
115: (0)            def test_optimal_branching2b():
116: (4)                G = build_branching(optimal_branching_2b)
117: (4)                assert recognition.is_arborescence(G), True
118: (4)                assert branchings.branching_weight(G) == 53
119: (0)            def test_optimal_arborescence2():
120: (4)                G = build_branching(optimal_arborescence_2)
121: (4)                assert recognition.is_arborescence(G), True
122: (4)                assert branchings.branching_weight(G) == 51
123: (0)            def test_greedy_suboptimal_branching1a():
124: (4)                G = build_branching(greedy_subopt_branching_1a)
125: (4)                assert recognition.is_arborescence(G), True
126: (4)                assert branchings.branching_weight(G) == 128
127: (0)            def test_greedy_suboptimal_branching1b():
128: (4)                G = build_branching(greedy_subopt_branching_1b)
129: (4)                assert recognition.is_arborescence(G), True
130: (4)                assert branchings.branching_weight(G) == 127
131: (0)            def test_greedy_max1():
132: (4)                G = G1()
```

```
133: (4)                B = branchings.greedy_branching(G)
134: (4)                B_ = build_branching(greedy_subopt_branching_1b)
135: (4)                assert_equal_branchings(B, B_)
136: (0)            def test_greedy_branching_kwarg_kind():
137: (4)                G = G1()
138: (4)                with pytest.raises(nx.NetworkXException, match="Unknown value for
`kind`."):
139: (8)                    B = branchings.greedy_branching(G, kind="lol")
140: (0)            def test_greedy_branching_for_unsortable_nodes():
141: (4)                G = nx.DiGraph()
142: (4)                G.add_weighted_edges_from([((2, 3), 5, 1), (3, "a", 1), (2, 4, 5)])
143: (4)                edges = [(u, v, data.get("weight", 1)) for (u, v, data) in
G.edges(data=True)]
144: (4)                with pytest.raises(TypeError):
145: (8)                    edges.sort(key=itemgetter(2, 0, 1), reverse=True)
146: (4)                B = branchings.greedy_branching(G, kind="max").edges(data=True)
147: (4)                assert list(B) == [
148: (8)                    ((2, 3), 5, {"weight": 1}),
149: (8)                    (3, "a", {"weight": 1}),
150: (8)                    (2, 4, {"weight": 5}),
151: (4)                ]
152: (0)            def test_greedy_max2():
153: (4)                G = G1()
154: (4)                del G[1][0][0]["weight"]
155: (4)                B = branchings.greedy_branching(G, default=6)
156: (4)                edges = [
157: (8)                    (1, 0, 6),
158: (8)                    (1, 5, 13),
159: (8)                    (7, 6, 15),
160: (8)                    (2, 1, 17),
161: (8)                    (3, 4, 17),
162: (8)                    (8, 7, 18),
163: (8)                    (2, 3, 21),
164: (8)                    (6, 2, 21),
165: (4)                ]
166: (4)                B_ = build_branching(edges)
167: (4)                assert_equal_branchings(B, B_)
168: (0)            def test_greedy_max3():
169: (4)                G = G1()
170: (4)                B = branchings.greedy_branching(G, attr=None)
171: (4)                edges = [
172: (8)                    (2, 1, 1),
173: (8)                    (3, 0, 1),
174: (8)                    (3, 4, 1),
175: (8)                    (5, 8, 1),
176: (8)                    (6, 2, 1),
177: (8)                    (7, 3, 1),
178: (8)                    (7, 6, 1),
179: (8)                    (8, 7, 1),
180: (4)                ]
181: (4)                B_ = build_branching(edges)
182: (4)                assert_equal_branchings(B, B_, default=1)
183: (0)            def test_greedy_min():
184: (4)                G = G1()
185: (4)                B = branchings.greedy_branching(G, kind="min")
186: (4)                edges = [
187: (8)                    (1, 0, 4),
188: (8)                    (0, 2, 12),
189: (8)                    (0, 4, 12),
190: (8)                    (2, 5, 12),
191: (8)                    (4, 7, 12),
192: (8)                    (5, 8, 12),
193: (8)                    (5, 6, 14),
194: (8)                    (7, 3, 19),
195: (4)                ]
196: (4)                B_ = build_branching(edges)
197: (4)                assert_equal_branchings(B, B_)
198: (0)            def test_edmonds1_maxbranch():
199: (4)                G = G1()
```

```
(4)                 x = branchings.maximum_branching(G)
(4)                 x_ = build_branching(optimal_arborescence_1)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edmonds1_maxarbor():
(4)                 G = G1()
(4)                 x = branchings.maximum_spanning_arborescence(G)
(4)                 x_ = build_branching(optimal_arborescence_1)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edmonds1_minimal_branching():
(4)                 G = nx.from_numpy_array(G_big_array, create_using=nx.DiGraph)
(4)                 B = branchings.minimal_branching(G)
(4)                 edges = [
(8)                     (3, 0, 5),
(8)                     (0, 2, 12),
(8)                     (0, 4, 12),
(8)                     (2, 5, 12),
(8)                     (4, 7, 12),
(8)                     (5, 8, 12),
(8)                     (5, 6, 14),
(8)                     (2, 1, 17),
(4)                 ]
(4)                 B_ = build_branching(edges, double=True)
(4)                 assert_equal_branchings(B, B_)
(0)             def test_edmonds2_maxbranch():
(4)                 G = G2()
(4)                 x = branchings.maximum_branching(G)
(4)                 x_ = build_branching(optimal_branching_2a)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edmonds2_maxarbor():
(4)                 G = G2()
(4)                 x = branchings.maximum_spanning_arborescence(G)
(4)                 x_ = build_branching(optimal_arborescence_2)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edmonds2_minarbor():
(4)                 G = G1()
(4)                 x = branchings.minimum_spanning_arborescence(G)
(4)                 edges = [
(8)                     (3, 0, 5),
(8)                     (0, 2, 12),
(8)                     (0, 4, 12),
(8)                     (2, 5, 12),
(8)                     (4, 7, 12),
(8)                     (5, 8, 12),
(8)                     (5, 6, 14),
(8)                     (2, 1, 17),
(4)                 ]
(4)                 x_ = build_branching(edges)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edmonds3_minbranch1():
(4)                 G = G1()
(4)                 x = branchings.minimum_branching(G)
(4)                 edges = []
(4)                 x_ = build_branching(edges)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edmonds3_minbranch2():
(4)                 G = G1()
(4)                 G.add_edge(8, 9, weight=-10)
(4)                 x = branchings.minimum_branching(G)
(4)                 edges = [(8, 9, -10)]
(4)                 x_ = build_branching(edges)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_mst():
(4)                 G = nx.Graph()
(4)                 edgelist = [
(8)                     (0, 3, [("weight", 5)]),
(8)                     (0, 1, [("weight", 7)]),
(8)                     (1, 3, [("weight", 9)]),
(8)                     (1, 2, [("weight", 8)]),
(8)                     (1, 4, [("weight", 7)]),
```

```
(8)                     (3, 4, [("weight", 15)]),
(8)                     (3, 5, [("weight", 6)]),
(8)                     (2, 4, [("weight", 5)]),
(8)                     (4, 5, [("weight", 8)]),
(8)                     (4, 6, [("weight", 9)]),
(8)                     (5, 6, [("weight", 11)]),
(4)                 ]
(4)                 G.add_edges_from(edgelist)
(4)                 G = G.to_directed()
(4)                 x = branchings.minimum_spanning_arborescence(G)
(4)                 edges = [
(8)                     ({0, 1}, 7),
(8)                     ({0, 3}, 5),
(8)                     ({3, 5}, 6),
(8)                     ({1, 4}, 7),
(8)                     ({4, 2}, 5),
(8)                     ({4, 6}, 9),
(4)                 ]
(4)                 assert x.number_of_edges() == len(edges)
(4)                 for u, v, d in x.edges(data=True):
(8)                     assert ({u, v}, d["weight"]) in edges
(0)             def test_mixed_nodetypes():
(4)                 G = nx.Graph()
(4)                 edgelist = [(0, 3, [("weight", 5)]), (0, "1", [("weight", 5)])]
(4)                 G.add_edges_from(edgelist)
(4)                 G = G.to_directed()
(4)                 x = branchings.minimum_spanning_arborescence(G)
(0)             def test_edmonds1_minbranch():
(4)                 edges = [(u, v, -w) for (u, v, w) in optimal_arborescence_1]
(4)                 G = nx.from_numpy_array(-G_array, create_using=nx.DiGraph)
(4)                 x = branchings.maximum_branching(G)
(4)                 x_ = build_branching([])
(4)                 assert_equal_branchings(x, x_)
(4)                 x = branchings.minimum_branching(G)
(4)                 x_ = build_branching(edges)
(4)                 assert_equal_branchings(x, x_)
(0)             def test_edge_attribute_preservation_normal_graph():
(4)                 G = nx.Graph()
(4)                 edgelist = [
(8)                     (0, 1, [("weight", 5), ("otherattr", 1), ("otherattr2", 3)]),
(8)                     (0, 2, [("weight", 5), ("otherattr", 2), ("otherattr2", 2)]),
(8)                     (1, 2, [("weight", 6), ("otherattr", 3), ("otherattr2", 1)]),
(4)                 ]
(4)                 G.add_edges_from(edgelist)
(4)                 B = branchings.maximum_branching(G, preserve_attrs=True)
(4)                 assert B[0][1]["otherattr"] == 1
(4)                 assert B[0][1]["otherattr2"] == 3
(0)             def test_edge_attribute_preservation_multigraph():
(4)                 G = nx.MultiGraph()
(4)                 edgelist = [
(8)                     (0, 1, [("weight", 5), ("otherattr", 1), ("otherattr2", 3)]),
(8)                     (0, 2, [("weight", 5), ("otherattr", 2), ("otherattr2", 2)]),
(8)                     (1, 2, [("weight", 6), ("otherattr", 3), ("otherattr2", 1)]),
(4)                 ]
(4)                 G.add_edges_from(edgelist * 2)  # Make sure we have duplicate edge paths
(4)                 B = branchings.maximum_branching(G, preserve_attrs=True)
(4)                 assert B[0][1][0]["otherattr"] == 1
(4)                 assert B[0][1][0]["otherattr2"] == 3
(0)             def test_Edmond_kind():
(4)                 G = nx.MultiGraph()
(4)                 edgelist = [
(8)                     (0, 1, [("weight", 5), ("otherattr", 1), ("otherattr2", 3)]),
(8)                     (0, 2, [("weight", 5), ("otherattr", 2), ("otherattr2", 2)]),
(8)                     (1, 2, [("weight", 6), ("otherattr", 3), ("otherattr2", 1)]),
(4)                 ]
(4)                 G.add_edges_from(edgelist * 2)  # Make sure we have duplicate edge paths
(4)                 ed = branchings.Edmonds(G)
(4)                 with pytest.raises(nx.NetworkXException, match="Unknown value for
`kind`."):
```

```
(8)                 ed.find_optimum(kind="lol", preserve_attrs=True)
(0)         def test_MultiDiGraph_EdgeKey():
(4)             G = branchings.MultiDiGraph_EdgeKey()
(4)             G.add_edge(1, 2, "A")
(4)             with pytest.raises(Exception, match="Key 'A' is already in use."):
(8)                 G.add_edge(3, 4, "A")
(4)             with pytest.raises(KeyError, match="Invalid edge key 'B'"):
(8)                 G.remove_edge_with_key("B")
(4)             if G.remove_edge_with_key("A"):
(8)                 assert list(G.edges(data=True)) == []
(4)             G.add_edge(1, 3, "A")
(4)             with pytest.raises(NotImplementedError):
(8)                 G.remove_edges_from([(1, 3)])
(0)         def test_edge_attribute_discard():
(4)             G = nx.Graph()
(4)             edgelist = [
(8)                 (0, 1, [("weight", 5), ("otherattr", 1), ("otherattr2", 3)]),
(8)                 (0, 2, [("weight", 5), ("otherattr", 2), ("otherattr2", 2)]),
(8)                 (1, 2, [("weight", 6), ("otherattr", 3), ("otherattr2", 1)]),
(4)             ]
(4)             G.add_edges_from(edgelist)
(4)             B = branchings.maximum_branching(G, preserve_attrs=False)
(4)             edge_dict = B[0][1]
(4)             with pytest.raises(KeyError):
(8)                 _ = edge_dict["otherattr"]
(0)         def test_partition_spanning_arborescence():
(4)             """
(4)             Test that we can generate minimum spanning arborescences which respect the
(4)             given partition.
(4)             """
(4)             G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)             G[3][0]["partition"] = nx.EdgePartition.EXCLUDED
(4)             G[2][3]["partition"] = nx.EdgePartition.INCLUDED
(4)             G[7][3]["partition"] = nx.EdgePartition.EXCLUDED
(4)             G[0][2]["partition"] = nx.EdgePartition.EXCLUDED
(4)             G[6][2]["partition"] = nx.EdgePartition.INCLUDED
(4)             actual_edges = [
(8)                 (0, 4, 12),
(8)                 (1, 0, 4),
(8)                 (1, 5, 13),
(8)                 (2, 3, 21),
(8)                 (4, 7, 12),
(8)                 (5, 6, 14),
(8)                 (5, 8, 12),
(8)                 (6, 2, 21),
(4)             ]
(4)             B = branchings.minimum_spanning_arborescence(G, partition="partition")
(4)             assert_equal_branchings(build_branching(actual_edges), B)
(0)         def test_arborescence_iterator_min():
(4)             """
(4)             Tests the arborescence iterator.
(4)             A brute force method found 680 arborescences in this graph.
(4)             This test will not verify all of them individually, but will check two
(4)             things
(4)             * The iterator returns 680 arborescences
(4)             * The weight of the arborescences is non-strictly increasing
(4)             for more information please visit
(4)             https://mjschwenne.github.io/2021/06/10/implementing-the-iterators.html
(4)             """
(4)             G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
(4)             arborescence_count = 0
(4)             arborescence_weight = -math.inf
(4)             for B in branchings.ArborescenceIterator(G):
(8)                 arborescence_count += 1
(8)                 new_arborescence_weight = B.size(weight="weight")
(8)                 assert new_arborescence_weight >= arborescence_weight
(8)                 arborescence_weight = new_arborescence_weight
(4)             assert arborescence_count == 680
(0)         def test_arborescence_iterator_max():
```

```
406: (4)                """
407: (4)                Tests the arborescence iterator.
408: (4)                A brute force method found 680 arborescences in this graph.
409: (4)                This test will not verify all of them individually, but will check two
410: (4)                things
411: (4)                * The iterator returns 680 arborescences
412: (4)                * The weight of the arborescences is non-strictly decreasing
413: (4)                for more information please visit
414: (4)                https://mjschwenne.github.io/2021/06/10/implementing-the-iterators.html
415: (4)                """
416: (4)                G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
417: (4)                arborescence_count = 0
418: (4)                arborescence_weight = math.inf
419: (4)                for B in branchings.ArborescenceIterator(G, minimum=False):
420: (8)                    arborescence_count += 1
421: (8)                    new_arborescence_weight = B.size(weight="weight")
422: (8)                    assert new_arborescence_weight <= arborescence_weight
423: (8)                    arborescence_weight = new_arborescence_weight
424: (4)                assert arborescence_count == 680
425: (0)            def test_arborescence_iterator_initial_partition():
426: (4)                """
427: (4)                Tests the arborescence iterator with three included edges and three
excluded
428: (4)                in the initial partition.
429: (4)                A brute force method similar to the one used in the above tests found that
430: (4)                there are 16 arborescences which contain the included edges and not the
431: (4)                excluded edges.
432: (4)                """
433: (4)                G = nx.from_numpy_array(G_array, create_using=nx.DiGraph)
434: (4)                included_edges = [(1, 0), (5, 6), (8, 7)]
435: (4)                excluded_edges = [(0, 2), (3, 6), (1, 5)]
436: (4)                arborescence_count = 0
437: (4)                arborescence_weight = -math.inf
438: (4)                for B in branchings.ArborescenceIterator(
439: (8)                    G, init_partition=(included_edges, excluded_edges)
440: (4)                ):
441: (8)                    arborescence_count += 1
442: (8)                    new_arborescence_weight = B.size(weight="weight")
443: (8)                    assert new_arborescence_weight >= arborescence_weight
444: (8)                    arborescence_weight = new_arborescence_weight
445: (8)                    for e in included_edges:
446: (12)                       assert e in B.edges
447: (8)                    for e in excluded_edges:
448: (12)                       assert e not in B.edges
449: (4)                assert arborescence_count == 16
450: (0)            def test_branchings_with_default_weights():
451: (4)                """
452: (4)                Tests that various brancing algorithms work on graphs without weights.
453: (4)                For more information, see issue #7279.
454: (4)                """
455: (4)                graph = nx.erdos_renyi_graph(10, p=0.2, directed=True, seed=123)
456: (4)                assert all(
457: (8)                    "weight" not in d for (u, v, d) in graph.edges(data=True)
458: (4)                ), "test is for graphs without a weight attribute"
459: (4)                nx.minimum_spanning_arborescence(graph.copy())
460: (4)                nx.maximum_spanning_arborescence(graph.copy())
461: (4)                nx.minimum_branching(graph.copy())
462: (4)                nx.maximum_branching(graph.copy())
463: (4)                nx.algorithms.tree.minimal_branching(graph.copy())
464: (4)                nx.algorithms.tree.branching_weight(graph.copy())
465: (4)                nx.algorithms.tree.greedy_branching(graph.copy())
466: (4)                assert all(
467: (8)                    "weight" not in d for (u, v, d) in graph.edges(data=True)
468: (4)                ), "The above calls should not modify the initial graph in-place"

----------------------------------------

File 384 - test_decomposition.py:
```

```
1: (0)              import networkx as nx
2: (0)              from networkx.algorithms.tree.decomposition import junction_tree
3: (0)              def test_junction_tree_directed_confounders():
4: (4)                  B = nx.DiGraph()
5: (4)                  B.add_edges_from([("A", "C"), ("B", "C"), ("C", "D"), ("C", "E")])
6: (4)                  G = junction_tree(B)
7: (4)                  J = nx.Graph()
8: (4)                  J.add_edges_from(
9: (8)                      [
10: (12)                        (("C", "E"), ("C",)),
11: (12)                        (("C",), ("A", "B", "C")),
12: (12)                        (("A", "B", "C"), ("C",)),
13: (12)                        (("C",), ("C", "D")),
14: (8)                     ]
15: (4)                 )
16: (4)                 assert nx.is_isomorphic(G, J)
17: (0)             def test_junction_tree_directed_unconnected_nodes():
18: (4)                 B = nx.DiGraph()
19: (4)                 B.add_nodes_from([("A", "B", "C", "D")])
20: (4)                 G = junction_tree(B)
21: (4)                 J = nx.Graph()
22: (4)                 J.add_nodes_from([("A", "B", "C", "D")])
23: (4)                 assert nx.is_isomorphic(G, J)
24: (0)             def test_junction_tree_directed_cascade():
25: (4)                 B = nx.DiGraph()
26: (4)                 B.add_edges_from([("A", "B"), ("B", "C"), ("C", "D")])
27: (4)                 G = junction_tree(B)
28: (4)                 J = nx.Graph()
29: (4)                 J.add_edges_from(
30: (8)                     [
31: (12)                        (("A", "B"), ("B",)),
32: (12)                        (("B",), ("B", "C")),
33: (12)                        (("B", "C"), ("C",)),
34: (12)                        (("C",), ("C", "D")),
35: (8)                     ]
36: (4)                 )
37: (4)                 assert nx.is_isomorphic(G, J)
38: (0)             def test_junction_tree_directed_unconnected_edges():
39: (4)                 B = nx.DiGraph()
40: (4)                 B.add_edges_from([("A", "B"), ("C", "D"), ("E", "F")])
41: (4)                 G = junction_tree(B)
42: (4)                 J = nx.Graph()
43: (4)                 J.add_nodes_from([("A", "B"), ("C", "D"), ("E", "F")])
44: (4)                 assert nx.is_isomorphic(G, J)
45: (0)             def test_junction_tree_undirected():
46: (4)                 B = nx.Graph()
47: (4)                 B.add_edges_from([("A", "C"), ("A", "D"), ("B", "C"), ("C", "E")])
48: (4)                 G = junction_tree(B)
49: (4)                 J = nx.Graph()
50: (4)                 J.add_edges_from(
51: (8)                     [
52: (12)                        (("A", "D"), ("A",)),
53: (12)                        (("A",), ("A", "C")),
54: (12)                        (("A", "C"), ("C",)),
55: (12)                        (("C",), ("B", "C")),
56: (12)                        (("B", "C"), ("C",)),
57: (12)                        (("C",), ("C", "E")),
58: (8)                     ]
59: (4)                 )
60: (4)                 assert nx.is_isomorphic(G, J)
```

File 385 - test_mst.py:

```
1: (0)              """Unit tests for the :mod:`networkx.algorithms.tree.mst` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import edges_equal, nodes_equal
```

```
5: (0)              def test_unknown_algorithm():
6: (4)                  with pytest.raises(ValueError):
7: (8)                      nx.minimum_spanning_tree(nx.Graph(), algorithm="random")
8: (4)                  with pytest.raises(
9: (8)                      ValueError, match="random is not a valid choice for an algorithm."
10: (4)                 ):
11: (8)                     nx.maximum_spanning_edges(nx.Graph(), algorithm="random")
12: (0)             class MinimumSpanningTreeTestBase:
13: (4)                 """Base class for test classes for minimum spanning tree algorithms.
14: (4)                 This class contains some common tests that will be inherited by
15: (4)                 subclasses. Each subclass must have a class attribute
16: (4)                 :data:`algorithm` that is a string representing the algorithm to
17: (4)                 run, as described under the ``algorithm`` keyword argument for the
18: (4)                 :func:`networkx.minimum_spanning_edges` function.  Subclasses can
19: (4)                 then implement any algorithm-specific tests.
20: (4)                 """
21: (4)                 def setup_method(self, method):
22: (8)                     """Creates an example graph and stores the expected minimum and
23: (8)                     maximum spanning tree edges.
24: (8)                     """
25: (8)                     self.algo = self.algorithm
26: (8)                     edges = [
27: (12)                        (0, 1, 7),
28: (12)                        (0, 3, 5),
29: (12)                        (1, 2, 8),
30: (12)                        (1, 3, 9),
31: (12)                        (1, 4, 7),
32: (12)                        (2, 4, 5),
33: (12)                        (3, 4, 15),
34: (12)                        (3, 5, 6),
35: (12)                        (4, 5, 8),
36: (12)                        (4, 6, 9),
37: (12)                        (5, 6, 11),
38: (8)                     ]
39: (8)                     self.G = nx.Graph()
40: (8)                     self.G.add_weighted_edges_from(edges)
41: (8)                     self.minimum_spanning_edgelist = [
42: (12)                        (0, 1, {"weight": 7}),
43: (12)                        (0, 3, {"weight": 5}),
44: (12)                        (1, 4, {"weight": 7}),
45: (12)                        (2, 4, {"weight": 5}),
46: (12)                        (3, 5, {"weight": 6}),
47: (12)                        (4, 6, {"weight": 9}),
48: (8)                     ]
49: (8)                     self.maximum_spanning_edgelist = [
50: (12)                        (0, 1, {"weight": 7}),
51: (12)                        (1, 2, {"weight": 8}),
52: (12)                        (1, 3, {"weight": 9}),
53: (12)                        (3, 4, {"weight": 15}),
54: (12)                        (4, 6, {"weight": 9}),
55: (12)                        (5, 6, {"weight": 11}),
56: (8)                     ]
57: (4)                 def test_minimum_edges(self):
58: (8)                     edges = nx.minimum_spanning_edges(self.G, algorithm=self.algo)
59: (8)                     actual = sorted((min(u, v), max(u, v), d) for u, v, d in edges)
60: (8)                     assert edges_equal(actual, self.minimum_spanning_edgelist)
61: (4)                 def test_maximum_edges(self):
62: (8)                     edges = nx.maximum_spanning_edges(self.G, algorithm=self.algo)
63: (8)                     actual = sorted((min(u, v), max(u, v), d) for u, v, d in edges)
64: (8)                     assert edges_equal(actual, self.maximum_spanning_edgelist)
65: (4)                 def test_without_data(self):
66: (8)                     edges = nx.minimum_spanning_edges(self.G, algorithm=self.algo,
data=False)
67: (8)                     actual = sorted((min(u, v), max(u, v)) for u, v in edges)
68: (8)                     expected = [(u, v) for u, v, d in self.minimum_spanning_edgelist]
69: (8)                     assert edges_equal(actual, expected)
70: (4)                 def test_nan_weights(self):
71: (8)                     G = self.G
72: (8)                     G.add_edge(0, 12, weight=float("nan"))
```

```
(8)                     edges = nx.minimum_spanning_edges(
(12)                        G, algorithm=self.algo, data=False, ignore_nan=True
(8)                     )
(8)                     actual = sorted((min(u, v), max(u, v)) for u, v in edges)
(8)                     expected = [(u, v) for u, v, d in self.minimum_spanning_edgelist]
(8)                     assert edges_equal(actual, expected)
(8)                     edges = nx.minimum_spanning_edges(
(12)                        G, algorithm=self.algo, data=False, ignore_nan=False
(8)                     )
(8)                     with pytest.raises(ValueError):
(12)                        list(edges)
(8)                     edges = nx.minimum_spanning_edges(G, algorithm=self.algo, data=False)
(8)                     with pytest.raises(ValueError):
(12)                        list(edges)
(4)                 def test_nan_weights_MultiGraph(self):
(8)                     G = nx.MultiGraph()
(8)                     G.add_edge(0, 12, weight=float("nan"))
(8)                     edges = nx.minimum_spanning_edges(
(12)                        G, algorithm="prim", data=False, ignore_nan=False
(8)                     )
(8)                     with pytest.raises(ValueError):
(12)                        list(edges)
(8)                     edges = nx.minimum_spanning_edges(G, algorithm="prim", data=False)
(8)                     with pytest.raises(ValueError):
(12)                        list(edges)
(4)                 def test_nan_weights_order(self):
(8)                     edges = [
(12)                        (0, 1, 7),
(12)                        (0, 3, 5),
(12)                        (1, 2, 8),
(12)                        (1, 3, 9),
(12)                        (1, 4, 7),
(12)                        (2, 4, 5),
(12)                        (3, 4, 15),
(12)                        (3, 5, 6),
(12)                        (4, 5, 8),
(12)                        (4, 6, 9),
(12)                        (5, 6, 11),
(8)                     ]
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from([(u + 1, v + 1, wt) for u, v, wt in edges])
(8)                     G.add_edge(0, 7, weight=float("nan"))
(8)                     edges = nx.minimum_spanning_edges(
(12)                        G, algorithm=self.algo, data=False, ignore_nan=True
(8)                     )
(8)                     actual = sorted((min(u, v), max(u, v)) for u, v in edges)
(8)                     shift = [(u + 1, v + 1) for u, v, d in self.minimum_spanning_edgelist]
(8)                     assert edges_equal(actual, shift)
(4)                 def test_isolated_node(self):
(8)                     edges = [
(12)                        (0, 1, 7),
(12)                        (0, 3, 5),
(12)                        (1, 2, 8),
(12)                        (1, 3, 9),
(12)                        (1, 4, 7),
(12)                        (2, 4, 5),
(12)                        (3, 4, 15),
(12)                        (3, 5, 6),
(12)                        (4, 5, 8),
(12)                        (4, 6, 9),
(12)                        (5, 6, 11),
(8)                     ]
(8)                     G = nx.Graph()
(8)                     G.add_weighted_edges_from([(u + 1, v + 1, wt) for u, v, wt in edges])
(8)                     G.add_node(0)
(8)                     edges = nx.minimum_spanning_edges(
(12)                        G, algorithm=self.algo, data=False, ignore_nan=True
(8)                     )
(8)                     actual = sorted((min(u, v), max(u, v)) for u, v in edges)
```

```
142: (8)                    shift = [(u + 1, v + 1) for u, v, d in self.minimum_spanning_edgelist]
143: (8)                    assert edges_equal(actual, shift)
144: (4)                def test_minimum_tree(self):
145: (8)                    T = nx.minimum_spanning_tree(self.G, algorithm=self.algo)
146: (8)                    actual = sorted(T.edges(data=True))
147: (8)                    assert edges_equal(actual, self.minimum_spanning_edgelist)
148: (4)                def test_maximum_tree(self):
149: (8)                    T = nx.maximum_spanning_tree(self.G, algorithm=self.algo)
150: (8)                    actual = sorted(T.edges(data=True))
151: (8)                    assert edges_equal(actual, self.maximum_spanning_edgelist)
152: (4)                def test_disconnected(self):
153: (8)                    G = nx.Graph([(0, 1, {"weight": 1}), (2, 3, {"weight": 2})])
154: (8)                    T = nx.minimum_spanning_tree(G, algorithm=self.algo)
155: (8)                    assert nodes_equal(list(T), list(range(4)))
156: (8)                    assert edges_equal(list(T.edges()), [(0, 1), (2, 3)])
157: (4)                def test_empty_graph(self):
158: (8)                    G = nx.empty_graph(3)
159: (8)                    T = nx.minimum_spanning_tree(G, algorithm=self.algo)
160: (8)                    assert nodes_equal(sorted(T), list(range(3)))
161: (8)                    assert T.number_of_edges() == 0
162: (4)                def test_attributes(self):
163: (8)                    G = nx.Graph()
164: (8)                    G.add_edge(1, 2, weight=1, color="red", distance=7)
165: (8)                    G.add_edge(2, 3, weight=1, color="green", distance=2)
166: (8)                    G.add_edge(1, 3, weight=10, color="blue", distance=1)
167: (8)                    G.graph["foo"] = "bar"
168: (8)                    T = nx.minimum_spanning_tree(G, algorithm=self.algo)
169: (8)                    assert T.graph == G.graph
170: (8)                    assert nodes_equal(T, G)
171: (8)                    for u, v in T.edges():
172: (12)                       assert T.adj[u][v] == G.adj[u][v]
173: (4)                def test_weight_attribute(self):
174: (8)                    G = nx.Graph()
175: (8)                    G.add_edge(0, 1, weight=1, distance=7)
176: (8)                    G.add_edge(0, 2, weight=30, distance=1)
177: (8)                    G.add_edge(1, 2, weight=1, distance=1)
178: (8)                    G.add_node(3)
179: (8)                    T = nx.minimum_spanning_tree(G, algorithm=self.algo,
weight="distance")
180: (8)                    assert nodes_equal(sorted(T), list(range(4)))
181: (8)                    assert edges_equal(sorted(T.edges()), [(0, 2), (1, 2)])
182: (8)                    T = nx.maximum_spanning_tree(G, algorithm=self.algo,
weight="distance")
183: (8)                    assert nodes_equal(sorted(T), list(range(4)))
184: (8)                    assert edges_equal(sorted(T.edges()), [(0, 1), (0, 2)])
185: (0)            class TestBoruvka(MinimumSpanningTreeTestBase):
186: (4)                """Unit tests for computing a minimum (or maximum) spanning tree
187: (4)                using Borůvka's algorithm.
188: (4)                """
189: (4)                algorithm = "boruvka"
190: (4)                def test_unicode_name(self):
191: (8)                    """Tests that using a Unicode string can correctly indicate
192: (8)                    Borůvka's algorithm.
193: (8)                    """
194: (8)                    edges = nx.minimum_spanning_edges(self.G, algorithm="borůvka")
195: (8)                    actual = sorted((min(u, v), max(u, v), d) for u, v, d in edges)
196: (8)                    assert edges_equal(actual, self.minimum_spanning_edgelist)
197: (0)            class MultigraphMSTTestBase(MinimumSpanningTreeTestBase):
198: (4)                def test_multigraph_keys_min(self):
199: (8)                    """Tests that the minimum spanning edges of a multigraph
200: (8)                    preserves edge keys.
201: (8)                    """
202: (8)                    G = nx.MultiGraph()
203: (8)                    G.add_edge(0, 1, key="a", weight=2)
204: (8)                    G.add_edge(0, 1, key="b", weight=1)
205: (8)                    min_edges = nx.minimum_spanning_edges
206: (8)                    mst_edges = min_edges(G, algorithm=self.algo, data=False)
207: (8)                    assert edges_equal([(0, 1, "b")], list(mst_edges))
208: (4)                def test_multigraph_keys_max(self):
```

```
209: (8)                    """Tests that the maximum spanning edges of a multigraph
210: (8)                    preserves edge keys.
211: (8)                    """
212: (8)                    G = nx.MultiGraph()
213: (8)                    G.add_edge(0, 1, key="a", weight=2)
214: (8)                    G.add_edge(0, 1, key="b", weight=1)
215: (8)                    max_edges = nx.maximum_spanning_edges
216: (8)                    mst_edges = max_edges(G, algorithm=self.algo, data=False)
217: (8)                    assert edges_equal([(0, 1, "a")], list(mst_edges))
218: (0)            class TestKruskal(MultigraphMSTTestBase):
219: (4)                """Unit tests for computing a minimum (or maximum) spanning tree
220: (4)                using Kruskal's algorithm.
221: (4)                """
222: (4)                algorithm = "kruskal"
223: (4)                def test_key_data_bool(self):
224: (8)                    """Tests that the keys and data values are included in
225: (8)                    MST edges based on whether keys and data parameters are
226: (8)                    true or false"""
227: (8)                    G = nx.MultiGraph()
228: (8)                    G.add_edge(1, 2, key=1, weight=2)
229: (8)                    G.add_edge(1, 2, key=2, weight=3)
230: (8)                    G.add_edge(3, 2, key=1, weight=2)
231: (8)                    G.add_edge(3, 1, key=1, weight=4)
232: (8)                    mst_edges = nx.minimum_spanning_edges(
233: (12)                       G, algorithm=self.algo, keys=True, data=False
234: (8)                    )
235: (8)                    assert edges_equal([(1, 2, 1), (2, 3, 1)], list(mst_edges))
236: (8)                    mst_edges = nx.minimum_spanning_edges(
237: (12)                       G, algorithm=self.algo, keys=False, data=True
238: (8)                    )
239: (8)                    assert edges_equal(
240: (12)                       [(1, 2, {"weight": 2}), (2, 3, {"weight": 2})], list(mst_edges)
241: (8)                    )
242: (8)                    mst_edges = nx.minimum_spanning_edges(
243: (12)                       G, algorithm=self.algo, keys=False, data=False
244: (8)                    )
245: (8)                    assert edges_equal([(1, 2), (2, 3)], list(mst_edges))
246: (8)                    mst_edges = nx.minimum_spanning_edges(
247: (12)                       G, algorithm=self.algo, keys=True, data=True
248: (8)                    )
249: (8)                    assert edges_equal(
250: (12)                       [(1, 2, 1, {"weight": 2}), (2, 3, 1, {"weight": 2})],
list(mst_edges)
251: (8)                    )
252: (0)            class TestPrim(MultigraphMSTTestBase):
253: (4)                """Unit tests for computing a minimum (or maximum) spanning tree
254: (4)                using Prim's algorithm.
255: (4)                """
256: (4)                algorithm = "prim"
257: (4)                def test_prim_mst_edges_simple_graph(self):
258: (8)                    H = nx.Graph()
259: (8)                    H.add_edge(1, 2, key=2, weight=3)
260: (8)                    H.add_edge(3, 2, key=1, weight=2)
261: (8)                    H.add_edge(3, 1, key=1, weight=4)
262: (8)                    mst_edges = nx.minimum_spanning_edges(H, algorithm=self.algo,
ignore_nan=True)
263: (8)                    assert edges_equal(
264: (12)                       [(1, 2, {"key": 2, "weight": 3}), (2, 3, {"key": 1, "weight":
2})],
265: (12)                       list(mst_edges),
266: (8)                    )
267: (4)                def test_ignore_nan(self):
268: (8)                    """Tests that the edges with NaN weights are ignored or
269: (8)                    raise an Error based on ignore_nan is true or false"""
270: (8)                    H = nx.MultiGraph()
271: (8)                    H.add_edge(1, 2, key=1, weight=float("nan"))
272: (8)                    H.add_edge(1, 2, key=2, weight=3)
273: (8)                    H.add_edge(3, 2, key=1, weight=2)
274: (8)                    H.add_edge(3, 1, key=1, weight=4)
```

```
275: (8)                    mst_edges = nx.minimum_spanning_edges(H, algorithm=self.algo,
ignore_nan=True)
276: (8)                    assert edges_equal(
277: (12)                       [(1, 2, 2, {"weight": 3}), (2, 3, 1, {"weight": 2})],
list(mst_edges)
278: (8)                    )
279: (8)                    with pytest.raises(ValueError):
280: (12)                       list(nx.minimum_spanning_edges(H, algorithm=self.algo,
ignore_nan=False))
281: (4)                def test_multigraph_keys_tree(self):
282: (8)                    G = nx.MultiGraph()
283: (8)                    G.add_edge(0, 1, key="a", weight=2)
284: (8)                    G.add_edge(0, 1, key="b", weight=1)
285: (8)                    T = nx.minimum_spanning_tree(G, algorithm=self.algo)
286: (8)                    assert edges_equal([(0, 1, 1)], list(T.edges(data="weight")))
287: (4)                def test_multigraph_keys_tree_max(self):
288: (8)                    G = nx.MultiGraph()
289: (8)                    G.add_edge(0, 1, key="a", weight=2)
290: (8)                    G.add_edge(0, 1, key="b", weight=1)
291: (8)                    T = nx.maximum_spanning_tree(G, algorithm=self.algo)
292: (8)                    assert edges_equal([(0, 1, 2)], list(T.edges(data="weight")))
293: (0)            class TestSpanningTreeIterator:
294: (4)                """
295: (4)                Tests the spanning tree iterator on the example graph in the 2005 Sörensen
296: (4)                and Janssens paper An Algorithm to Generate all Spanning Trees of a Graph
in
297: (4)                Order of Increasing Cost
298: (4)                """
299: (4)                def setup_method(self):
300: (8)                    edges = [(0, 1, 5), (1, 2, 4), (1, 4, 6), (2, 3, 5), (2, 4, 7), (3, 4,
3)]
301: (8)                    self.G = nx.Graph()
302: (8)                    self.G.add_weighted_edges_from(edges)
303: (8)                    self.spanning_trees = [
304: (12)                       [
305: (16)                           (0, 1, {"weight": 5}),
306: (16)                           (1, 2, {"weight": 4}),
307: (16)                           (2, 3, {"weight": 5}),
308: (16)                           (3, 4, {"weight": 3}),
309: (12)                       ],
310: (12)                       [
311: (16)                           (0, 1, {"weight": 5}),
312: (16)                           (1, 2, {"weight": 4}),
313: (16)                           (1, 4, {"weight": 6}),
314: (16)                           (3, 4, {"weight": 3}),
315: (12)                       ],
316: (12)                       [
317: (16)                           (0, 1, {"weight": 5}),
318: (16)                           (1, 4, {"weight": 6}),
319: (16)                           (2, 3, {"weight": 5}),
320: (16)                           (3, 4, {"weight": 3}),
321: (12)                       ],
322: (12)                       [
323: (16)                           (0, 1, {"weight": 5}),
324: (16)                           (1, 2, {"weight": 4}),
325: (16)                           (2, 4, {"weight": 7}),
326: (16)                           (3, 4, {"weight": 3}),
327: (12)                       ],
328: (12)                       [
329: (16)                           (0, 1, {"weight": 5}),
330: (16)                           (1, 2, {"weight": 4}),
331: (16)                           (1, 4, {"weight": 6}),
332: (16)                           (2, 3, {"weight": 5}),
333: (12)                       ],
334: (12)                       [
335: (16)                           (0, 1, {"weight": 5}),
336: (16)                           (1, 4, {"weight": 6}),
337: (16)                           (2, 4, {"weight": 7}),
338: (16)                           (3, 4, {"weight": 3}),
```

```
339: (12)                        ],
340: (12)                        [
341: (16)                            (0, 1, {"weight": 5}),
342: (16)                            (1, 2, {"weight": 4}),
343: (16)                            (2, 3, {"weight": 5}),
344: (16)                            (2, 4, {"weight": 7}),
345: (12)                        ],
346: (12)                        [
347: (16)                            (0, 1, {"weight": 5}),
348: (16)                            (1, 4, {"weight": 6}),
349: (16)                            (2, 3, {"weight": 5}),
350: (16)                            (2, 4, {"weight": 7}),
351: (12)                        ],
352: (8)                    ]
353: (4)                def test_minimum_spanning_tree_iterator(self):
354: (8)                    """
355: (8)                    Tests that the spanning trees are correctly returned in increasing
order
356: (8)                    """
357: (8)                    tree_index = 0
358: (8)                    for tree in nx.SpanningTreeIterator(self.G):
359: (12)                       actual = sorted(tree.edges(data=True))
360: (12)                       assert edges_equal(actual, self.spanning_trees[tree_index])
361: (12)                       tree_index += 1
362: (4)                def test_maximum_spanning_tree_iterator(self):
363: (8)                    """
364: (8)                    Tests that the spanning trees are correctly returned in decreasing
order
365: (8)                    """
366: (8)                    tree_index = 7
367: (8)                    for tree in nx.SpanningTreeIterator(self.G, minimum=False):
368: (12)                       actual = sorted(tree.edges(data=True))
369: (12)                       assert edges_equal(actual, self.spanning_trees[tree_index])
370: (12)                       tree_index -= 1
371: (0)            def test_random_spanning_tree_multiplicative_small():
372: (4)                """
373: (4)                Using a fixed seed, sample one tree for repeatability.
374: (4)                """
375: (4)                from math import exp
376: (4)                pytest.importorskip("scipy")
377: (4)                gamma = {
378: (8)                    (0, 1): -0.6383,
379: (8)                    (0, 2): -0.6827,
380: (8)                    (0, 5): 0,
381: (8)                    (1, 2): -1.0781,
382: (8)                    (1, 4): 0,
383: (8)                    (2, 3): 0,
384: (8)                    (5, 3): -0.2820,
385: (8)                    (5, 4): -0.3327,
386: (8)                    (4, 3): -0.9927,
387: (4)                }
388: (4)                G = nx.Graph()
389: (4)                for u, v in gamma:
390: (8)                    G.add_edge(u, v, lambda_key=exp(gamma[(u, v)]))
391: (4)                solution_edges = [(2, 3), (3, 4), (0, 5), (5, 4), (4, 1)]
392: (4)                solution = nx.Graph()
393: (4)                solution.add_edges_from(solution_edges)
394: (4)                sampled_tree = nx.random_spanning_tree(G, "lambda_key", seed=42)
395: (4)                assert nx.utils.edges_equal(solution.edges, sampled_tree.edges)
396: (0)            @pytest.mark.slow
397: (0)            def test_random_spanning_tree_multiplicative_large():
398: (4)                """
399: (4)                Sample many trees from the distribution created in the last test
400: (4)                """
401: (4)                from math import exp
402: (4)                from random import Random
403: (4)                pytest.importorskip("numpy")
404: (4)                stats = pytest.importorskip("scipy.stats")
405: (4)                gamma = {
```

```
(8)                     (0, 1): -0.6383,
(8)                     (0, 2): -0.6827,
(8)                     (0, 5): 0,
(8)                     (1, 2): -1.0781,
(8)                     (1, 4): 0,
(8)                     (2, 3): 0,
(8)                     (5, 3): -0.2820,
(8)                     (5, 4): -0.3327,
(8)                     (4, 3): -0.9927,
(4)                 }
(4)                 G = nx.Graph()
(4)                 for u, v in gamma:
(8)                     G.add_edge(u, v, lambda_key=exp(gamma[(u, v)]))
(4)                 total_weight = 0
(4)                 tree_expected = {}
(4)                 for t in nx.SpanningTreeIterator(G):
(8)                     weight = 1
(8)                     for u, v, d in t.edges(data="lambda_key"):
(12)                        weight *= d
(8)                     tree_expected[t] = weight
(8)                     total_weight += weight
(4)                 assert len(tree_expected) == 75
(4)                 sample_size = 1200
(4)                 tree_actual = {}
(4)                 for t in tree_expected:
(8)                     tree_expected[t] = (tree_expected[t] / total_weight) * sample_size
(8)                     tree_actual[t] = 0
(4)                 rng = Random(37)
(4)                 for _ in range(sample_size):
(8)                     sampled_tree = nx.random_spanning_tree(G, "lambda_key", seed=rng)
(8)                     assert nx.is_tree(sampled_tree)
(8)                     for t in tree_expected:
(12)                        if nx.utils.edges_equal(t.edges, sampled_tree.edges):
(16)                            tree_actual[t] += 1
(16)                            break
(4)                 _, p = stats.chisquare(list(tree_actual.values()),
list(tree_expected.values()))
(4)                 assert not p < 0.05
(0)             def test_random_spanning_tree_additive_small():
(4)                 """
(4)                 Sample a single spanning tree from the additive method.
(4)                 """
(4)                 pytest.importorskip("scipy")
(4)                 edges = {
(8)                     (0, 1): 1,
(8)                     (0, 2): 1,
(8)                     (0, 5): 3,
(8)                     (1, 2): 2,
(8)                     (1, 4): 3,
(8)                     (2, 3): 3,
(8)                     (5, 3): 4,
(8)                     (5, 4): 5,
(8)                     (4, 3): 4,
(4)                 }
(4)                 G = nx.Graph()
(4)                 for u, v in edges:
(8)                     G.add_edge(u, v, weight=edges[(u, v)])
(4)                 solution_edges = [(0, 2), (1, 2), (2, 3), (3, 4), (3, 5)]
(4)                 solution = nx.Graph()
(4)                 solution.add_edges_from(solution_edges)
(4)                 sampled_tree = nx.random_spanning_tree(
(8)                     G, weight="weight", multiplicative=False, seed=37
(4)                 )
(4)                 assert nx.utils.edges_equal(solution.edges, sampled_tree.edges)
(0)             @pytest.mark.slow
(0)             def test_random_spanning_tree_additive_large():
(4)                 """
(4)                 Sample many spanning trees from the additive method.
(4)                 """
```

```
(4)                 from random import Random
(4)                 pytest.importorskip("numpy")
(4)                 stats = pytest.importorskip("scipy.stats")
(4)                 edges = {
(8)                     (0, 1): 1,
(8)                     (0, 2): 1,
(8)                     (0, 5): 3,
(8)                     (1, 2): 2,
(8)                     (1, 4): 3,
(8)                     (2, 3): 3,
(8)                     (5, 3): 4,
(8)                     (5, 4): 5,
(8)                     (4, 3): 4,
(4)                 }
(4)                 G = nx.Graph()
(4)                 for u, v in edges:
(8)                     G.add_edge(u, v, weight=edges[(u, v)])
(4)                 total_weight = 0
(4)                 tree_expected = {}
(4)                 for t in nx.SpanningTreeIterator(G):
(8)                     weight = 0
(8)                     for u, v, d in t.edges(data="weight"):
(12)                        weight += d
(8)                     tree_expected[t] = weight
(8)                     total_weight += weight
(4)                 assert len(tree_expected) == 75
(4)                 sample_size = 500
(4)                 tree_actual = {}
(4)                 for t in tree_expected:
(8)                     tree_expected[t] = (tree_expected[t] / total_weight) * sample_size
(8)                     tree_actual[t] = 0
(4)                 rng = Random(37)
(4)                 for _ in range(sample_size):
(8)                     sampled_tree = nx.random_spanning_tree(
(12)                        G, "weight", multiplicative=False, seed=rng
(8)                     )
(8)                     assert nx.is_tree(sampled_tree)
(8)                     for t in tree_expected:
(12)                        if nx.utils.edges_equal(t.edges, sampled_tree.edges):
(16)                            tree_actual[t] += 1
(16)                            break
(4)                 _, p = stats.chisquare(list(tree_actual.values()),
list(tree_expected.values()))
(4)                 assert not p < 0.05
(0)             def test_random_spanning_tree_empty_graph():
(4)                 G = nx.Graph()
(4)                 rst = nx.tree.random_spanning_tree(G)
(4)                 assert len(rst.nodes) == 0
(4)                 assert len(rst.edges) == 0
(0)             def test_random_spanning_tree_single_node_graph():
(4)                 G = nx.Graph()
(4)                 G.add_node(0)
(4)                 rst = nx.tree.random_spanning_tree(G)
(4)                 assert len(rst.nodes) == 1
(4)                 assert len(rst.edges) == 0
(0)             def test_random_spanning_tree_single_node_loop():
(4)                 G = nx.Graph()
(4)                 G.add_node(0)
(4)                 G.add_edge(0, 0)
(4)                 rst = nx.tree.random_spanning_tree(G)
(4)                 assert len(rst.nodes) == 1
(4)                 assert len(rst.edges) == 0
(0)             class TestNumberSpanningTrees:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     global np
(8)                     np = pytest.importorskip("numpy")
(8)                     sp = pytest.importorskip("scipy")
(4)                 def test_nst_disconnected(self):
```

```
(8)                 G = nx.empty_graph(2)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 0)
(4)             def test_nst_no_nodes(self):
(8)                 G = nx.Graph()
(8)                 with pytest.raises(nx.NetworkXPointlessConcept):
(12)                    nx.number_of_spanning_trees(G)
(4)             def test_nst_weight(self):
(8)                 G = nx.Graph()
(8)                 G.add_edge(1, 2, weight=1)
(8)                 G.add_edge(1, 3, weight=1)
(8)                 G.add_edge(2, 3, weight=2)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 3)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G, weight="weight"), 5)
(4)             def test_nst_negative_weight(self):
(8)                 G = nx.Graph()
(8)                 G.add_edge(1, 2, weight=1)
(8)                 G.add_edge(1, 3, weight=-1)
(8)                 G.add_edge(2, 3, weight=-2)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 3)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G, weight="weight"), -1)
(4)             def test_nst_selfloop(self):
(8)                 G = nx.complete_graph(3)
(8)                 G.add_edge(1, 1)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 3)
(4)             def test_nst_multigraph(self):
(8)                 G = nx.MultiGraph()
(8)                 G.add_edge(1, 2)
(8)                 G.add_edge(1, 2)
(8)                 G.add_edge(1, 3)
(8)                 G.add_edge(2, 3)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 5)
(4)             def test_nst_complete_graph(self):
(8)                 N = 5
(8)                 G = nx.complete_graph(N)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), N ** (N - 2))
(4)             def test_nst_path_graph(self):
(8)                 G = nx.path_graph(5)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 1)
(4)             def test_nst_cycle_graph(self):
(8)                 G = nx.cycle_graph(5)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G), 5)
(4)             def test_nst_directed_noroot(self):
(8)                 G = nx.empty_graph(3, create_using=nx.MultiDiGraph)
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    nx.number_of_spanning_trees(G)
(4)             def test_nst_directed_root_not_exist(self):
(8)                 G = nx.empty_graph(3, create_using=nx.MultiDiGraph)
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    nx.number_of_spanning_trees(G, root=42)
(4)             def test_nst_directed_not_weak_connected(self):
(8)                 G = nx.DiGraph()
(8)                 G.add_edge(1, 2)
(8)                 G.add_edge(3, 4)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G, root=1), 0)
(4)             def test_nst_directed_cycle_graph(self):
(8)                 G = nx.DiGraph()
(8)                 G = nx.cycle_graph(7, G)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G, root=0), 1)
(4)             def test_nst_directed_complete_graph(self):
(8)                 G = nx.DiGraph()
(8)                 G = nx.complete_graph(7, G)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G, root=0), 7**5)
(4)             def test_nst_directed_multi(self):
(8)                 G = nx.MultiDiGraph()
(8)                 G = nx.cycle_graph(3, G)
(8)                 G.add_edge(1, 2)
(8)                 assert np.isclose(nx.number_of_spanning_trees(G, root=0), 2)
(4)             def test_nst_directed_selfloop(self):
(8)                 G = nx.MultiDiGraph()
```

```
611: (8)                    G = nx.cycle_graph(3, G)
612: (8)                    G.add_edge(1, 1)
613: (8)                    assert np.isclose(nx.number_of_spanning_trees(G, root=0), 1)
614: (4)                def test_nst_directed_weak_connected(self):
615: (8)                    G = nx.MultiDiGraph()
616: (8)                    G = nx.cycle_graph(3, G)
617: (8)                    G.remove_edge(1, 2)
618: (8)                    assert np.isclose(nx.number_of_spanning_trees(G, root=0), 0)
619: (4)                def test_nst_directed_weighted(self):
620: (8)                    G = nx.DiGraph()
621: (8)                    G.add_edge(1, 2, weight=2)
622: (8)                    G.add_edge(1, 3, weight=1)
623: (8)                    G.add_edge(2, 3, weight=3)
624: (8)                    Nst = nx.number_of_spanning_trees(G, root=1, weight="weight")
625: (8)                    assert np.isclose(Nst, 8)
626: (8)                    Nst = nx.number_of_spanning_trees(G, root=2, weight="weight")
627: (8)                    assert np.isclose(Nst, 0)
628: (8)                    Nst = nx.number_of_spanning_trees(G, root=3, weight="weight")
629: (8)                    assert np.isclose(Nst, 0)

----------------------------------------

File 386 - test_operations.py:

1: (0)              from itertools import chain
2: (0)              import networkx as nx
3: (0)              from networkx.utils import edges_equal, nodes_equal
4: (0)              def _check_custom_label_attribute(input_trees, res_tree, label_attribute):
5: (4)                  res_attr_dict = nx.get_node_attributes(res_tree, label_attribute)
6: (4)                  res_attr_set = set(res_attr_dict.values())
7: (4)                  input_label = (tree for tree, root in input_trees)
8: (4)                  input_label_set = set(chain.from_iterable(input_label))
9: (4)                  return res_attr_set == input_label_set
10: (0)             def test_empty_sequence():
11: (4)                 """Joining the empty sequence results in the tree with one node."""
12: (4)                 T = nx.join_trees([])
13: (4)                 assert len(T) == 1
14: (4)                 assert T.number_of_edges() == 0
15: (0)             def test_single():
16: (4)                 """Joining just one tree yields a tree with one more node."""
17: (4)                 T = nx.empty_graph(1)
18: (4)                 trees = [(T, 0)]
19: (4)                 actual_with_label = nx.join_trees(trees, label_attribute="custom_label")
20: (4)                 expected = nx.path_graph(2)
21: (4)                 assert nodes_equal(list(expected), list(actual_with_label))
22: (4)                 assert edges_equal(list(expected.edges()),
list(actual_with_label.edges()))
23: (0)             def test_basic():
24: (4)                 """Joining multiple subtrees at a root node."""
25: (4)                 trees = [(nx.full_rary_tree(2, 2**2 - 1), 0) for i in range(2)]
26: (4)                 expected = nx.full_rary_tree(2, 2**3 - 1)
27: (4)                 actual = nx.join_trees(trees, label_attribute="old_labels")
28: (4)                 assert nx.is_isomorphic(actual, expected)
29: (4)                 assert _check_custom_label_attribute(trees, actual, "old_labels")
30: (4)                 actual_without_label = nx.join_trees(trees)
31: (4)                 assert nx.is_isomorphic(actual_without_label, expected)
32: (4)                 assert all(not data for _, data in actual_without_label.nodes(data=True))
33: (0)             def test_first_label():
34: (4)                 """Test the functionality of the first_label argument."""
35: (4)                 T1 = nx.path_graph(3)
36: (4)                 T2 = nx.path_graph(2)
37: (4)                 actual = nx.join_trees([(T1, 0), (T2, 0)], first_label=10)
38: (4)                 expected_nodes = set(range(10, 16))
39: (4)                 assert set(actual.nodes()) == expected_nodes
40: (4)                 assert set(actual.neighbors(10)) == {11, 14}

----------------------------------------

File 387 - test_recognition.py:
```

```
(0)  import pytest
(0)  import networkx as nx
(0)  class TestTreeRecognition:
(4)      graph = nx.Graph
(4)      multigraph = nx.MultiGraph
(4)      @classmethod
(4)      def setup_class(cls):
(8)          cls.T1 = cls.graph()
(8)          cls.T2 = cls.graph()
(8)          cls.T2.add_node(1)
(8)          cls.T3 = cls.graph()
(8)          cls.T3.add_nodes_from(range(5))
(8)          edges = [(i, i + 1) for i in range(4)]
(8)          cls.T3.add_edges_from(edges)
(8)          cls.T5 = cls.multigraph()
(8)          cls.T5.add_nodes_from(range(5))
(8)          edges = [(i, i + 1) for i in range(4)]
(8)          cls.T5.add_edges_from(edges)
(8)          cls.T6 = cls.graph()
(8)          cls.T6.add_nodes_from([6, 7])
(8)          cls.T6.add_edge(6, 7)
(8)          cls.F1 = nx.compose(cls.T6, cls.T3)
(8)          cls.N4 = cls.graph()
(8)          cls.N4.add_node(1)
(8)          cls.N4.add_edge(1, 1)
(8)          cls.N5 = cls.graph()
(8)          cls.N5.add_nodes_from(range(5))
(8)          cls.N6 = cls.graph()
(8)          cls.N6.add_nodes_from(range(3))
(8)          cls.N6.add_edges_from([(0, 1), (1, 2), (2, 0)])
(8)          cls.NF1 = nx.compose(cls.T6, cls.N6)
(4)      def test_null_tree(self):
(8)          with pytest.raises(nx.NetworkXPointlessConcept):
(12)             nx.is_tree(self.graph())
(4)      def test_null_tree2(self):
(8)          with pytest.raises(nx.NetworkXPointlessConcept):
(12)             nx.is_tree(self.multigraph())
(4)      def test_null_forest(self):
(8)          with pytest.raises(nx.NetworkXPointlessConcept):
(12)             nx.is_forest(self.graph())
(4)      def test_null_forest2(self):
(8)          with pytest.raises(nx.NetworkXPointlessConcept):
(12)             nx.is_forest(self.multigraph())
(4)      def test_is_tree(self):
(8)          assert nx.is_tree(self.T2)
(8)          assert nx.is_tree(self.T3)
(8)          assert nx.is_tree(self.T5)
(4)      def test_is_not_tree(self):
(8)          assert not nx.is_tree(self.N4)
(8)          assert not nx.is_tree(self.N5)
(8)          assert not nx.is_tree(self.N6)
(4)      def test_is_forest(self):
(8)          assert nx.is_forest(self.T2)
(8)          assert nx.is_forest(self.T3)
(8)          assert nx.is_forest(self.T5)
(8)          assert nx.is_forest(self.F1)
(8)          assert nx.is_forest(self.N5)
(4)      def test_is_not_forest(self):
(8)          assert not nx.is_forest(self.N4)
(8)          assert not nx.is_forest(self.N6)
(8)          assert not nx.is_forest(self.NF1)
(0)  class TestDirectedTreeRecognition(TestTreeRecognition):
(4)      graph = nx.DiGraph
(4)      multigraph = nx.MultiDiGraph
(0)  def test_disconnected_graph():
(4)      G = nx.Graph()
(4)      G.add_edges_from([(0, 1), (1, 2), (2, 0), (3, 4)])
(4)      assert not nx.is_tree(G)
```

```
69: (4)                 G = nx.DiGraph()
70: (4)                 G.add_edges_from([(0, 1), (1, 2), (2, 0), (3, 4)])
71: (4)                 assert not nx.is_tree(G)
72: (0)             def test_dag_nontree():
73: (4)                 G = nx.DiGraph()
74: (4)                 G.add_edges_from([(0, 1), (0, 2), (1, 2)])
75: (4)                 assert not nx.is_tree(G)
76: (4)                 assert nx.is_directed_acyclic_graph(G)
77: (0)             def test_multicycle():
78: (4)                 G = nx.MultiDiGraph()
79: (4)                 G.add_edges_from([(0, 1), (0, 1)])
80: (4)                 assert not nx.is_tree(G)
81: (4)                 assert nx.is_directed_acyclic_graph(G)
82: (0)             def test_emptybranch():
83: (4)                 G = nx.DiGraph()
84: (4)                 G.add_nodes_from(range(10))
85: (4)                 assert nx.is_branching(G)
86: (4)                 assert not nx.is_arborescence(G)
87: (0)             def test_is_branching_empty_graph_raises():
88: (4)                 G = nx.DiGraph()
89: (4)                 with pytest.raises(nx.NetworkXPointlessConcept, match="G has no nodes."):
90: (8)                     nx.is_branching(G)
91: (0)             def test_path():
92: (4)                 G = nx.DiGraph()
93: (4)                 nx.add_path(G, range(5))
94: (4)                 assert nx.is_branching(G)
95: (4)                 assert nx.is_arborescence(G)
96: (0)             def test_notbranching1():
97: (4)                 G = nx.MultiDiGraph()
98: (4)                 G.add_nodes_from(range(10))
99: (4)                 G.add_edges_from([(0, 1), (1, 0)])
100: (4)                assert not nx.is_branching(G)
101: (4)                assert not nx.is_arborescence(G)
102: (0)            def test_notbranching2():
103: (4)                G = nx.MultiDiGraph()
104: (4)                G.add_nodes_from(range(10))
105: (4)                G.add_edges_from([(0, 1), (0, 2), (3, 2)])
106: (4)                assert not nx.is_branching(G)
107: (4)                assert not nx.is_arborescence(G)
108: (0)            def test_notarborescence1():
109: (4)                G = nx.MultiDiGraph()
110: (4)                G.add_nodes_from(range(10))
111: (4)                G.add_edges_from([(0, 1), (0, 2), (1, 3), (5, 6)])
112: (4)                assert nx.is_branching(G)
113: (4)                assert not nx.is_arborescence(G)
114: (0)            def test_notarborescence2():
115: (4)                G = nx.MultiDiGraph()
116: (4)                nx.add_path(G, range(5))
117: (4)                G.add_edge(6, 4)
118: (4)                assert not nx.is_branching(G)
119: (4)                assert not nx.is_arborescence(G)
120: (0)            def test_is_arborescense_empty_graph_raises():
121: (4)                G = nx.DiGraph()
122: (4)                with pytest.raises(nx.NetworkXPointlessConcept, match="G has no nodes."):
123: (8)                    nx.is_arborescence(G)

----------------------------------------

File 388 - __init__.py:

1: (0)              from .graph import Graph
2: (0)              from .digraph import DiGraph
3: (0)              from .multigraph import MultiGraph
4: (0)              from .multidigraph import MultiDiGraph
5: (0)              from .function import *
6: (0)              from .graphviews import subgraph_view, reverse_view
7: (0)              from networkx.classes import filters
8: (0)              from networkx.classes import coreviews
9: (0)              from networkx.classes import graphviews
```

```
10: (0)             from networkx.classes import reportviews

----------------------------------------

File 389 - coreviews.py:

1: (0)              """Views of core data structures such as nested Mappings (e.g. dict-of-dicts).
2: (0)              These ``Views`` often restrict element access, with either the entire view or
3: (0)              layers of nested mappings being read-only.
4: (0)              """
5: (0)              from collections.abc import Mapping
6: (0)              __all__ = [
7: (4)                  "AtlasView",
8: (4)                  "AdjacencyView",
9: (4)                  "MultiAdjacencyView",
10: (4)                 "UnionAtlas",
11: (4)                 "UnionAdjacency",
12: (4)                 "UnionMultiInner",
13: (4)                 "UnionMultiAdjacency",
14: (4)                 "FilterAtlas",
15: (4)                 "FilterAdjacency",
16: (4)                 "FilterMultiInner",
17: (4)                 "FilterMultiAdjacency",
18: (0)             ]
19: (0)             class AtlasView(Mapping):
20: (4)                 """An AtlasView is a Read-only Mapping of Mappings.
21: (4)                 It is a View into a dict-of-dict data structure.
22: (4)                 The inner level of dict is read-write. But the
23: (4)                 outer level is read-only.
24: (4)                 See Also
25: (4)                 ========
26: (4)                 AdjacencyView: View into dict-of-dict-of-dict
27: (4)                 MultiAdjacencyView: View into dict-of-dict-of-dict-of-dict
28: (4)                 """
29: (4)                 __slots__ = ("_atlas",)
30: (4)                 def __getstate__(self):
31: (8)                     return {"_atlas": self._atlas}
32: (4)                 def __setstate__(self, state):
33: (8)                     self._atlas = state["_atlas"]
34: (4)                 def __init__(self, d):
35: (8)                     self._atlas = d
36: (4)                 def __len__(self):
37: (8)                     return len(self._atlas)
38: (4)                 def __iter__(self):
39: (8)                     return iter(self._atlas)
40: (4)                 def __getitem__(self, key):
41: (8)                     return self._atlas[key]
42: (4)                 def copy(self):
43: (8)                     return {n: self[n].copy() for n in self._atlas}
44: (4)                 def __str__(self):
45: (8)                     return str(self._atlas)  # {nbr: self[nbr] for nbr in self})
46: (4)                 def __repr__(self):
47: (8)                     return f"{self.__class__.__name__}({self._atlas!r})"
48: (0)             class AdjacencyView(AtlasView):
49: (4)                 """An AdjacencyView is a Read-only Map of Maps of Maps.
50: (4)                 It is a View into a dict-of-dict-of-dict data structure.
51: (4)                 The inner level of dict is read-write. But the
52: (4)                 outer levels are read-only.
53: (4)                 See Also
54: (4)                 ========
55: (4)                 AtlasView: View into dict-of-dict
56: (4)                 MultiAdjacencyView: View into dict-of-dict-of-dict-of-dict
57: (4)                 """
58: (4)                 __slots__ = ()  # Still uses AtlasView slots names _atlas
59: (4)                 def __getitem__(self, name):
60: (8)                     return AtlasView(self._atlas[name])
61: (4)                 def copy(self):
62: (8)                     return {n: self[n].copy() for n in self._atlas}
63: (0)             class MultiAdjacencyView(AdjacencyView):
```

```
(4)                 """An MultiAdjacencyView is a Read-only Map of Maps of Maps of Maps.
(4)                 It is a View into a dict-of-dict-of-dict-of-dict data structure.
(4)                 The inner level of dict is read-write. But the
(4)                 outer levels are read-only.
(4)                 See Also
(4)                 ========
(4)                 AtlasView: View into dict-of-dict
(4)                 AdjacencyView: View into dict-of-dict-of-dict
(4)                 """
(4)                 __slots__ = ()  # Still uses AtlasView slots names _atlas
(4)                 def __getitem__(self, name):
(8)                     return AdjacencyView(self._atlas[name])
(4)                 def copy(self):
(8)                     return {n: self[n].copy() for n in self._atlas}
(0)             class UnionAtlas(Mapping):
(4)                 """A read-only union of two atlases (dict-of-dict).
(4)                 The two dict-of-dicts represent the inner dict of
(4)                 an Adjacency:  `G.succ[node]` and `G.pred[node]`.
(4)                 The inner level of dict of both hold attribute key:value
(4)                 pairs and is read-write. But the outer level is read-only.
(4)                 See Also
(4)                 ========
(4)                 UnionAdjacency: View into dict-of-dict-of-dict
(4)                 UnionMultiAdjacency: View into dict-of-dict-of-dict-of-dict
(4)                 """
(4)                 __slots__ = ("_succ", "_pred")
(4)                 def __getstate__(self):
(8)                     return {"_succ": self._succ, "_pred": self._pred}
(4)                 def __setstate__(self, state):
(8)                     self._succ = state["_succ"]
(8)                     self._pred = state["_pred"]
(4)                 def __init__(self, succ, pred):
(8)                     self._succ = succ
(8)                     self._pred = pred
(4)                 def __len__(self):
(8)                     return len(self._succ.keys() | self._pred.keys())
(4)                 def __iter__(self):
(8)                     return iter(set(self._succ.keys()) | set(self._pred.keys()))
(4)                 def __getitem__(self, key):
(8)                     try:
(12)                        return self._succ[key]
(8)                     except KeyError:
(12)                        return self._pred[key]
(4)                 def copy(self):
(8)                     result = {nbr: dd.copy() for nbr, dd in self._succ.items()}
(8)                     for nbr, dd in self._pred.items():
(12)                        if nbr in result:
(16)                            result[nbr].update(dd)
(12)                        else:
(16)                            result[nbr] = dd.copy()
(8)                     return result
(4)                 def __str__(self):
(8)                     return str({nbr: self[nbr] for nbr in self})
(4)                 def __repr__(self):
(8)                     return f"{self.__class__.__name__}({self._succ!r}, {self._pred!r})"
(0)             class UnionAdjacency(Mapping):
(4)                 """A read-only union of dict Adjacencies as a Map of Maps of Maps.
(4)                 The two input dict-of-dict-of-dicts represent the union of
(4)                 `G.succ` and `G.pred`. Return values are UnionAtlas
(4)                 The inner level of dict is read-write. But the
(4)                 middle and outer levels are read-only.
(4)                 succ : a dict-of-dict-of-dict {node: nbrdict}
(4)                 pred : a dict-of-dict-of-dict {node: nbrdict}
(4)                 The keys for the two dicts should be the same
(4)                 See Also
(4)                 ========
(4)                 UnionAtlas: View into dict-of-dict
(4)                 UnionMultiAdjacency: View into dict-of-dict-of-dict-of-dict
(4)                 """
```

```
(4)         __slots__ = ("_succ", "_pred")
(4)         def __getstate__(self):
(8)             return {"_succ": self._succ, "_pred": self._pred}
(4)         def __setstate__(self, state):
(8)             self._succ = state["_succ"]
(8)             self._pred = state["_pred"]
(4)         def __init__(self, succ, pred):
(8)             assert len(set(succ.keys()) ^ set(pred.keys())) == 0
(8)             self._succ = succ
(8)             self._pred = pred
(4)         def __len__(self):
(8)             return len(self._succ)  # length of each dict should be the same
(4)         def __iter__(self):
(8)             return iter(self._succ)
(4)         def __getitem__(self, nbr):
(8)             return UnionAtlas(self._succ[nbr], self._pred[nbr])
(4)         def copy(self):
(8)             return {n: self[n].copy() for n in self._succ}
(4)         def __str__(self):
(8)             return str({nbr: self[nbr] for nbr in self})
(4)         def __repr__(self):
(8)             return f"{self.__class__.__name__}({self._succ!r}, {self._pred!r})"
(0)     class UnionMultiInner(UnionAtlas):
(4)         """A read-only union of two inner dicts of MultiAdjacencies.
(4)         The two input dict-of-dict-of-dicts represent the union of
(4)         `G.succ[node]` and `G.pred[node]` for MultiDiGraphs.
(4)         Return values are UnionAtlas.
(4)         The inner level of dict is read-write. But the outer levels are read-only.
(4)         See Also
(4)         ========
(4)         UnionAtlas: View into dict-of-dict
(4)         UnionAdjacency:  View into dict-of-dict-of-dict
(4)         UnionMultiAdjacency:  View into dict-of-dict-of-dict-of-dict
(4)         """
(4)         __slots__ = ()  # Still uses UnionAtlas slots names _succ, _pred
(4)         def __getitem__(self, node):
(8)             in_succ = node in self._succ
(8)             in_pred = node in self._pred
(8)             if in_succ:
(12)                if in_pred:
(16)                    return UnionAtlas(self._succ[node], self._pred[node])
(12)                return UnionAtlas(self._succ[node], {})
(8)             return UnionAtlas({}, self._pred[node])
(4)         def copy(self):
(8)             nodes = set(self._succ.keys()) | set(self._pred.keys())
(8)             return {n: self[n].copy() for n in nodes}
(0)     class UnionMultiAdjacency(UnionAdjacency):
(4)         """A read-only union of two dict MultiAdjacencies.
(4)         The two input dict-of-dict-of-dict-of-dicts represent the union of
(4)         `G.succ` and `G.pred` for MultiDiGraphs. Return values are UnionAdjacency.
(4)         The inner level of dict is read-write. But the outer levels are read-only.
(4)         See Also
(4)         ========
(4)         UnionAtlas:  View into dict-of-dict
(4)         UnionMultiInner:  View into dict-of-dict-of-dict
(4)         """
(4)         __slots__ = ()  # Still uses UnionAdjacency slots names _succ, _pred
(4)         def __getitem__(self, node):
(8)             return UnionMultiInner(self._succ[node], self._pred[node])
(0)     class FilterAtlas(Mapping):  # nodedict, nbrdict, keydict
(4)         """A read-only Mapping of Mappings with filtering criteria for nodes.
(4)         It is a view into a dict-of-dict data structure, and it selects only
(4)         nodes that meet the criteria defined by ``NODE_OK``.
(4)         See Also
(4)         ========
(4)         FilterAdjacency
(4)         FilterMultiInner
(4)         FilterMultiAdjacency
(4)         """
```

```
(4)                 def __init__(self, d, NODE_OK):
(8)                     self._atlas = d
(8)                     self.NODE_OK = NODE_OK
(4)                 def __len__(self):
(8)                     return sum(1 for n in self)
(4)                 def __iter__(self):
(8)                     try:  # check that NODE_OK has attr 'nodes'
(12)                        node_ok_shorter = 2 * len(self.NODE_OK.nodes) < len(self._atlas)
(8)                     except AttributeError:
(12)                        node_ok_shorter = False
(8)                     if node_ok_shorter:
(12)                        return (n for n in self.NODE_OK.nodes if n in self._atlas)
(8)                     return (n for n in self._atlas if self.NODE_OK(n))
(4)                 def __getitem__(self, key):
(8)                     if key in self._atlas and self.NODE_OK(key):
(12)                        return self._atlas[key]
(8)                     raise KeyError(f"Key {key} not found")
(4)                 def __str__(self):
(8)                     return str({nbr: self[nbr] for nbr in self})
(4)                 def __repr__(self):
(8)                     return f"{self.__class__.__name__}({self._atlas!r}, {self.NODE_OK!r})"
(0)             class FilterAdjacency(Mapping):  # edgedict
(4)                 """A read-only Mapping of Mappings with filtering criteria for nodes and
edges.
(4)                 It is a view into a dict-of-dict-of-dict data structure, and it selects
nodes
(4)                 and edges that satisfy specific criteria defined by ``NODE_OK`` and
``EDGE_OK``,
(4)                 respectively.
(4)                 See Also
(4)                 ========
(4)                 FilterAtlas
(4)                 FilterMultiInner
(4)                 FilterMultiAdjacency
(4)                 """
(4)                 def __init__(self, d, NODE_OK, EDGE_OK):
(8)                     self._atlas = d
(8)                     self.NODE_OK = NODE_OK
(8)                     self.EDGE_OK = EDGE_OK
(4)                 def __len__(self):
(8)                     return sum(1 for n in self)
(4)                 def __iter__(self):
(8)                     try:  # check that NODE_OK has attr 'nodes'
(12)                        node_ok_shorter = 2 * len(self.NODE_OK.nodes) < len(self._atlas)
(8)                     except AttributeError:
(12)                        node_ok_shorter = False
(8)                     if node_ok_shorter:
(12)                        return (n for n in self.NODE_OK.nodes if n in self._atlas)
(8)                     return (n for n in self._atlas if self.NODE_OK(n))
(4)                 def __getitem__(self, node):
(8)                     if node in self._atlas and self.NODE_OK(node):
(12)                        def new_node_ok(nbr):
(16)                            return self.NODE_OK(nbr) and self.EDGE_OK(node, nbr)
(12)                        return FilterAtlas(self._atlas[node], new_node_ok)
(8)                     raise KeyError(f"Key {node} not found")
(4)                 def __str__(self):
(8)                     return str({nbr: self[nbr] for nbr in self})
(4)                 def __repr__(self):
(8)                     name = self.__class__.__name__
(8)                     return f"{name}({self._atlas!r}, {self.NODE_OK!r}, {self.EDGE_OK!r})"
(0)             class FilterMultiInner(FilterAdjacency):  # muliedge_seconddict
(4)                 """A read-only Mapping of Mappings with filtering criteria for nodes and
edges.
(4)                 It is a view into a dict-of-dict-of-dict-of-dict data structure, and it
selects nodes
(4)                 and edges that meet specific criteria defined by ``NODE_OK`` and
``EDGE_OK``.
(4)                 See Also
(4)                 ========
```

```
265: (4)                FilterAtlas
266: (4)                FilterAdjacency
267: (4)                FilterMultiAdjacency
268: (4)                """
269: (4)                def __iter__(self):
270: (8)                    try:  # check that NODE_OK has attr 'nodes'
271: (12)                       node_ok_shorter = 2 * len(self.NODE_OK.nodes) < len(self._atlas)
272: (8)                    except AttributeError:
273: (12)                       node_ok_shorter = False
274: (8)                    if node_ok_shorter:
275: (12)                       my_nodes = (n for n in self.NODE_OK.nodes if n in self._atlas)
276: (8)                    else:
277: (12)                       my_nodes = (n for n in self._atlas if self.NODE_OK(n))
278: (8)                    for n in my_nodes:
279: (12)                       some_keys_ok = False
280: (12)                       for key in self._atlas[n]:
281: (16)                           if self.EDGE_OK(n, key):
282: (20)                               some_keys_ok = True
283: (20)                               break
284: (12)                       if some_keys_ok is True:
285: (16)                           yield n
286: (4)                def __getitem__(self, nbr):
287: (8)                    if nbr in self._atlas and self.NODE_OK(nbr):
288: (12)                       def new_node_ok(key):
289: (16)                           return self.EDGE_OK(nbr, key)
290: (12)                       return FilterAtlas(self._atlas[nbr], new_node_ok)
291: (8)                    raise KeyError(f"Key {nbr} not found")
292: (0)            class FilterMultiAdjacency(FilterAdjacency):  # multiedgedict
293: (4)                """A read-only Mapping of Mappings with filtering criteria
294: (4)                for nodes and edges.
295: (4)                It is a view into a dict-of-dict-of-dict-of-dict data structure,
296: (4)                and it selects nodes and edges that satisfy specific criteria
297: (4)                defined by ``NODE_OK`` and ``EDGE_OK``, respectively.
298: (4)                See Also
299: (4)                ========
300: (4)                FilterAtlas
301: (4)                FilterAdjacency
302: (4)                FilterMultiInner
303: (4)                """
304: (4)                def __getitem__(self, node):
305: (8)                    if node in self._atlas and self.NODE_OK(node):
306: (12)                       def edge_ok(nbr, key):
307: (16)                           return self.NODE_OK(nbr) and self.EDGE_OK(node, nbr, key)
308: (12)                       return FilterMultiInner(self._atlas[node], self.NODE_OK, edge_ok)
309: (8)                    raise KeyError(f"Key {node} not found")

----------------------------------------

File 390 - digraph.py:

1: (0)              """Base class for directed graphs."""
2: (0)              from copy import deepcopy
3: (0)              from functools import cached_property
4: (0)              import networkx as nx
5: (0)              from networkx import convert
6: (0)              from networkx.classes.coreviews import AdjacencyView
7: (0)              from networkx.classes.graph import Graph
8: (0)              from networkx.classes.reportviews import (
9: (4)                  DiDegreeView,
10: (4)                 InDegreeView,
11: (4)                 InEdgeView,
12: (4)                 OutDegreeView,
13: (4)                 OutEdgeView,
14: (0)             )
15: (0)             from networkx.exception import NetworkXError
16: (0)             __all__ = ["DiGraph"]
17: (0)             class _CachedPropertyResetterAdjAndSucc:
18: (4)                 """Data Descriptor class that syncs and resets cached properties adj and
succ
```

```
19: (4)                 The cached properties `adj` and `succ` are reset whenever `_adj` or
`_succ`
20: (4)                 are set to new objects. In addition, the attributes `_succ` and `_adj`
21: (4)                 are synced so these two names point to the same object.
22: (4)                 This object sits on a class and ensures that any instance of that
23: (4)                 class clears its cached properties "succ" and "adj" whenever the
24: (4)                 underlying instance attributes "_succ" or "_adj" are set to a new object.
25: (4)                 It only affects the set process of the obj._adj and obj._succ attribute.
26: (4)                 All get/del operations act as they normally would.
27: (4)                 For info on Data Descriptors see:
https://docs.python.org/3/howto/descriptor.html
28: (4)                 """
29: (4)                 def __set__(self, obj, value):
30: (8)                     od = obj.__dict__
31: (8)                     od["_adj"] = value
32: (8)                     od["_succ"] = value
33: (8)                     if "adj" in od:
34: (12)                        del od["adj"]
35: (8)                     if "succ" in od:
36: (12)                        del od["succ"]
37: (0)             class _CachedPropertyResetterPred:
38: (4)                 """Data Descriptor class for _pred that resets ``pred`` cached_property
when needed
39: (4)                 This assumes that the ``cached_property`` ``G.pred`` should be reset
whenever
40: (4)                 ``G._pred`` is set to a new value.
41: (4)                 This object sits on a class and ensures that any instance of that
42: (4)                 class clears its cached property "pred" whenever the underlying
43: (4)                 instance attribute "_pred" is set to a new object. It only affects
44: (4)                 the set process of the obj._pred attribute. All get/del operations
45: (4)                 act as they normally would.
46: (4)                 For info on Data Descriptors see:
https://docs.python.org/3/howto/descriptor.html
47: (4)                 """
48: (4)                 def __set__(self, obj, value):
49: (8)                     od = obj.__dict__
50: (8)                     od["_pred"] = value
51: (8)                     if "pred" in od:
52: (12)                        del od["pred"]
53: (0)             class DiGraph(Graph):
54: (4)                 """
55: (4)                 Base class for directed graphs.
56: (4)                 A DiGraph stores nodes and edges with optional data, or attributes.
57: (4)                 DiGraphs hold directed edges.  Self loops are allowed but multiple
58: (4)                 (parallel) edges are not.
59: (4)                 Nodes can be arbitrary (hashable) Python objects with optional
60: (4)                 key/value attributes. By convention `None` is not used as a node.
61: (4)                 Edges are represented as links between nodes with optional
62: (4)                 key/value attributes.
63: (4)                 Parameters
64: (4)                 ----------
65: (4)                 incoming_graph_data : input graph (optional, default: None)
66: (8)                     Data to initialize graph. If None (default) an empty
67: (8)                     graph is created.  The data can be any format that is supported
68: (8)                     by the to_networkx_graph() function, currently including edge list,
69: (8)                     dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy
70: (8)                     sparse matrix, or PyGraphviz graph.
71: (4)                 attr : keyword arguments, optional (default= no attributes)
72: (8)                     Attributes to add to graph as key=value pairs.
73: (4)                 See Also
74: (4)                 --------
75: (4)                 Graph
76: (4)                 MultiGraph
77: (4)                 MultiDiGraph
78: (4)                 Examples
79: (4)                 --------
80: (4)                 Create an empty graph structure (a "null graph") with no nodes and
81: (4)                 no edges.
82: (4)                 >>> G = nx.DiGraph()
```

```
(4)                 G can be grown in several ways.
(4)                 **Nodes:**
(4)                 Add one node at a time:
(4)                 >>> G.add_node(1)
(4)                 Add the nodes from any container (a list, dict, set or
(4)                 even the lines from a file or the nodes from another graph).
(4)                 >>> G.add_nodes_from([2, 3])
(4)                 >>> G.add_nodes_from(range(100, 110))
(4)                 >>> H = nx.path_graph(10)
(4)                 >>> G.add_nodes_from(H)
(4)                 In addition to strings and integers any hashable Python object
(4)                 (except None) can represent a node, e.g. a customized node object,
(4)                 or even another Graph.
(4)                 >>> G.add_node(H)
(4)                 **Edges:**
(4)                 G can also be grown by adding edges.
(4)                 Add one edge,
(4)                 >>> G.add_edge(1, 2)
(4)                 a list of edges,
(4)                 >>> G.add_edges_from([(1, 2), (1, 3)])
(4)                 or a collection of edges,
(4)                 >>> G.add_edges_from(H.edges)
(4)                 If some edges connect nodes not yet in the graph, the nodes
(4)                 are added automatically.  There are no errors when adding
(4)                 nodes or edges that already exist.
(4)                 **Attributes:**
(4)                 Each graph, node, and edge can hold key/value attribute pairs
(4)                 in an associated attribute dictionary (the keys must be hashable).
(4)                 By default these are empty, but can be added or changed using
(4)                 add_edge, add_node or direct manipulation of the attribute
(4)                 dictionaries named graph, node and edge respectively.
(4)                 >>> G = nx.DiGraph(day="Friday")
(4)                 >>> G.graph
(4)                 {'day': 'Friday'}
(4)                 Add node attributes using add_node(), add_nodes_from() or G.nodes
(4)                 >>> G.add_node(1, time="5pm")
(4)                 >>> G.add_nodes_from([3], time="2pm")
(4)                 >>> G.nodes[1]
(4)                 {'time': '5pm'}
(4)                 >>> G.nodes[1]["room"] = 714
(4)                 >>> del G.nodes[1]["room"]  # remove attribute
(4)                 >>> list(G.nodes(data=True))
(4)                 [(1, {'time': '5pm'}), (3, {'time': '2pm'})]
(4)                 Add edge attributes using add_edge(), add_edges_from(), subscript
(4)                 notation, or G.edges.
(4)                 >>> G.add_edge(1, 2, weight=4.7)
(4)                 >>> G.add_edges_from([(3, 4), (4, 5)], color="red")
(4)                 >>> G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
(4)                 >>> G[1][2]["weight"] = 4.7
(4)                 >>> G.edges[1, 2]["weight"] = 4
(4)                 Warning: we protect the graph data structure by making `G.edges[1, 2]` a
(4)                 read-only dict-like structure. However, you can assign to attributes
(4)                 in e.g. `G.edges[1, 2]`. Thus, use 2 sets of brackets to add/change
(4)                 data attributes: `G.edges[1, 2]['weight'] = 4`
(4)                 (For multigraphs: `MG.edges[u, v, key][name] = value`).
(4)                 **Shortcuts:**
(4)                 Many common graph features allow python syntax to speed reporting.
(4)                 >>> 1 in G  # check if node in graph
(4)                 True
(4)                 >>> [n for n in G if n < 3]  # iterate through nodes
(4)                 [1, 2]
(4)                 >>> len(G)  # number of nodes in graph
(4)                 5
(4)                 Often the best way to traverse all edges of a graph is via the neighbors.
(4)                 The neighbors are reported as an adjacency-dict `G.adj` or `G.adjacency()`
(4)                 >>> for n, nbrsdict in G.adjacency():
(4)                 ...     for nbr, eattr in nbrsdict.items():
(4)                 ...         if "weight" in eattr:
(4)                 ...             # Do something useful with the edges
```

```
(4)                 ...             pass
(4)                 But the edges reporting object is often more convenient:
(4)                 >>> for u, v, weight in G.edges(data="weight"):
(4)                 ...     if weight is not None:
(4)                 ...         # Do something useful with the edges
(4)                 ...         pass
(4)                 **Reporting:**
(4)                 Simple graph information is obtained using object-attributes and methods.
(4)                 Reporting usually provides views instead of containers to reduce memory
(4)                 usage. The views update as the graph is updated similarly to dict-views.
(4)                 The objects `nodes`, `edges` and `adj` provide access to data attributes
(4)                 via lookup (e.g. `nodes[n]`, `edges[u, v]`, `adj[u][v]`) and iteration
(4)                 (e.g. `nodes.items()`, `nodes.data('color')`,
(4)                 `nodes.data('color', default='blue')` and similarly for `edges`)
(4)                 Views exist for `nodes`, `edges`, `neighbors()`/`adj` and `degree`.
(4)                 For details on these and other miscellaneous methods, see below.
(4)                 **Subclasses (Advanced):**
(4)                 The Graph class uses a dict-of-dict-of-dict data structure.
(4)                 The outer dict (node_dict) holds adjacency information keyed by node.
(4)                 The next dict (adjlist_dict) represents the adjacency information and
holds
(4)                 edge data keyed by neighbor.  The inner dict (edge_attr_dict) represents
(4)                 the edge data and holds edge attribute values keyed by attribute names.
(4)                 Each of these three dicts can be replaced in a subclass by a user defined
(4)                 dict-like object. In general, the dict-like features should be
(4)                 maintained but extra features can be added. To replace one of the
(4)                 dicts create a new graph class by changing the class(!) variable
(4)                 holding the factory for that dict-like structure. The variable names are
(4)                 node_dict_factory, node_attr_dict_factory, adjlist_inner_dict_factory,
(4)                 adjlist_outer_dict_factory, edge_attr_dict_factory and
graph_attr_dict_factory.
(4)                 node_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the dict containing node
(8)                     attributes, keyed by node id.
(8)                     It should require no arguments and return a dict-like object
(4)                 node_attr_dict_factory: function, (default: dict)
(8)                     Factory function to be used to create the node attribute
(8)                     dict which holds attribute values keyed by attribute name.
(8)                     It should require no arguments and return a dict-like object
(4)                 adjlist_outer_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the outer-most dict
(8)                     in the data structure that holds adjacency info keyed by node.
(8)                     It should require no arguments and return a dict-like object.
(4)                 adjlist_inner_dict_factory : function, optional (default: dict)
(8)                     Factory function to be used to create the adjacency list
(8)                     dict which holds edge data keyed by neighbor.
(8)                     It should require no arguments and return a dict-like object
(4)                 edge_attr_dict_factory : function, optional (default: dict)
(8)                     Factory function to be used to create the edge attribute
(8)                     dict which holds attribute values keyed by attribute name.
(8)                     It should require no arguments and return a dict-like object.
(4)                 graph_attr_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the graph attribute
(8)                     dict which holds attribute values keyed by attribute name.
(8)                     It should require no arguments and return a dict-like object.
(4)                 Typically, if your extension doesn't impact the data structure all
(4)                 methods will inherited without issue except: `to_directed/to_undirected`.
(4)                 By default these methods create a DiGraph/Graph class and you probably
(4)                 want them to create your extension of a DiGraph/Graph. To facilitate
(4)                 this we define two class variables that you can set in your subclass.
(4)                 to_directed_class : callable, (default: DiGraph or MultiDiGraph)
(8)                     Class to create a new graph structure in the `to_directed` method.
(8)                     If `None`, a NetworkX class (DiGraph or MultiDiGraph) is used.
(4)                 to_undirected_class : callable, (default: Graph or MultiGraph)
(8)                     Class to create a new graph structure in the `to_undirected` method.
(8)                     If `None`, a NetworkX class (Graph or MultiGraph) is used.
(4)                 **Subclassing Example**
(4)                 Create a low memory graph class that effectively disallows edge
(4)                 attributes by using a single attribute dict for all edges.
```

```
    This reduces the memory used, but you lose edge attributes.
    >>> class ThinGraph(nx.Graph):
    ...     all_edge_dict = {"weight": 1}
    ...
    ...     def single_edge_dict(self):
    ...         return self.all_edge_dict
    ...
    ...     edge_attr_dict_factory = single_edge_dict
    >>> G = ThinGraph()
    >>> G.add_edge(2, 1)
    >>> G[2][1]
    {'weight': 1}
    >>> G.add_edge(2, 2)
    >>> G[2][1] is G[2][2]
    True
    """
    _adj = _CachedPropertyResetterAdjAndSucc()  # type: ignore[assignment]
    _succ = _adj  # type: ignore[has-type]
    _pred = _CachedPropertyResetterPred()
    def __init__(self, incoming_graph_data=None, **attr):
        """Initialize a graph with edges, name, or graph attributes.
        Parameters
        ----------
        incoming_graph_data : input graph (optional, default: None)
            Data to initialize graph.  If None (default) an empty
            graph is created.  The data can be an edge list, or any
            NetworkX graph object.  If the corresponding optional Python
            packages are installed the data can also be a 2D NumPy array, a
            SciPy sparse array, or a PyGraphviz graph.
        attr : keyword arguments, optional (default= no attributes)
            Attributes to add to graph as key=value pairs.
        See Also
        --------
        convert
        Examples
        --------
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G = nx.Graph(name="my graph")
        >>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
        >>> G = nx.Graph(e)
        Arbitrary graph attribute pairs (key=value) may be assigned
        >>> G = nx.Graph(e, day="Friday")
        >>> G.graph
        {'day': 'Friday'}
        """
        self.graph = self.graph_attr_dict_factory()  # dictionary for graph attributes
        self._node = self.node_dict_factory()  # dictionary for node attr
        self._adj = self.adjlist_outer_dict_factory()  # empty adjacency dict successor
        self._pred = self.adjlist_outer_dict_factory()  # predecessor
        self.__networkx_cache__ = {}
        if incoming_graph_data is not None:
            convert.to_networkx_graph(incoming_graph_data, create_using=self)
        self.graph.update(attr)
    @cached_property
    def adj(self):
        """Graph adjacency object holding the neighbors of each node.
        This object is a read-only dict-like structure with node keys
        and neighbor-dict values.  The neighbor-dict is keyed by neighbor
        to the edge-data-dict.  So `G.adj[3][2]['color'] = 'blue'` sets
        the color of the edge `(3, 2)` to `"blue"`.
        Iterating over G.adj behaves like a dict. Useful idioms include
        `for nbr, datadict in G.adj[n].items():`.
        The neighbor information is also provided by subscripting the graph.
        So `for nbr, foovalue in G[node].data('foo', default=1):` works.
        For directed graphs, `G.adj` holds outgoing (successor) info.
        """
        return AdjacencyView(self._succ)
```

```
    @cached_property
    def succ(self):
        """Graph adjacency object holding the successors of each node.
        This object is a read-only dict-like structure with node keys
        and neighbor-dict values.  The neighbor-dict is keyed by neighbor
        to the edge-data-dict.  So `G.succ[3][2]['color'] = 'blue'` sets
        the color of the edge `(3, 2)` to `"blue"`.
        Iterating over G.succ behaves like a dict. Useful idioms include
        `for nbr, datadict in G.succ[n].items():`.  A data-view not provided
        by dicts also exists: `for nbr, foovalue in G.succ[node].data('foo'):`
        and a default can be set via a `default` argument to the `data`
method.
        The neighbor information is also provided by subscripting the graph.
        So `for nbr, foovalue in G[node].data('foo', default=1):` works.
        For directed graphs, `G.adj` is identical to `G.succ`.
        """
        return AdjacencyView(self._succ)
    @cached_property
    def pred(self):
        """Graph adjacency object holding the predecessors of each node.
        This object is a read-only dict-like structure with node keys
        and neighbor-dict values.  The neighbor-dict is keyed by neighbor
        to the edge-data-dict.  So `G.pred[2][3]['color'] = 'blue'` sets
        the color of the edge `(3, 2)` to `"blue"`.
        Iterating over G.pred behaves like a dict. Useful idioms include
        `for nbr, datadict in G.pred[n].items():`.  A data-view not provided
        by dicts also exists: `for nbr, foovalue in G.pred[node].data('foo'):`
        A default can be set via a `default` argument to the `data` method.
        """
        return AdjacencyView(self._pred)
    def add_node(self, node_for_adding, **attr):
        """Add a single node `node_for_adding` and update node attributes.
        Parameters
        ----------
        node_for_adding : node
            A node can be any hashable Python object except None.
        attr : keyword arguments, optional
            Set or change node attributes using key=value.
        See Also
        --------
        add_nodes_from
        Examples
        --------
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.add_node(1)
        >>> G.add_node("Hello")
        >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
        >>> G.add_node(K3)
        >>> G.number_of_nodes()
        3
        Use keywords set/change node attributes:
        >>> G.add_node(1, size=10)
        >>> G.add_node(3, weight=0.4, UTM=("13S", 382871, 3972649))
        Notes
        -----
        A hashable object is one that can be used as a key in a Python
        dictionary. This includes strings, numbers, tuples of strings
        and numbers, etc.
        On many platforms hashable items also include mutables such as
        NetworkX Graphs, though one should be careful that the hash
        doesn't change on mutables.
        """
        if node_for_adding not in self._succ:
            if node_for_adding is None:
                raise ValueError("None cannot be a node")
            self._succ[node_for_adding] = self.adjlist_inner_dict_factory()
            self._pred[node_for_adding] = self.adjlist_inner_dict_factory()
            attr_dict = self._node[node_for_adding] =
self.node_attr_dict_factory()
```

```
(12)                        attr_dict.update(attr)
(8)                     else:  # update attr even if node already exists
(12)                        self._node[node_for_adding].update(attr)
(8)                     nx._clear_cache(self)
(4)                 def add_nodes_from(self, nodes_for_adding, **attr):
(8)                     """Add multiple nodes.
(8)                     Parameters
(8)                     ----------
(8)                     nodes_for_adding : iterable container
(12)                        A container of nodes (list, dict, set, etc.).
(12)                        OR
(12)                        A container of (node, attribute dict) tuples.
(12)                        Node attributes are updated using the attribute dict.
(8)                     attr : keyword arguments, optional (default= no attributes)
(12)                        Update attributes for all nodes in nodes.
(12)                        Node attributes specified in nodes as a tuple take
(12)                        precedence over attributes specified via keyword arguments.
(8)                     See Also
(8)                     --------
(8)                     add_node
(8)                     Notes
(8)                     -----
(8)                     When adding nodes from an iterator over the graph you are changing,
(8)                     a `RuntimeError` can be raised with message:
(8)                     `RuntimeError: dictionary changed size during iteration`. This
(8)                     happens when the graph's underlying dictionary is modified during
(8)                     iteration. To avoid this error, evaluate the iterator into a separate
(8)                     object, e.g. by using `list(iterator_of_nodes)`, and pass this
(8)                     object to `G.add_nodes_from`.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.add_nodes_from("Hello")
(8)                     >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
(8)                     >>> G.add_nodes_from(K3)
(8)                     >>> sorted(G.nodes(), key=str)
(8)                     [0, 1, 2, 'H', 'e', 'l', 'o']
(8)                     Use keywords to update specific node attributes for every node.
(8)                     >>> G.add_nodes_from([1, 2], size=10)
(8)                     >>> G.add_nodes_from([3, 4], weight=0.4)
(8)                     Use (node, attrdict) tuples to update attributes for specific nodes.
(8)                     >>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
(8)                     >>> G.nodes[1]["size"]
(8)                     11
(8)                     >>> H = nx.Graph()
(8)                     >>> H.add_nodes_from(G.nodes(data=True))
(8)                     >>> H.nodes[1]["size"]
(8)                     11
(8)                     Evaluate an iterator over a graph if using it to modify the same graph
(8)                     >>> G = nx.DiGraph([(0, 1), (1, 2), (3, 4)])
(8)                     >>> # wrong way - will raise RuntimeError
(8)                     >>> # G.add_nodes_from(n + 1 for n in G.nodes)
(8)                     >>> # correct way
(8)                     >>> G.add_nodes_from(list(n + 1 for n in G.nodes))
(8)                     """
(8)                     for n in nodes_for_adding:
(12)                        try:
(16)                            newnode = n not in self._node
(16)                            newdict = attr
(12)                        except TypeError:
(16)                            n, ndict = n
(16)                            newnode = n not in self._node
(16)                            newdict = attr.copy()
(16)                            newdict.update(ndict)
(12)                        if newnode:
(16)                            if n is None:
(20)                                raise ValueError("None cannot be a node")
(16)                            self._succ[n] = self.adjlist_inner_dict_factory()
(16)                            self._pred[n] = self.adjlist_inner_dict_factory()
```

```
(16)                            self._node[n] = self.node_attr_dict_factory()
(12)                        self._node[n].update(newdict)
(8)                     nx._clear_cache(self)
(4)                 def remove_node(self, n):
(8)                     """Remove node n.
(8)                     Removes the node n and all adjacent edges.
(8)                     Attempting to remove a nonexistent node will raise an exception.
(8)                     Parameters
(8)                     ----------
(8)                     n : node
(11)                       A node in the graph
(8)                     Raises
(8)                     ------
(8)                     NetworkXError
(11)                       If n is not in the graph.
(8)                     See Also
(8)                     --------
(8)                     remove_nodes_from
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> list(G.edges)
(8)                     [(0, 1), (1, 2)]
(8)                     >>> G.remove_node(1)
(8)                     >>> list(G.edges)
(8)                     []
(8)                     """
(8)                     try:
(12)                        nbrs = self._succ[n]
(12)                        del self._node[n]
(8)                     except KeyError as err:  # NetworkXError if n not in self
(12)                        raise NetworkXError(f"The node {n} is not in the digraph.") from
err
(8)                     for u in nbrs:
(12)                        del self._pred[u][n]  # remove all edges n-u in digraph
(8)                     del self._succ[n]  # remove node from succ
(8)                     for u in self._pred[n]:
(12)                        del self._succ[u][n]  # remove all edges n-u in digraph
(8)                     del self._pred[n]  # remove node from pred
(8)                     nx._clear_cache(self)
(4)                 def remove_nodes_from(self, nodes):
(8)                     """Remove multiple nodes.
(8)                     Parameters
(8)                     ----------
(8)                     nodes : iterable container
(12)                        A container of nodes (list, dict, set, etc.).  If a node
(12)                        in the container is not in the graph it is silently ignored.
(8)                     See Also
(8)                     --------
(8)                     remove_node
(8)                     Notes
(8)                     -----
(8)                     When removing nodes from an iterator over the graph you are changing,
(8)                     a `RuntimeError` will be raised with message:
(8)                     `RuntimeError: dictionary changed size during iteration`. This
(8)                     happens when the graph's underlying dictionary is modified during
(8)                     iteration. To avoid this error, evaluate the iterator into a separate
(8)                     object, e.g. by using `list(iterator_of_nodes)`, and pass this
(8)                     object to `G.remove_nodes_from`.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> e = list(G.nodes)
(8)                     >>> e
(8)                     [0, 1, 2]
(8)                     >>> G.remove_nodes_from(e)
(8)                     >>> list(G.nodes)
(8)                     []
(8)                     Evaluate an iterator over a graph if using it to modify the same graph
```

```
490: (8)                    >>> G = nx.DiGraph([(0, 1), (1, 2), (3, 4)])
491: (8)                    >>> # this command will fail, as the graph's dict is modified during
iteration
492: (8)                    >>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
493: (8)                    >>> # this command will work, since the dictionary underlying graph is
not modified
494: (8)                    >>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
495: (8)                    """
496: (8)                    for n in nodes:
497: (12)                       try:
498: (16)                           succs = self._succ[n]
499: (16)                           del self._node[n]
500: (16)                           for u in succs:
501: (20)                               del self._pred[u][n]  # remove all edges n-u in digraph
502: (16)                           del self._succ[n]  # now remove node
503: (16)                           for u in self._pred[n]:
504: (20)                               del self._succ[u][n]  # remove all edges n-u in digraph
505: (16)                           del self._pred[n]  # now remove node
506: (12)                       except KeyError:
507: (16)                           pass  # silent failure on remove
508: (8)                    nx._clear_cache(self)
509: (4)                def add_edge(self, u_of_edge, v_of_edge, **attr):
510: (8)                    """Add an edge between u and v.
511: (8)                    The nodes u and v will be automatically added if they are
512: (8)                    not already in the graph.
513: (8)                    Edge attributes can be specified with keywords or by directly
514: (8)                    accessing the edge's attribute dictionary. See examples below.
515: (8)                    Parameters
516: (8)                    ----------
517: (8)                    u_of_edge, v_of_edge : nodes
518: (12)                       Nodes can be, for example, strings or numbers.
519: (12)                       Nodes must be hashable (and not None) Python objects.
520: (8)                    attr : keyword arguments, optional
521: (12)                       Edge data (or labels or objects) can be assigned using
522: (12)                       keyword arguments.
523: (8)                    See Also
524: (8)                    --------
525: (8)                    add_edges_from : add a collection of edges
526: (8)                    Notes
527: (8)                    -----
528: (8)                    Adding an edge that already exists updates the edge data.
529: (8)                    Many NetworkX algorithms designed for weighted graphs use
530: (8)                    an edge attribute (by default `weight`) to hold a numerical value.
531: (8)                    Examples
532: (8)                    --------
533: (8)                    The following all add the edge e=(1, 2) to graph G:
534: (8)                    >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
535: (8)                    >>> e = (1, 2)
536: (8)                    >>> G.add_edge(1, 2)  # explicit two-node form
537: (8)                    >>> G.add_edge(*e)  # single edge as tuple of two nodes
538: (8)                    >>> G.add_edges_from([(1, 2)])  # add edges from iterable container
539: (8)                    Associate data to edges using keywords:
540: (8)                    >>> G.add_edge(1, 2, weight=3)
541: (8)                    >>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
542: (8)                    For non-string attribute keys, use subscript notation.
543: (8)                    >>> G.add_edge(1, 2)
544: (8)                    >>> G[1][2].update({0: 5})
545: (8)                    >>> G.edges[1, 2].update({0: 5})
546: (8)                    """
547: (8)                    u, v = u_of_edge, v_of_edge
548: (8)                    if u not in self._succ:
549: (12)                       if u is None:
550: (16)                           raise ValueError("None cannot be a node")
551: (12)                       self._succ[u] = self.adjlist_inner_dict_factory()
552: (12)                       self._pred[u] = self.adjlist_inner_dict_factory()
553: (12)                       self._node[u] = self.node_attr_dict_factory()
554: (8)                    if v not in self._succ:
555: (12)                       if v is None:
556: (16)                           raise ValueError("None cannot be a node")
```

```
            self._succ[v] = self.adjlist_inner_dict_factory()
            self._pred[v] = self.adjlist_inner_dict_factory()
            self._node[v] = self.node_attr_dict_factory()
        datadict = self._adj[u].get(v, self.edge_attr_dict_factory())
        datadict.update(attr)
        self._succ[u][v] = datadict
        self._pred[v][u] = datadict
        nx._clear_cache(self)
    def add_edges_from(self, ebunch_to_add, **attr):
        """Add all the edges in ebunch_to_add.
        Parameters
        ----------
        ebunch_to_add : container of edges
            Each edge given in the container will be added to the
            graph. The edges must be given as 2-tuples (u, v) or
            3-tuples (u, v, d) where d is a dictionary containing edge data.
        attr : keyword arguments, optional
            Edge data (or labels or objects) can be assigned using
            keyword arguments.
        See Also
        --------
        add_edge : add a single edge
        add_weighted_edges_from : convenient way to add weighted edges
        Notes
        -----
        Adding the same edge twice has no effect but any edge data
        will be updated when each duplicate edge is added.
        Edge attributes specified in an ebunch take precedence over
        attributes specified via keyword arguments.
        When adding edges from an iterator over the graph you are changing,
        a `RuntimeError` can be raised with message:
        `RuntimeError: dictionary changed size during iteration`. This
        happens when the graph's underlying dictionary is modified during
        iteration. To avoid this error, evaluate the iterator into a separate
        object, e.g. by using `list(iterator_of_edges)`, and pass this
        object to `G.add_edges_from`.
        Examples
        --------
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
        >>> e = zip(range(0, 3), range(1, 4))
        >>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
        Associate data to edges
        >>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
        >>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
        Evaluate an iterator over a graph if using it to modify the same graph
        >>> G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
        >>> # Grow graph by one new node, adding edges to all existing nodes.
        >>> # wrong way - will raise RuntimeError
        >>> # G.add_edges_from(((5, n) for n in G.nodes))
        >>> # right way - note that there will be no self-edge for node 5
        >>> G.add_edges_from(list((5, n) for n in G.nodes))
        """
        for e in ebunch_to_add:
            ne = len(e)
            if ne == 3:
                u, v, dd = e
            elif ne == 2:
                u, v = e
                dd = {}
            else:
                raise NetworkXError(f"Edge tuple {e} must be a 2-tuple or 3-tuple.")
            if u not in self._succ:
                if u is None:
                    raise ValueError("None cannot be a node")
                self._succ[u] = self.adjlist_inner_dict_factory()
                self._pred[u] = self.adjlist_inner_dict_factory()
                self._node[u] = self.node_attr_dict_factory()
```

```
(12)                        if v not in self._succ:
(16)                            if v is None:
(20)                                raise ValueError("None cannot be a node")
(16)                            self._succ[v] = self.adjlist_inner_dict_factory()
(16)                            self._pred[v] = self.adjlist_inner_dict_factory()
(16)                            self._node[v] = self.node_attr_dict_factory()
(12)                        datadict = self._adj[u].get(v, self.edge_attr_dict_factory())
(12)                        datadict.update(attr)
(12)                        datadict.update(dd)
(12)                        self._succ[u][v] = datadict
(12)                        self._pred[v][u] = datadict
(8)                     nx._clear_cache(self)
(4)                 def remove_edge(self, u, v):
(8)                     """Remove the edge between u and v.
(8)                     Parameters
(8)                     ----------
(8)                     u, v : nodes
(12)                        Remove the edge between nodes u and v.
(8)                     Raises
(8)                     ------
(8)                     NetworkXError
(12)                        If there is not an edge between u and v.
(8)                     See Also
(8)                     --------
(8)                     remove_edges_from : remove a collection of edges
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.Graph()  # or DiGraph, etc
(8)                     >>> nx.add_path(G, [0, 1, 2, 3])
(8)                     >>> G.remove_edge(0, 1)
(8)                     >>> e = (1, 2)
(8)                     >>> G.remove_edge(*e)  # unpacks e from an edge tuple
(8)                     >>> e = (2, 3, {"weight": 7})  # an edge with attribute data
(8)                     >>> G.remove_edge(*e[:2])  # select first part of edge tuple
(8)                     """
(8)                     try:
(12)                        del self._succ[u][v]
(12)                        del self._pred[v][u]
(8)                     except KeyError as err:
(12)                        raise NetworkXError(f"The edge {u}-{v} not in graph.") from err
(8)                     nx._clear_cache(self)
(4)                 def remove_edges_from(self, ebunch):
(8)                     """Remove all edges specified in ebunch.
(8)                     Parameters
(8)                     ----------
(8)                     ebunch: list or container of edge tuples
(12)                        Each edge given in the list or container will be removed
(12)                        from the graph. The edges can be:
(16)                            - 2-tuples (u, v) edge between u and v.
(16)                            - 3-tuples (u, v, k) where k is ignored.
(8)                     See Also
(8)                     --------
(8)                     remove_edge : remove a single edge
(8)                     Notes
(8)                     -----
(8)                     Will fail silently if an edge in ebunch is not in the graph.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> ebunch = [(1, 2), (2, 3)]
(8)                     >>> G.remove_edges_from(ebunch)
(8)                     """
(8)                     for e in ebunch:
(12)                        u, v = e[:2]  # ignore edge data
(12)                        if u in self._succ and v in self._succ[u]:
(16)                            del self._succ[u][v]
(16)                            del self._pred[v][u]
(8)                     nx._clear_cache(self)
(4)                 def has_successor(self, u, v):
```

```
694: (8)                    """Returns True if node u has successor v.
695: (8)                    This is true if graph has the edge u->v.
696: (8)                    """
697: (8)                    return u in self._succ and v in self._succ[u]
698: (4)                def has_predecessor(self, u, v):
699: (8)                    """Returns True if node u has predecessor v.
700: (8)                    This is true if graph has the edge u<-v.
701: (8)                    """
702: (8)                    return u in self._pred and v in self._pred[u]
703: (4)                def successors(self, n):
704: (8)                    """Returns an iterator over successor nodes of n.
705: (8)                    A successor of n is a node m such that there exists a directed
706: (8)                    edge from n to m.
707: (8)                    Parameters
708: (8)                    ----------
709: (8)                    n : node
710: (11)                      A node in the graph
711: (8)                    Raises
712: (8)                    ------
713: (8)                    NetworkXError
714: (11)                      If n is not in the graph.
715: (8)                    See Also
716: (8)                    --------
717: (8)                    predecessors
718: (8)                    Notes
719: (8)                    -----
720: (8)                    neighbors() and successors() are the same.
721: (8)                    """
722: (8)                    try:
723: (12)                       return iter(self._succ[n])
724: (8)                    except KeyError as err:
725: (12)                       raise NetworkXError(f"The node {n} is not in the digraph.") from
err
726: (4)                neighbors = successors
727: (4)                def predecessors(self, n):
728: (8)                    """Returns an iterator over predecessor nodes of n.
729: (8)                    A predecessor of n is a node m such that there exists a directed
730: (8)                    edge from m to n.
731: (8)                    Parameters
732: (8)                    ----------
733: (8)                    n : node
734: (11)                      A node in the graph
735: (8)                    Raises
736: (8)                    ------
737: (8)                    NetworkXError
738: (11)                      If n is not in the graph.
739: (8)                    See Also
740: (8)                    --------
741: (8)                    successors
742: (8)                    """
743: (8)                    try:
744: (12)                       return iter(self._pred[n])
745: (8)                    except KeyError as err:
746: (12)                       raise NetworkXError(f"The node {n} is not in the digraph.") from
err
747: (4)                @cached_property
748: (4)                def edges(self):
749: (8)                    """An OutEdgeView of the DiGraph as G.edges or G.edges().
750: (8)                    edges(self, nbunch=None, data=False, default=None)
751: (8)                    The OutEdgeView provides set-like operations on the edge-tuples
752: (8)                    as well as edge attribute lookup. When called, it also provides
753: (8)                    an EdgeDataView object which allows control of access to edge
754: (8)                    attributes (but does not provide set-like operations).
755: (8)                    Hence, `G.edges[u, v]['color']` provides the value of the color
756: (8)                    attribute for edge `(u, v)` while
757: (8)                    `for (u, v, c) in G.edges.data('color', default='red'):`
758: (8)                    iterates through all the edges yielding the color attribute
759: (8)                    with default `'red'` if no color attribute exists.
760: (8)                    Parameters
```

```
(8)                     ----------
(8)                     nbunch : single node, container, or all nodes (default= all nodes)
(12)                        The view will only report edges from these nodes.
(8)                     data : string or bool, optional (default=False)
(12)                        The edge attribute returned in 3-tuple (u, v, ddict[data]).
(12)                        If True, return edge attribute dict in 3-tuple (u, v, ddict).
(12)                        If False, return 2-tuple (u, v).
(8)                     default : value, optional (default=None)
(12)                        Value used for edges that don't have the requested attribute.
(12)                        Only relevant if data is not True or False.
(8)                     Returns
(8)                     -------
(8)                     edges : OutEdgeView
(12)                        A view of edge attributes, usually it iterates over (u, v)
(12)                        or (u, v, d) tuples of edges, but can also be used for
(12)                        attribute lookup as `edges[u, v]['foo']`.
(8)                     See Also
(8)                     --------
(8)                     in_edges, out_edges
(8)                     Notes
(8)                     -----
(8)                     Nodes in nbunch that are not in the graph will be (quietly) ignored.
(8)                     For directed graphs this returns the out-edges.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.DiGraph()  # or MultiDiGraph, etc
(8)                     >>> nx.add_path(G, [0, 1, 2])
(8)                     >>> G.add_edge(2, 3, weight=5)
(8)                     >>> [e for e in G.edges]
(8)                     [(0, 1), (1, 2), (2, 3)]
(8)                     >>> G.edges.data()  # default data is {} (empty dict)
(8)                     OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
(8)                     >>> G.edges.data("weight", default=1)
(8)                     OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
(8)                     >>> G.edges([0, 2])  # only edges originating from these nodes
(8)                     OutEdgeDataView([(0, 1), (2, 3)])
(8)                     >>> G.edges(0)  # only edges from node 0
(8)                     OutEdgeDataView([(0, 1)])
(8)                     """
(8)                     return OutEdgeView(self)
(4)                 @cached_property
(4)                 def out_edges(self):
(8)                     return OutEdgeView(self)
(4)                 out_edges.__doc__ = edges.__doc__
(4)                 @cached_property
(4)                 def in_edges(self):
(8)                     """A view of the in edges of the graph as G.in_edges or G.in_edges().
(8)                     in_edges(self, nbunch=None, data=False, default=None):
(8)                     Parameters
(8)                     ----------
(8)                     nbunch : single node, container, or all nodes (default= all nodes)
(12)                        The view will only report edges incident to these nodes.
(8)                     data : string or bool, optional (default=False)
(12)                        The edge attribute returned in 3-tuple (u, v, ddict[data]).
(12)                        If True, return edge attribute dict in 3-tuple (u, v, ddict).
(12)                        If False, return 2-tuple (u, v).
(8)                     default : value, optional (default=None)
(12)                        Value used for edges that don't have the requested attribute.
(12)                        Only relevant if data is not True or False.
(8)                     Returns
(8)                     -------
(8)                     in_edges : InEdgeView or InEdgeDataView
(12)                        A view of edge attributes, usually it iterates over (u, v)
(12)                        or (u, v, d) tuples of edges, but can also be used for
(12)                        attribute lookup as `edges[u, v]['foo']`.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.DiGraph()
(8)                     >>> G.add_edge(1, 2, color="blue")
```

```
(8)                 >>> G.in_edges()
(8)                 InEdgeView([(1, 2)])
(8)                 >>> G.in_edges(nbunch=2)
(8)                 InEdgeDataView([(1, 2)])
(8)                 See Also
(8)                 --------
(8)                 edges
(8)                 """
(8)                 return InEdgeView(self)
(4)             @cached_property
(4)             def degree(self):
(8)                 """A DegreeView for the Graph as G.degree or G.degree().
(8)                 The node degree is the number of edges adjacent to the node.
(8)                 The weighted node degree is the sum of the edge weights for
(8)                 edges incident to that node.
(8)                 This object provides an iterator for (node, degree) as well as
(8)                 lookup for the degree for a single node.
(8)                 Parameters
(8)                 ----------
(8)                 nbunch : single node, container, or all nodes (default= all nodes)
(12)                    The view will only report edges incident to these nodes.
(8)                 weight : string or None, optional (default=None)
(11)                   The name of an edge attribute that holds the numerical value used
(11)                   as a weight.  If None, then each edge has weight 1.
(11)                   The degree is the sum of the edge weights adjacent to the node.
(8)                 Returns
(8)                 -------
(8)                 DiDegreeView or int
(12)                    If multiple nodes are requested (the default), returns a
`DiDegreeView`
(12)                    mapping nodes to their degree.
(12)                    If a single node is requested, returns the degree of the node as
an integer.
(8)                 See Also
(8)                 --------
(8)                 in_degree, out_degree
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.DiGraph()  # or MultiDiGraph
(8)                 >>> nx.add_path(G, [0, 1, 2, 3])
(8)                 >>> G.degree(0)  # node 0 with degree 1
(8)                 1
(8)                 >>> list(G.degree([0, 1, 2]))
(8)                 [(0, 1), (1, 2), (2, 2)]
(8)                 """
(8)                 return DiDegreeView(self)
(4)             @cached_property
(4)             def in_degree(self):
(8)                 """An InDegreeView for (node, in_degree) or in_degree for single node.
(8)                 The node in_degree is the number of edges pointing to the node.
(8)                 The weighted node degree is the sum of the edge weights for
(8)                 edges incident to that node.
(8)                 This object provides an iteration over (node, in_degree) as well as
(8)                 lookup for the degree for a single node.
(8)                 Parameters
(8)                 ----------
(8)                 nbunch : single node, container, or all nodes (default= all nodes)
(12)                    The view will only report edges incident to these nodes.
(8)                 weight : string or None, optional (default=None)
(11)                   The name of an edge attribute that holds the numerical value used
(11)                   as a weight.  If None, then each edge has weight 1.
(11)                   The degree is the sum of the edge weights adjacent to the node.
(8)                 Returns
(8)                 -------
(8)                 If a single node is requested
(8)                 deg : int
(12)                    In-degree of the node
(8)                 OR if multiple nodes are requested
(8)                 nd_iter : iterator
```

```
(12)                        The iterator returns two-tuples of (node, in-degree).
(8)                     See Also
(8)                     --------
(8)                     degree, out_degree
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.DiGraph()
(8)                     >>> nx.add_path(G, [0, 1, 2, 3])
(8)                     >>> G.in_degree(0)  # node 0 with degree 0
(8)                     0
(8)                     >>> list(G.in_degree([0, 1, 2]))
(8)                     [(0, 0), (1, 1), (2, 1)]
(8)                     """
(8)                     return InDegreeView(self)
(4)                 @cached_property
(4)                 def out_degree(self):
(8)                     """An OutDegreeView for (node, out_degree)
(8)                     The node out_degree is the number of edges pointing out of the node.
(8)                     The weighted node degree is the sum of the edge weights for
(8)                     edges incident to that node.
(8)                     This object provides an iterator over (node, out_degree) as well as
(8)                     lookup for the degree for a single node.
(8)                     Parameters
(8)                     ----------
(8)                     nbunch : single node, container, or all nodes (default= all nodes)
(12)                        The view will only report edges incident to these nodes.
(8)                     weight : string or None, optional (default=None)
(11)                       The name of an edge attribute that holds the numerical value used
(11)                       as a weight.  If None, then each edge has weight 1.
(11)                       The degree is the sum of the edge weights adjacent to the node.
(8)                     Returns
(8)                     -------
(8)                     If a single node is requested
(8)                     deg : int
(12)                        Out-degree of the node
(8)                     OR if multiple nodes are requested
(8)                     nd_iter : iterator
(12)                        The iterator returns two-tuples of (node, out-degree).
(8)                     See Also
(8)                     --------
(8)                     degree, in_degree
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.DiGraph()
(8)                     >>> nx.add_path(G, [0, 1, 2, 3])
(8)                     >>> G.out_degree(0)  # node 0 with degree 1
(8)                     1
(8)                     >>> list(G.out_degree([0, 1, 2]))
(8)                     [(0, 1), (1, 1), (2, 1)]
(8)                     """
(8)                     return OutDegreeView(self)
(4)                 def clear(self):
(8)                     """Remove all nodes and edges from the graph.
(8)                     This also removes the name, and all graph, node, and edge attributes.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.clear()
(8)                     >>> list(G.nodes)
(8)                     []
(8)                     >>> list(G.edges)
(8)                     []
(8)                     """
(8)                     self._succ.clear()
(8)                     self._pred.clear()
(8)                     self._node.clear()
(8)                     self.graph.clear()
(8)                     nx._clear_cache(self)
(4)                 def clear_edges(self):
```

```
        """Remove all edges from the graph without altering nodes.
        Examples
        --------
        >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.clear_edges()
        >>> list(G.nodes)
        [0, 1, 2, 3]
        >>> list(G.edges)
        []
        """
        for predecessor_dict in self._pred.values():
            predecessor_dict.clear()
        for successor_dict in self._succ.values():
            successor_dict.clear()
        nx._clear_cache(self)
    def is_multigraph(self):
        """Returns True if graph is a multigraph, False otherwise."""
        return False
    def is_directed(self):
        """Returns True if graph is directed, False otherwise."""
        return True
    def to_undirected(self, reciprocal=False, as_view=False):
        """Returns an undirected representation of the digraph.
        Parameters
        ----------
        reciprocal : bool (optional)
          If True only keep edges that appear in both directions
          in the original digraph.
        as_view : bool (optional, default=False)
          If True return an undirected view of the original directed graph.
        Returns
        -------
        G : Graph
            An undirected graph with the same name and nodes and
            with edge (u, v, data) if either (u, v, data) or (v, u, data)
            is in the digraph.  If both edges exist in digraph and
            their edge data is different, only one edge is created
            with an arbitrary choice of which edge data to use.
            You must check and correct for this manually if desired.
        See Also
        --------
        Graph, copy, add_edge, add_edges_from
        Notes
        -----
        If edges in both directions (u, v) and (v, u) exist in the
        graph, attributes for the new undirected edge will be a combination of
        the attributes of the directed edges.  The edge data is updated
        in the (arbitrary) order that the edges are encountered.  For
        more customized control of the edge attributes use add_edge().
        This returns a "deepcopy" of the edge, node, and
        graph attributes which attempts to completely copy
        all of the data and references.
        This is in contrast to the similar G=DiGraph(D) which returns a
        shallow copy of the data.
        See the Python copy module for more information on shallow
        and deep copies, https://docs.python.org/3/library/copy.html.
        Warning: If you have subclassed DiGraph to use dict-like objects
        in the data structure, those changes do not transfer to the
        Graph created by this method.
        Examples
        --------
        >>> G = nx.path_graph(2)  # or MultiGraph, etc
        >>> H = G.to_directed()
        >>> list(H.edges)
        [(0, 1), (1, 0)]
        >>> G2 = H.to_undirected()
        >>> list(G2.edges)
        [(0, 1)]
        """
```

```
1035: (8)                   graph_class = self.to_undirected_class()
1036: (8)                   if as_view is True:
1037: (12)                      return nx.graphviews.generic_graph_view(self, graph_class)
1038: (8)                   G = graph_class()
1039: (8)                   G.graph.update(deepcopy(self.graph))
1040: (8)                   G.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
1041: (8)                   if reciprocal is True:
1042: (12)                      G.add_edges_from(
1043: (16)                          (u, v, deepcopy(d))
1044: (16)                          for u, nbrs in self._adj.items()
1045: (16)                          for v, d in nbrs.items()
1046: (16)                          if v in self._pred[u]
1047: (12)                      )
1048: (8)                   else:
1049: (12)                      G.add_edges_from(
1050: (16)                          (u, v, deepcopy(d))
1051: (16)                          for u, nbrs in self._adj.items()
1052: (16)                          for v, d in nbrs.items()
1053: (12)                      )
1054: (8)                   return G
1055: (4)               def reverse(self, copy=True):
1056: (8)                   """Returns the reverse of the graph.
1057: (8)                   The reverse is a graph with the same nodes and edges
1058: (8)                   but with the directions of the edges reversed.
1059: (8)                   Parameters
1060: (8)                   ----------
1061: (8)                   copy : bool optional (default=True)
1062: (12)                      If True, return a new DiGraph holding the reversed edges.
1063: (12)                      If False, the reverse graph is created using a view of
1064: (12)                      the original graph.
1065: (8)                   """
1066: (8)                   if copy:
1067: (12)                      H = self.__class__()
1068: (12)                      H.graph.update(deepcopy(self.graph))
1069: (12)                      H.add_nodes_from((n, deepcopy(d)) for n, d in self.nodes.items())
1070: (12)                      H.add_edges_from((v, u, deepcopy(d)) for u, v, d in
self.edges(data=True))
1071: (12)                      return H
1072: (8)                   return nx.reverse_view(self)

----------------------------------------

File 391 - filters.py:

1: (0)              """Filter factories to hide or show sets of nodes and edges.
2: (0)              These filters return the function used when creating `SubGraph`.
3: (0)              """
4: (0)              __all__ = [
5: (4)                  "no_filter",
6: (4)                  "hide_nodes",
7: (4)                  "hide_edges",
8: (4)                  "hide_multiedges",
9: (4)                  "hide_diedges",
10: (4)                 "hide_multidiedges",
11: (4)                 "show_nodes",
12: (4)                 "show_edges",
13: (4)                 "show_multiedges",
14: (4)                 "show_diedges",
15: (4)                 "show_multidiedges",
16: (0)             ]
17: (0)             def no_filter(*items):
18: (4)                 """Returns a filter function that always evaluates to True."""
19: (4)                 return True
20: (0)             def hide_nodes(nodes):
21: (4)                 """Returns a filter function that hides specific nodes."""
22: (4)                 nodes = set(nodes)
23: (4)                 return lambda node: node not in nodes
24: (0)             def hide_diedges(edges):
25: (4)                 """Returns a filter function that hides specific directed edges."""
```

```
26: (4)                 edges = {(u, v) for u, v in edges}
27: (4)                 return lambda u, v: (u, v) not in edges
28: (0)             def hide_edges(edges):
29: (4)                 """Returns a filter function that hides specific undirected edges."""
30: (4)                 alledges = set(edges) | {(v, u) for (u, v) in edges}
31: (4)                 return lambda u, v: (u, v) not in alledges
32: (0)             def hide_multidiedges(edges):
33: (4)                 """Returns a filter function that hides specific multi-directed edges."""
34: (4)                 edges = {(u, v, k) for u, v, k in edges}
35: (4)                 return lambda u, v, k: (u, v, k) not in edges
36: (0)             def hide_multiedges(edges):
37: (4)                 """Returns a filter function that hides specific multi-undirected
edges."""
38: (4)                 alledges = set(edges) | {(v, u, k) for (u, v, k) in edges}
39: (4)                 return lambda u, v, k: (u, v, k) not in alledges
40: (0)             class show_nodes:
41: (4)                 """Filter class to show specific nodes."""
42: (4)                 def __init__(self, nodes):
43: (8)                     self.nodes = set(nodes)
44: (4)                 def __call__(self, node):
45: (8)                     return node in self.nodes
46: (0)             def show_diedges(edges):
47: (4)                 """Returns a filter function that shows specific directed edges."""
48: (4)                 edges = {(u, v) for u, v in edges}
49: (4)                 return lambda u, v: (u, v) in edges
50: (0)             def show_edges(edges):
51: (4)                 """Returns a filter function that shows specific undirected edges."""
52: (4)                 alledges = set(edges) | {(v, u) for (u, v) in edges}
53: (4)                 return lambda u, v: (u, v) in alledges
54: (0)             def show_multidiedges(edges):
55: (4)                 """Returns a filter function that shows specific multi-directed edges."""
56: (4)                 edges = {(u, v, k) for u, v, k in edges}
57: (4)                 return lambda u, v, k: (u, v, k) in edges
58: (0)             def show_multiedges(edges):
59: (4)                 """Returns a filter function that shows specific multi-undirected
edges."""
60: (4)                 alledges = set(edges) | {(v, u, k) for (u, v, k) in edges}
61: (4)                 return lambda u, v, k: (u, v, k) in alledges

----------------------------------------

File 392 - function.py:

1: (0)              """Functional interface to graph methods and assorted utilities.
2: (0)              """
3: (0)              from collections import Counter
4: (0)              from itertools import chain
5: (0)              import networkx as nx
6: (0)              from networkx.utils import not_implemented_for, pairwise
7: (0)              __all__ = [
8: (4)                  "nodes",
9: (4)                  "edges",
10: (4)                 "degree",
11: (4)                 "degree_histogram",
12: (4)                 "neighbors",
13: (4)                 "number_of_nodes",
14: (4)                 "number_of_edges",
15: (4)                 "density",
16: (4)                 "is_directed",
17: (4)                 "freeze",
18: (4)                 "is_frozen",
19: (4)                 "subgraph",
20: (4)                 "induced_subgraph",
21: (4)                 "edge_subgraph",
22: (4)                 "restricted_view",
23: (4)                 "to_directed",
24: (4)                 "to_undirected",
25: (4)                 "add_star",
26: (4)                 "add_path",
```

```
 27: (4)                 "add_cycle",
 28: (4)                 "create_empty_copy",
 29: (4)                 "set_node_attributes",
 30: (4)                 "get_node_attributes",
 31: (4)                 "set_edge_attributes",
 32: (4)                 "get_edge_attributes",
 33: (4)                 "all_neighbors",
 34: (4)                 "non_neighbors",
 35: (4)                 "non_edges",
 36: (4)                 "common_neighbors",
 37: (4)                 "is_weighted",
 38: (4)                 "is_negatively_weighted",
 39: (4)                 "is_empty",
 40: (4)                 "selfloop_edges",
 41: (4)                 "nodes_with_selfloops",
 42: (4)                 "number_of_selfloops",
 43: (4)                 "path_weight",
 44: (4)                 "is_path",
 45: (0)             ]
 46: (0)             def nodes(G):
 47: (4)                 """Returns a NodeView over the graph nodes.
 48: (4)                 This function wraps the :func:`G.nodes <networkx.Graph.nodes>` property.
 49: (4)                 """
 50: (4)                 return G.nodes()
 51: (0)             def edges(G, nbunch=None):
 52: (4)                 """Returns an edge view of edges incident to nodes in nbunch.
 53: (4)                 Return all edges if nbunch is unspecified or nbunch=None.
 54: (4)                 For digraphs, edges=out_edges
 55: (4)                 This function wraps the :func:`G.edges <networkx.Graph.edges>` property.
 56: (4)                 """
 57: (4)                 return G.edges(nbunch)
 58: (0)             def degree(G, nbunch=None, weight=None):
 59: (4)                 """Returns a degree view of single node or of nbunch of nodes.
 60: (4)                 If nbunch is omitted, then return degrees of *all* nodes.
 61: (4)                 This function wraps the :func:`G.degree <networkx.Graph.degree>` property.
 62: (4)                 """
 63: (4)                 return G.degree(nbunch, weight)
 64: (0)             def neighbors(G, n):
 65: (4)                 """Returns an iterator over all neighbors of node n.
 66: (4)                 This function wraps the :func:`G.neighbors <networkx.Graph.neighbors>`
 function.
 67: (4)                 """
 68: (4)                 return G.neighbors(n)
 69: (0)             def number_of_nodes(G):
 70: (4)                 """Returns the number of nodes in the graph.
 71: (4)                 This function wraps the :func:`G.number_of_nodes
 <networkx.Graph.number_of_nodes>` function.
 72: (4)                 """
 73: (4)                 return G.number_of_nodes()
 74: (0)             def number_of_edges(G):
 75: (4)                 """Returns the number of edges in the graph.
 76: (4)                 This function wraps the :func:`G.number_of_edges
 <networkx.Graph.number_of_edges>` function.
 77: (4)                 """
 78: (4)                 return G.number_of_edges()
 79: (0)             def density(G):
 80: (4)                 r"""Returns the density of a graph.
 81: (4)                 The density for undirected graphs is
 82: (4)                 .. math::
 83: (7)                    d = \frac{2m}{n(n-1)},
 84: (4)                 and for directed graphs is
 85: (4)                 .. math::
 86: (7)                    d = \frac{m}{n(n-1)},
 87: (4)                 where `n` is the number of nodes and `m`  is the number of edges in `G`.
 88: (4)                 Notes
 89: (4)                 -----
 90: (4)                 The density is 0 for a graph without edges and 1 for a complete graph.
 91: (4)                 The density of multigraphs can be higher than 1.
 92: (4)                 Self loops are counted in the total number of edges so graphs with self
```

```
93: (4)                 loops can have density higher than 1.
94: (4)                 """
95: (4)                 n = number_of_nodes(G)
96: (4)                 m = number_of_edges(G)
97: (4)                 if m == 0 or n <= 1:
98: (8)                     return 0
99: (4)                 d = m / (n * (n - 1))
100: (4)                if not G.is_directed():
101: (8)                    d *= 2
102: (4)                return d
103: (0)            def degree_histogram(G):
104: (4)                """Returns a list of the frequency of each degree value.
105: (4)                Parameters
106: (4)                ----------
107: (4)                G : Networkx graph
108: (7)                   A graph
109: (4)                Returns
110: (4)                -------
111: (4)                hist : list
112: (7)                   A list of frequencies of degrees.
113: (7)                   The degree values are the index in the list.
114: (4)                Notes
115: (4)                -----
116: (4)                Note: the bins are width one, hence len(list) can be large
117: (4)                (Order(number_of_edges))
118: (4)                """
119: (4)                counts = Counter(d for n, d in G.degree())
120: (4)                return [counts.get(i, 0) for i in range(max(counts) + 1 if counts else 0)]
121: (0)            def is_directed(G):
122: (4)                """Return True if graph is directed."""
123: (4)                return G.is_directed()
124: (0)            def frozen(*args, **kwargs):
125: (4)                """Dummy method for raising errors when trying to modify frozen graphs"""
126: (4)                raise nx.NetworkXError("Frozen graph can't be modified")
127: (0)            def freeze(G):
128: (4)                """Modify graph to prevent further change by adding or removing
129: (4)                nodes or edges.
130: (4)                Node and edge data can still be modified.
131: (4)                Parameters
132: (4)                ----------
133: (4)                G : graph
134: (6)                  A NetworkX graph
135: (4)                Examples
136: (4)                --------
137: (4)                >>> G = nx.path_graph(4)
138: (4)                >>> G = nx.freeze(G)
139: (4)                >>> try:
140: (4)                ...     G.add_edge(4, 5)
141: (4)                ... except nx.NetworkXError as err:
142: (4)                ...     print(str(err))
143: (4)                Frozen graph can't be modified
144: (4)                Notes
145: (4)                -----
146: (4)                To "unfreeze" a graph you must make a copy by creating a new graph object:
147: (4)                >>> graph = nx.path_graph(4)
148: (4)                >>> frozen_graph = nx.freeze(graph)
149: (4)                >>> unfrozen_graph = nx.Graph(frozen_graph)
150: (4)                >>> nx.is_frozen(unfrozen_graph)
151: (4)                False
152: (4)                See Also
153: (4)                --------
154: (4)                is_frozen
155: (4)                """
156: (4)                G.add_node = frozen
157: (4)                G.add_nodes_from = frozen
158: (4)                G.remove_node = frozen
159: (4)                G.remove_nodes_from = frozen
160: (4)                G.add_edge = frozen
161: (4)                G.add_edges_from = frozen
```

```
(4)                 G.add_weighted_edges_from = frozen
(4)                 G.remove_edge = frozen
(4)                 G.remove_edges_from = frozen
(4)                 G.clear = frozen
(4)                 G.clear_edges = frozen
(4)                 G.frozen = True
(4)                 return G
(0)             def is_frozen(G):
(4)                 """Returns True if graph is frozen.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(6)                   A NetworkX graph
(4)                 See Also
(4)                 --------
(4)                 freeze
(4)                 """
(4)                 try:
(8)                     return G.frozen
(4)                 except AttributeError:
(8)                     return False
(0)             def add_star(G_to_add_to, nodes_for_star, **attr):
(4)                 """Add a star to Graph G_to_add_to.
(4)                 The first node in `nodes_for_star` is the middle of the star.
(4)                 It is connected to all other nodes.
(4)                 Parameters
(4)                 ----------
(4)                 G_to_add_to : graph
(8)                     A NetworkX graph
(4)                 nodes_for_star : iterable container
(8)                     A container of nodes.
(4)                 attr : keyword arguments, optional (default= no attributes)
(8)                     Attributes to add to every edge in star.
(4)                 See Also
(4)                 --------
(4)                 add_path, add_cycle
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> nx.add_star(G, [0, 1, 2, 3])
(4)                 >>> nx.add_star(G, [10, 11, 12], weight=2)
(4)                 """
(4)                 nlist = iter(nodes_for_star)
(4)                 try:
(8)                     v = next(nlist)
(4)                 except StopIteration:
(8)                     return
(4)                 G_to_add_to.add_node(v)
(4)                 edges = ((v, n) for n in nlist)
(4)                 G_to_add_to.add_edges_from(edges, **attr)
(0)             def add_path(G_to_add_to, nodes_for_path, **attr):
(4)                 """Add a path to the Graph G_to_add_to.
(4)                 Parameters
(4)                 ----------
(4)                 G_to_add_to : graph
(8)                     A NetworkX graph
(4)                 nodes_for_path : iterable container
(8)                     A container of nodes.  A path will be constructed from
(8)                     the nodes (in order) and added to the graph.
(4)                 attr : keyword arguments, optional (default= no attributes)
(8)                     Attributes to add to every edge in path.
(4)                 See Also
(4)                 --------
(4)                 add_star, add_cycle
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> nx.add_path(G, [0, 1, 2, 3])
(4)                 >>> nx.add_path(G, [10, 11, 12], weight=7)
```

```
(4)             """
(4)             nlist = iter(nodes_for_path)
(4)             try:
(8)                 first_node = next(nlist)
(4)             except StopIteration:
(8)                 return
(4)             G_to_add_to.add_node(first_node)
(4)             G_to_add_to.add_edges_from(pairwise(chain((first_node,), nlist)), **attr)
(0)         def add_cycle(G_to_add_to, nodes_for_cycle, **attr):
(4)             """Add a cycle to the Graph G_to_add_to.
(4)             Parameters
(4)             ----------
(4)             G_to_add_to : graph
(8)                 A NetworkX graph
(4)             nodes_for_cycle: iterable container
(8)                 A container of nodes.  A cycle will be constructed from
(8)                 the nodes (in order) and added to the graph.
(4)             attr : keyword arguments, optional (default= no attributes)
(8)                 Attributes to add to every edge in cycle.
(4)             See Also
(4)             --------
(4)             add_path, add_star
(4)             Examples
(4)             --------
(4)             >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
(4)             >>> nx.add_cycle(G, [0, 1, 2, 3])
(4)             >>> nx.add_cycle(G, [10, 11, 12], weight=7)
(4)             """
(4)             nlist = iter(nodes_for_cycle)
(4)             try:
(8)                 first_node = next(nlist)
(4)             except StopIteration:
(8)                 return
(4)             G_to_add_to.add_node(first_node)
(4)             G_to_add_to.add_edges_from(
(8)                 pairwise(chain((first_node,), nlist), cyclic=True), **attr
(4)             )
(0)         def subgraph(G, nbunch):
(4)             """Returns the subgraph induced on nodes in nbunch.
(4)             Parameters
(4)             ----------
(4)             G : graph
(7)                A NetworkX graph
(4)             nbunch : list, iterable
(7)                A container of nodes that will be iterated through once (thus
(7)                it should be an iterator or be iterable).  Each element of the
(7)                container should be a valid node type: any hashable type except
(7)                None.  If nbunch is None, return all edges data in the graph.
(7)                Nodes in nbunch that are not in the graph will be (quietly)
(7)                ignored.
(4)             Notes
(4)             -----
(4)             subgraph(G) calls G.subgraph()
(4)             """
(4)             return G.subgraph(nbunch)
(0)         def induced_subgraph(G, nbunch):
(4)             """Returns a SubGraph view of `G` showing only nodes in nbunch.
(4)             The induced subgraph of a graph on a set of nodes N is the
(4)             graph with nodes N and edges from G which have both ends in N.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(4)             nbunch : node, container of nodes or None (for all nodes)
(4)             Returns
(4)             -------
(4)             subgraph : SubGraph View
(8)                 A read-only view of the subgraph in `G` induced by the nodes.
(8)                 Changes to the graph `G` will be reflected in the view.
(4)             Notes
```

```
(4)                 -----
(4)                 To create a mutable subgraph with its own copies of nodes
(4)                 edges and attributes use `subgraph.copy()` or `Graph(subgraph)`
(4)                 For an inplace reduction of a graph to a subgraph you can remove nodes:
(4)                 `G.remove_nodes_from(n in G if n not in set(nbunch))`
(4)                 If you are going to compute subgraphs of your subgraphs you could
(4)                 end up with a chain of views that can be very slow once the chain
(4)                 has about 15 views in it. If they are all induced subgraphs, you
(4)                 can short-cut the chain by making them all subgraphs of the original
(4)                 graph. The graph class method `G.subgraph` does this when `G` is
(4)                 a subgraph. In contrast, this function allows you to choose to build
(4)                 chains or not, as you wish. The returned subgraph is a view on `G`.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(4)                 >>> H = nx.induced_subgraph(G, [0, 1, 3])
(4)                 >>> list(H.edges)
(4)                 [(0, 1)]
(4)                 >>> list(H.nodes)
(4)                 [0, 1, 3]
(4)                 """
(4)                 induced_nodes = nx.filters.show_nodes(G.nbunch_iter(nbunch))
(4)                 return nx.subgraph_view(G, filter_node=induced_nodes)
(0)             def edge_subgraph(G, edges):
(4)                 """Returns a view of the subgraph induced by the specified edges.
(4)                 The induced subgraph contains each edge in `edges` and each
(4)                 node incident to any of those edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 edges : iterable
(8)                     An iterable of edges. Edges not present in `G` are ignored.
(4)                 Returns
(4)                 -------
(4)                 subgraph : SubGraph View
(8)                     A read-only edge-induced subgraph of `G`.
(8)                     Changes to `G` are reflected in the view.
(4)                 Notes
(4)                 -----
(4)                 To create a mutable subgraph with its own copies of nodes
(4)                 edges and attributes use `subgraph.copy()` or `Graph(subgraph)`
(4)                 If you create a subgraph of a subgraph recursively you can end up
(4)                 with a chain of subgraphs that becomes very slow with about 15
(4)                 nested subgraph views. Luckily the edge_subgraph filter nests
(4)                 nicely so you can use the original graph as G in this function
(4)                 to avoid chains. We do not rule out chains programmatically so
(4)                 that odd cases like an `edge_subgraph` of a `restricted_view`
(4)                 can be created.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(5)
(4)                 >>> H = G.edge_subgraph([(0, 1), (3, 4)])
(4)                 >>> list(H.nodes)
(4)                 [0, 1, 3, 4]
(4)                 >>> list(H.edges)
(4)                 [(0, 1), (3, 4)]
(4)                 """
(4)                 nxf = nx.filters
(4)                 edges = set(edges)
(4)                 nodes = set()
(4)                 for e in edges:
(8)                     nodes.update(e[:2])
(4)                 induced_nodes = nxf.show_nodes(nodes)
(4)                 if G.is_multigraph():
(8)                     if G.is_directed():
(12)                        induced_edges = nxf.show_multidiedges(edges)
(8)                     else:
(12)                        induced_edges = nxf.show_multiedges(edges)
(4)                 else:
```

```
369: (8)                    if G.is_directed():
370: (12)                       induced_edges = nxf.show_diedges(edges)
371: (8)                    else:
372: (12)                       induced_edges = nxf.show_edges(edges)
373: (4)                return nx.subgraph_view(G, filter_node=induced_nodes,
filter_edge=induced_edges)
374: (0)            def restricted_view(G, nodes, edges):
375: (4)                """Returns a view of `G` with hidden nodes and edges.
376: (4)                The resulting subgraph filters out node `nodes` and edges `edges`.
377: (4)                Filtered out nodes also filter out any of their edges.
378: (4)                Parameters
379: (4)                ----------
380: (4)                G : NetworkX Graph
381: (4)                nodes : iterable
382: (8)                    An iterable of nodes. Nodes not present in `G` are ignored.
383: (4)                edges : iterable
384: (8)                    An iterable of edges. Edges not present in `G` are ignored.
385: (4)                Returns
386: (4)                -------
387: (4)                subgraph : SubGraph View
388: (8)                    A read-only restricted view of `G` filtering out nodes and edges.
389: (8)                    Changes to `G` are reflected in the view.
390: (4)                Notes
391: (4)                -----
392: (4)                To create a mutable subgraph with its own copies of nodes
393: (4)                edges and attributes use `subgraph.copy()` or `Graph(subgraph)`
394: (4)                If you create a subgraph of a subgraph recursively you may end up
395: (4)                with a chain of subgraph views. Such chains can get quite slow
396: (4)                for lengths near 15. To avoid long chains, try to make your subgraph
397: (4)                based on the original graph.  We do not rule out chains programmatically
398: (4)                so that odd cases like an `edge_subgraph` of a `restricted_view`
399: (4)                can be created.
400: (4)                Examples
401: (4)                --------
402: (4)                >>> G = nx.path_graph(5)
403: (4)                >>> H = nx.restricted_view(G, [0], [(1, 2), (3, 4)])
404: (4)                >>> list(H.nodes)
405: (4)                [1, 2, 3, 4]
406: (4)                >>> list(H.edges)
407: (4)                [(2, 3)]
408: (4)                """
409: (4)                nxf = nx.filters
410: (4)                hide_nodes = nxf.hide_nodes(nodes)
411: (4)                if G.is_multigraph():
412: (8)                    if G.is_directed():
413: (12)                       hide_edges = nxf.hide_multidiedges(edges)
414: (8)                    else:
415: (12)                       hide_edges = nxf.hide_multiedges(edges)
416: (4)                else:
417: (8)                    if G.is_directed():
418: (12)                       hide_edges = nxf.hide_diedges(edges)
419: (8)                    else:
420: (12)                       hide_edges = nxf.hide_edges(edges)
421: (4)                return nx.subgraph_view(G, filter_node=hide_nodes, filter_edge=hide_edges)
422: (0)            def to_directed(graph):
423: (4)                """Returns a directed view of the graph `graph`.
424: (4)                Identical to graph.to_directed(as_view=True)
425: (4)                Note that graph.to_directed defaults to `as_view=False`
426: (4)                while this function always provides a view.
427: (4)                """
428: (4)                return graph.to_directed(as_view=True)
429: (0)            def to_undirected(graph):
430: (4)                """Returns an undirected view of the graph `graph`.
431: (4)                Identical to graph.to_undirected(as_view=True)
432: (4)                Note that graph.to_undirected defaults to `as_view=False`
433: (4)                while this function always provides a view.
434: (4)                """
435: (4)                return graph.to_undirected(as_view=True)
436: (0)            def create_empty_copy(G, with_data=True):
```

```
(4)                 """Returns a copy of the graph G with all of the edges removed.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph
(4)                 with_data :  bool (default=True)
(7)                    Propagate Graph and Nodes data to the new graph.
(4)                 See Also
(4)                 --------
(4)                 empty_graph
(4)                 """
(4)                 H = G.__class__()
(4)                 H.add_nodes_from(G.nodes(data=with_data))
(4)                 if with_data:
(8)                     H.graph.update(G.graph)
(4)                 return H
(0)             def set_node_attributes(G, values, name=None):
(4)                 """Sets node attributes from a given value or dictionary of values.
(4)                 .. Warning:: The call order of arguments `values` and `name`
(8)                     switched between v1.x & v2.x.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 values : scalar value, dict-like
(8)                     What the node attribute should be set to.  If `values` is
(8)                     not a dictionary, then it is treated as a single attribute value
(8)                     that is then applied to every node in `G`.  This means that if
(8)                     you provide a mutable object, like a list, updates to that object
(8)                     will be reflected in the node attribute for every node.
(8)                     The attribute name will be `name`.
(8)                     If `values` is a dict or a dict of dict, it should be keyed
(8)                     by node to either an attribute value or a dict of attribute key/value
(8)                     pairs used to update the node's attributes.
(4)                 name : string (optional, default=None)
(8)                     Name of the node attribute to set if values is a scalar.
(4)                 Examples
(4)                 --------
(4)                 After computing some property of the nodes of a graph, you may want
(4)                 to assign a node attribute to store the value of that property for
(4)                 each node::
(8)                     >>> G = nx.path_graph(3)
(8)                     >>> bb = nx.betweenness_centrality(G)
(8)                     >>> isinstance(bb, dict)
(8)                     True
(8)                     >>> nx.set_node_attributes(G, bb, "betweenness")
(8)                     >>> G.nodes[1]["betweenness"]
(8)                     1.0
(4)                 If you provide a list as the second argument, updates to the list
(4)                 will be reflected in the node attribute for each node::
(8)                     >>> G = nx.path_graph(3)
(8)                     >>> labels = []
(8)                     >>> nx.set_node_attributes(G, labels, "labels")
(8)                     >>> labels.append("foo")
(8)                     >>> G.nodes[0]["labels"]
(8)                     ['foo']
(8)                     >>> G.nodes[1]["labels"]
(8)                     ['foo']
(8)                     >>> G.nodes[2]["labels"]
(8)                     ['foo']
(4)                 If you provide a dictionary of dictionaries as the second argument,
(4)                 the outer dictionary is assumed to be keyed by node to an inner
(4)                 dictionary of node attributes for that node::
(8)                     >>> G = nx.path_graph(3)
(8)                     >>> attrs = {0: {"attr1": 20, "attr2": "nothing"}, 1: {"attr2": 3}}
(8)                     >>> nx.set_node_attributes(G, attrs)
(8)                     >>> G.nodes[0]["attr1"]
(8)                     20
(8)                     >>> G.nodes[0]["attr2"]
(8)                     'nothing'
```

```
(8)                 >>> G.nodes[1]["attr2"]
(8)                 3
(8)                 >>> G.nodes[2]
(8)                 {}
(4)             Note that if the dictionary contains nodes that are not in `G`, the
(4)             values are silently ignored::
(8)                 >>> G = nx.Graph()
(8)                 >>> G.add_node(0)
(8)                 >>> nx.set_node_attributes(G, {0: "red", 1: "blue"}, name="color")
(8)                 >>> G.nodes[0]["color"]
(8)                 'red'
(8)                 >>> 1 in G.nodes
(8)                 False
(4)             """
(4)             if name is not None:  # `values` must not be a dict of dict
(8)                 try:  # `values` is a dict
(12)                    for n, v in values.items():
(16)                        try:
(20)                            G.nodes[n][name] = values[n]
(16)                        except KeyError:
(20)                            pass
(8)                 except AttributeError:  # `values` is a constant
(12)                    for n in G:
(16)                        G.nodes[n][name] = values
(4)             else:  # `values` must be dict of dict
(8)                 for n, d in values.items():
(12)                    try:
(16)                        G.nodes[n].update(d)
(12)                    except KeyError:
(16)                        pass
(4)             nx._clear_cache(G)
(0)         def get_node_attributes(G, name, default=None):
(4)             """Get node attributes from graph
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(4)             name : string
(7)                Attribute name
(4)             default: object (default=None)
(7)                Default value of the node attribute if there is no value set for that
(7)                node in graph. If `None` then nodes without this attribute are not
(7)                included in the returned dict.
(4)             Returns
(4)             -------
(4)             Dictionary of attributes keyed by node.
(4)             Examples
(4)             --------
(4)             >>> G = nx.Graph()
(4)             >>> G.add_nodes_from([1, 2, 3], color="red")
(4)             >>> color = nx.get_node_attributes(G, "color")
(4)             >>> color[1]
(4)             'red'
(4)             >>> G.add_node(4)
(4)             >>> color = nx.get_node_attributes(G, "color", default="yellow")
(4)             >>> color[4]
(4)             'yellow'
(4)             """
(4)             if default is not None:
(8)                 return {n: d.get(name, default) for n, d in G.nodes.items()}
(4)             return {n: d[name] for n, d in G.nodes.items() if name in d}
(0)         def set_edge_attributes(G, values, name=None):
(4)             """Sets edge attributes from a given value or dictionary of values.
(4)             .. Warning:: The call order of arguments `values` and `name`
(8)                 switched between v1.x & v2.x.
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(4)             values : scalar value, dict-like
(8)                 What the edge attribute should be set to.  If `values` is
```

```
(8)                 not a dictionary, then it is treated as a single attribute value
(8)                 that is then applied to every edge in `G`.  This means that if
(8)                 you provide a mutable object, like a list, updates to that object
(8)                 will be reflected in the edge attribute for each edge.  The attribute
(8)                 name will be `name`.
(8)                 If `values` is a dict or a dict of dict, it should be keyed
(8)                 by edge tuple to either an attribute value or a dict of attribute
(8)                 key/value pairs used to update the edge's attributes.
(8)                 For multigraphs, the edge tuples must be of the form ``(u, v, key)``,
(8)                 where `u` and `v` are nodes and `key` is the edge key.
(8)                 For non-multigraphs, the keys must be tuples of the form ``(u, v)``.
(4)             name : string (optional, default=None)
(8)                 Name of the edge attribute to set if values is a scalar.
(4)             Examples
(4)             --------
(4)             After computing some property of the edges of a graph, you may want
(4)             to assign a edge attribute to store the value of that property for
(4)             each edge::
(8)                 >>> G = nx.path_graph(3)
(8)                 >>> bb = nx.edge_betweenness_centrality(G, normalized=False)
(8)                 >>> nx.set_edge_attributes(G, bb, "betweenness")
(8)                 >>> G.edges[1, 2]["betweenness"]
(8)                 2.0
(4)             If you provide a list as the second argument, updates to the list
(4)             will be reflected in the edge attribute for each edge::
(8)                 >>> labels = []
(8)                 >>> nx.set_edge_attributes(G, labels, "labels")
(8)                 >>> labels.append("foo")
(8)                 >>> G.edges[0, 1]["labels"]
(8)                 ['foo']
(8)                 >>> G.edges[1, 2]["labels"]
(8)                 ['foo']
(4)             If you provide a dictionary of dictionaries as the second argument,
(4)             the entire dictionary will be used to update edge attributes::
(8)                 >>> G = nx.path_graph(3)
(8)                 >>> attrs = {(0, 1): {"attr1": 20, "attr2": "nothing"}, (1, 2):
{"attr2": 3}}
(8)                 >>> nx.set_edge_attributes(G, attrs)
(8)                 >>> G[0][1]["attr1"]
(8)                 20
(8)                 >>> G[0][1]["attr2"]
(8)                 'nothing'
(8)                 >>> G[1][2]["attr2"]
(8)                 3
(4)             The attributes of one Graph can be used to set those of another.
(8)                 >>> H = nx.path_graph(3)
(8)                 >>> nx.set_edge_attributes(H, G.edges)
(4)             Note that if the dict contains edges that are not in `G`, they are
(4)             silently ignored::
(8)                 >>> G = nx.Graph([(0, 1)])
(8)                 >>> nx.set_edge_attributes(G, {(1, 2): {"weight": 2.0}})
(8)                 >>> (1, 2) in G.edges()
(8)                 False
(4)             For multigraphs, the `values` dict is expected to be keyed by 3-tuples
(4)             including the edge key::
(8)                 >>> MG = nx.MultiGraph()
(8)                 >>> edges = [(0, 1), (0, 1)]
(8)                 >>> MG.add_edges_from(edges)  # Returns list of edge keys
(8)                 [0, 1]
(8)                 >>> attributes = {(0, 1, 0): {"cost": 21}, (0, 1, 1): {"cost": 7}}
(8)                 >>> nx.set_edge_attributes(MG, attributes)
(8)                 >>> MG[0][1][0]["cost"]
(8)                 21
(8)                 >>> MG[0][1][1]["cost"]
(8)                 7
(4)             If MultiGraph attributes are desired for a Graph, you must convert the 3-
tuple
(4)             multiedge to a 2-tuple edge and the last multiedge's attribute value will
(4)             overwrite the previous values. Continuing from the previous case we get::
```

```
(8)                     >>> H = nx.path_graph([0, 1, 2])
(8)                     >>> nx.set_edge_attributes(H, {(u, v): ed for u, v, ed in
MG.edges.data()})
(8)                     >>> nx.get_edge_attributes(H, "cost")
(8)                     {(0, 1): 7}
(4)                 """
(4)                 if name is not None:
(8)                     try:
(12)                        if G.is_multigraph():
(16)                            for (u, v, key), value in values.items():
(20)                                try:
(24)                                    G._adj[u][v][key][name] = value
(20)                                except KeyError:
(24)                                    pass
(12)                        else:
(16)                            for (u, v), value in values.items():
(20)                                try:
(24)                                    G._adj[u][v][name] = value
(20)                                except KeyError:
(24)                                    pass
(8)                     except AttributeError:
(12)                        for u, v, data in G.edges(data=True):
(16)                            data[name] = values
(4)                 else:
(8)                     if G.is_multigraph():
(12)                        for (u, v, key), d in values.items():
(16)                            try:
(20)                                G._adj[u][v][key].update(d)
(16)                            except KeyError:
(20)                                pass
(8)                     else:
(12)                        for (u, v), d in values.items():
(16)                            try:
(20)                                G._adj[u][v].update(d)
(16)                            except KeyError:
(20)                                pass
(4)                 nx._clear_cache(G)
(0)             def get_edge_attributes(G, name, default=None):
(4)                 """Get edge attributes from graph
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 name : string
(7)                    Attribute name
(4)                 default: object (default=None)
(7)                    Default value of the edge attribute if there is no value set for that
(7)                    edge in graph. If `None` then edges without this attribute are not
(7)                    included in the returned dict.
(4)                 Returns
(4)                 -------
(4)                 Dictionary of attributes keyed by edge. For (di)graphs, the keys are
(4)                 2-tuples of the form: (u, v). For multi(di)graphs, the keys are 3-tuples
of
(4)                 the form: (u, v, key).
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> nx.add_path(G, [1, 2, 3], color="red")
(4)                 >>> color = nx.get_edge_attributes(G, "color")
(4)                 >>> color[(1, 2)]
(4)                 'red'
(4)                 >>> G.add_edge(3, 4)
(4)                 >>> color = nx.get_edge_attributes(G, "color", default="yellow")
(4)                 >>> color[(3, 4)]
(4)                 'yellow'
(4)                 """
(4)                 if G.is_multigraph():
(8)                     edges = G.edges(keys=True, data=True)
(4)                 else:
```

```
(8)                 edges = G.edges(data=True)
(4)             if default is not None:
(8)                 return {x[:-1]: x[-1].get(name, default) for x in edges}
(4)             return {x[:-1]: x[-1][name] for x in edges if name in x[-1]}
(0)         def all_neighbors(graph, node):
(4)             """Returns all of the neighbors of a node in the graph.
(4)             If the graph is directed returns predecessors as well as successors.
(4)             Parameters
(4)             ----------
(4)             graph : NetworkX graph
(8)                 Graph to find neighbors.
(4)             node : node
(8)                 The node whose neighbors will be returned.
(4)             Returns
(4)             -------
(4)             neighbors : iterator
(8)                 Iterator of neighbors
(4)             """
(4)             if graph.is_directed():
(8)                 values = chain(graph.predecessors(node), graph.successors(node))
(4)             else:
(8)                 values = graph.neighbors(node)
(4)             return values
(0)         def non_neighbors(graph, node):
(4)             """Returns the non-neighbors of the node in the graph.
(4)             Parameters
(4)             ----------
(4)             graph : NetworkX graph
(8)                 Graph to find neighbors.
(4)             node : node
(8)                 The node whose neighbors will be returned.
(4)             Returns
(4)             -------
(4)             non_neighbors : set
(8)                 Set of nodes in the graph that are not neighbors of the node.
(4)             """
(4)             return graph._adj.keys() - graph._adj[node].keys() - {node}
(0)         def non_edges(graph):
(4)             """Returns the nonexistent edges in the graph.
(4)             Parameters
(4)             ----------
(4)             graph : NetworkX graph.
(8)                 Graph to find nonexistent edges.
(4)             Returns
(4)             -------
(4)             non_edges : iterator
(8)                 Iterator of edges that are not in the graph.
(4)             """
(4)             if graph.is_directed():
(8)                 for u in graph:
(12)                    for v in non_neighbors(graph, u):
(16)                        yield (u, v)
(4)             else:
(8)                 nodes = set(graph)
(8)                 while nodes:
(12)                    u = nodes.pop()
(12)                    for v in nodes - set(graph[u]):
(16)                        yield (u, v)
(0)         @not_implemented_for("directed")
(0)         def common_neighbors(G, u, v):
(4)             """Returns the common neighbors of two nodes in a graph.
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A NetworkX undirected graph.
(4)             u, v : nodes
(8)                 Nodes in the graph.
(4)             Returns
(4)             -------
```

```
(4)                 cnbors : set
(8)                     Set of common neighbors of u and v in the graph.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If u or v is not a node in the graph.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.complete_graph(5)
(4)                 >>> sorted(nx.common_neighbors(G, 0, 1))
(4)                 [2, 3, 4]
(4)                 """
(4)                 if u not in G:
(8)                     raise nx.NetworkXError("u is not in the graph.")
(4)                 if v not in G:
(8)                     raise nx.NetworkXError("v is not in the graph.")
(4)                 return G._adj[u].keys() & G._adj[v].keys() - {u, v}
(0)             def is_weighted(G, edge=None, weight="weight"):
(4)                 """Returns True if `G` has weighted edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX graph.
(4)                 edge : tuple, optional
(8)                     A 2-tuple specifying the only edge in `G` that will be tested. If
(8)                     None, then every edge in `G` is tested.
(4)                 weight: string, optional
(8)                     The attribute name used to query for edge weights.
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     A boolean signifying if `G`, or the specified edge, is weighted.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If the specified edge does not exist.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> nx.is_weighted(G)
(4)                 False
(4)                 >>> nx.is_weighted(G, (2, 3))
(4)                 False
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_edge(1, 2, weight=1)
(4)                 >>> nx.is_weighted(G)
(4)                 True
(4)                 """
(4)                 if edge is not None:
(8)                     data = G.get_edge_data(*edge)
(8)                     if data is None:
(12)                        msg = f"Edge {edge!r} does not exist."
(12)                        raise nx.NetworkXError(msg)
(8)                     return weight in data
(4)                 if is_empty(G):
(8)                     return False
(4)                 return all(weight in data for u, v, data in G.edges(data=True))
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def is_negatively_weighted(G, edge=None, weight="weight"):
(4)                 """Returns True if `G` has negatively weighted edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A NetworkX graph.
(4)                 edge : tuple, optional
(8)                     A 2-tuple specifying the only edge in `G` that will be tested. If
(8)                     None, then every edge in `G` is tested.
(4)                 weight: string, optional
(8)                     The attribute name used to query for edge weights.
```

```
847: (4)                Returns
848: (4)                -------
849: (4)                bool
850: (8)                    A boolean signifying if `G`, or the specified edge, is negatively
851: (8)                    weighted.
852: (4)                Raises
853: (4)                ------
854: (4)                NetworkXError
855: (8)                    If the specified edge does not exist.
856: (4)                Examples
857: (4)                --------
858: (4)                >>> G = nx.Graph()
859: (4)                >>> G.add_edges_from([(1, 3), (2, 4), (2, 6)])
860: (4)                >>> G.add_edge(1, 2, weight=4)
861: (4)                >>> nx.is_negatively_weighted(G, (1, 2))
862: (4)                False
863: (4)                >>> G[2][4]["weight"] = -2
864: (4)                >>> nx.is_negatively_weighted(G)
865: (4)                True
866: (4)                >>> G = nx.DiGraph()
867: (4)                >>> edges = [("0", "3", 3), ("0", "1", -5), ("1", "0", -2)]
868: (4)                >>> G.add_weighted_edges_from(edges)
869: (4)                >>> nx.is_negatively_weighted(G)
870: (4)                True
871: (4)                """
872: (4)                if edge is not None:
873: (8)                    data = G.get_edge_data(*edge)
874: (8)                    if data is None:
875: (12)                       msg = f"Edge {edge!r} does not exist."
876: (12)                       raise nx.NetworkXError(msg)
877: (8)                    return weight in data and data[weight] < 0
878: (4)                return any(weight in data and data[weight] < 0 for u, v, data in
G.edges(data=True))
879: (0)            def is_empty(G):
880: (4)                """Returns True if `G` has no edges.
881: (4)                Parameters
882: (4)                ----------
883: (4)                G : graph
884: (8)                    A NetworkX graph.
885: (4)                Returns
886: (4)                -------
887: (4)                bool
888: (8)                    True if `G` has no edges, and False otherwise.
889: (4)                Notes
890: (4)                -----
891: (4)                An empty graph can have nodes but not edges. The empty graph with zero
892: (4)                nodes is known as the null graph. This is an $O(n)$ operation where n
893: (4)                is the number of nodes in the graph.
894: (4)                """
895: (4)                return not any(G._adj.values())
896: (0)            def nodes_with_selfloops(G):
897: (4)                """Returns an iterator over nodes with self loops.
898: (4)                A node with a self loop has an edge with both ends adjacent
899: (4)                to that node.
900: (4)                Returns
901: (4)                -------
902: (4)                nodelist : iterator
903: (8)                    A iterator over nodes with self loops.
904: (4)                See Also
905: (4)                --------
906: (4)                selfloop_edges, number_of_selfloops
907: (4)                Examples
908: (4)                --------
909: (4)                >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
910: (4)                >>> G.add_edge(1, 1)
911: (4)                >>> G.add_edge(1, 2)
912: (4)                >>> list(nx.nodes_with_selfloops(G))
913: (4)                [1]
914: (4)                """
```

```
915: (4)                return (n for n, nbrs in G._adj.items() if n in nbrs)
916: (0)            def selfloop_edges(G, data=False, keys=False, default=None):
917: (4)                """Returns an iterator over selfloop edges.
918: (4)                A selfloop edge has the same node at both ends.
919: (4)                Parameters
920: (4)                ----------
921: (4)                G : graph
922: (8)                    A NetworkX graph.
923: (4)                data : string or bool, optional (default=False)
924: (8)                    Return selfloop edges as two tuples (u, v) (data=False)
925: (8)                    or three-tuples (u, v, datadict) (data=True)
926: (8)                    or three-tuples (u, v, datavalue) (data='attrname')
927: (4)                keys : bool, optional (default=False)
928: (8)                    If True, return edge keys with each edge.
929: (4)                default : value, optional (default=None)
930: (8)                    Value used for edges that don't have the requested attribute.
931: (8)                    Only relevant if data is not True or False.
932: (4)                Returns
933: (4)                -------
934: (4)                edgeiter : iterator over edge tuples
935: (8)                    An iterator over all selfloop edges.
936: (4)                See Also
937: (4)                --------
938: (4)                nodes_with_selfloops, number_of_selfloops
939: (4)                Examples
940: (4)                --------
941: (4)                >>> G = nx.MultiGraph()  # or Graph, DiGraph, MultiDiGraph, etc
942: (4)                >>> ekey = G.add_edge(1, 1)
943: (4)                >>> ekey = G.add_edge(1, 2)
944: (4)                >>> list(nx.selfloop_edges(G))
945: (4)                [(1, 1)]
946: (4)                >>> list(nx.selfloop_edges(G, data=True))
947: (4)                [(1, 1, {})]
948: (4)                >>> list(nx.selfloop_edges(G, keys=True))
949: (4)                [(1, 1, 0)]
950: (4)                >>> list(nx.selfloop_edges(G, keys=True, data=True))
951: (4)                [(1, 1, 0, {})]
952: (4)                """
953: (4)                if data is True:
954: (8)                    if G.is_multigraph():
955: (12)                       if keys is True:
956: (16)                           return (
957: (20)                               (n, n, k, d)
958: (20)                               for n, nbrs in G._adj.items()
959: (20)                               if n in nbrs
960: (20)                               for k, d in nbrs[n].items()
961: (16)                           )
962: (12)                       else:
963: (16)                           return (
964: (20)                               (n, n, d)
965: (20)                               for n, nbrs in G._adj.items()
966: (20)                               if n in nbrs
967: (20)                               for d in nbrs[n].values()
968: (16)                           )
969: (8)                    else:
970: (12)                       return ((n, n, nbrs[n]) for n, nbrs in G._adj.items() if n in
nbrs)
971: (4)                elif data is not False:
972: (8)                    if G.is_multigraph():
973: (12)                       if keys is True:
974: (16)                           return (
975: (20)                               (n, n, k, d.get(data, default))
976: (20)                               for n, nbrs in G._adj.items()
977: (20)                               if n in nbrs
978: (20)                               for k, d in nbrs[n].items()
979: (16)                           )
980: (12)                       else:
981: (16)                           return (
982: (20)                               (n, n, d.get(data, default))
```

```
983: (20)                                for n, nbrs in G._adj.items()
984: (20)                                if n in nbrs
985: (20)                                for d in nbrs[n].values()
986: (16)                            )
987: (8)                    else:
988: (12)                       return (
989: (16)                           (n, n, nbrs[n].get(data, default))
990: (16)                           for n, nbrs in G._adj.items()
991: (16)                           if n in nbrs
992: (12)                       )
993: (4)                else:
994: (8)                    if G.is_multigraph():
995: (12)                       if keys is True:
996: (16)                           return (
997: (20)                               (n, n, k)
998: (20)                               for n, nbrs in G._adj.items()
999: (20)                               if n in nbrs
1000: (20)                              for k in nbrs[n]
1001: (16)                          )
1002: (12)                      else:
1003: (16)                          return (
1004: (20)                              (n, n)
1005: (20)                              for n, nbrs in G._adj.items()
1006: (20)                              if n in nbrs
1007: (20)                              for i in range(len(nbrs[n]))  # for easy edge removal
(#4068)
1008: (16)                          )
1009: (8)                   else:
1010: (12)                      return ((n, n) for n, nbrs in G._adj.items() if n in nbrs)
1011: (0)           def number_of_selfloops(G):
1012: (4)               """Returns the number of selfloop edges.
1013: (4)               A selfloop edge has the same node at both ends.
1014: (4)               Returns
1015: (4)               -------
1016: (4)               nloops : int
1017: (8)                   The number of selfloops.
1018: (4)               See Also
1019: (4)               --------
1020: (4)               nodes_with_selfloops, selfloop_edges
1021: (4)               Examples
1022: (4)               --------
1023: (4)               >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
1024: (4)               >>> G.add_edge(1, 1)
1025: (4)               >>> G.add_edge(1, 2)
1026: (4)               >>> nx.number_of_selfloops(G)
1027: (4)               1
1028: (4)               """
1029: (4)               return sum(1 for _ in nx.selfloop_edges(G))
1030: (0)           def is_path(G, path):
1031: (4)               """Returns whether or not the specified path exists.
1032: (4)               For it to return True, every node on the path must exist and
1033: (4)               each consecutive pair must be connected via one or more edges.
1034: (4)               Parameters
1035: (4)               ----------
1036: (4)               G : graph
1037: (8)                   A NetworkX graph.
1038: (4)               path : list
1039: (8)                   A list of nodes which defines the path to traverse
1040: (4)               Returns
1041: (4)               -------
1042: (4)               bool
1043: (8)                   True if `path` is a valid path in `G`
1044: (4)               """
1045: (4)               try:
1046: (8)                   return all(nbr in G._adj[node] for node, nbr in
nx.utils.pairwise(path))
1047: (4)               except (KeyError, TypeError):
1048: (8)                   return False
1049: (0)           def path_weight(G, path, weight):
```

```
1050: (4)               """Returns total cost associated with specified path and weight
1051: (4)               Parameters
1052: (4)               ----------
1053: (4)               G : graph
1054: (8)                   A NetworkX graph.
1055: (4)               path: list
1056: (8)                   A list of node labels which defines the path to traverse
1057: (4)               weight: string
1058: (8)                   A string indicating which edge attribute to use for path cost
1059: (4)               Returns
1060: (4)               -------
1061: (4)               cost: int or float
1062: (8)                   An integer or a float representing the total cost with respect to the
1063: (8)                   specified weight of the specified path
1064: (4)               Raises
1065: (4)               ------
1066: (4)               NetworkXNoPath
1067: (8)                   If the specified edge does not exist.
1068: (4)               """
1069: (4)               multigraph = G.is_multigraph()
1070: (4)               cost = 0
1071: (4)               if not nx.is_path(G, path):
1072: (8)                   raise nx.NetworkXNoPath("path does not exist")
1073: (4)               for node, nbr in nx.utils.pairwise(path):
1074: (8)                   if multigraph:
1075: (12)                      cost += min(v[weight] for v in G._adj[node][nbr].values())
1076: (8)                   else:
1077: (12)                      cost += G._adj[node][nbr][weight]
1078: (4)               return cost
```

File 393 - graph.py:

```
1: (0)              """Base class for undirected graphs.
2: (0)              The Graph class allows any hashable object as a node
3: (0)              and can associate key/value attribute pairs with each undirected edge.
4: (0)              Self-loops are allowed but multiple edges are not (see MultiGraph).
5: (0)              For directed graphs see DiGraph and MultiDiGraph.
6: (0)              """
7: (0)              from copy import deepcopy
8: (0)              from functools import cached_property
9: (0)              import networkx as nx
10: (0)             from networkx import convert
11: (0)             from networkx.classes.coreviews import AdjacencyView
12: (0)             from networkx.classes.reportviews import DegreeView, EdgeView, NodeView
13: (0)             from networkx.exception import NetworkXError
14: (0)             __all__ = ["Graph"]
15: (0)             class _CachedPropertyResetterAdj:
16: (4)                 """Data Descriptor class for _adj that resets ``adj`` cached_property when
needed
17: (4)                 This assumes that the ``cached_property`` ``G.adj`` should be reset
whenever
18: (4)                 ``G._adj`` is set to a new value.
19: (4)                 This object sits on a class and ensures that any instance of that
20: (4)                 class clears its cached property "adj" whenever the underlying
21: (4)                 instance attribute "_adj" is set to a new object. It only affects
22: (4)                 the set process of the obj._adj attribute. All get/del operations
23: (4)                 act as they normally would.
24: (4)                 For info on Data Descriptors see:
https://docs.python.org/3/howto/descriptor.html
25: (4)                 """
26: (4)                 def __set__(self, obj, value):
27: (8)                     od = obj.__dict__
28: (8)                     od["_adj"] = value
29: (8)                     if "adj" in od:
30: (12)                        del od["adj"]
31: (0)             class _CachedPropertyResetterNode:
32: (4)                 """Data Descriptor class for _node that resets ``nodes`` cached_property
```

```
when needed
33: (4)                 This assumes that the ``cached_property`` ``G.node`` should be reset
whenever
34: (4)                 ``G._node`` is set to a new value.
35: (4)                 This object sits on a class and ensures that any instance of that
36: (4)                 class clears its cached property "nodes" whenever the underlying
37: (4)                 instance attribute "_node" is set to a new object. It only affects
38: (4)                 the set process of the obj._adj attribute. All get/del operations
39: (4)                 act as they normally would.
40: (4)                 For info on Data Descriptors see:
https://docs.python.org/3/howto/descriptor.html
41: (4)                 """
42: (4)                 def __set__(self, obj, value):
43: (8)                     od = obj.__dict__
44: (8)                     od["_node"] = value
45: (8)                     if "nodes" in od:
46: (12)                        del od["nodes"]
47: (0)             class Graph:
48: (4)                 """
49: (4)                 Base class for undirected graphs.
50: (4)                 A Graph stores nodes and edges with optional data, or attributes.
51: (4)                 Graphs hold undirected edges.  Self loops are allowed but multiple
52: (4)                 (parallel) edges are not.
53: (4)                 Nodes can be arbitrary (hashable) Python objects with optional
54: (4)                 key/value attributes, except that `None` is not allowed as a node.
55: (4)                 Edges are represented as links between nodes with optional
56: (4)                 key/value attributes.
57: (4)                 Parameters
58: (4)                 ----------
59: (4)                 incoming_graph_data : input graph (optional, default: None)
60: (8)                     Data to initialize graph. If None (default) an empty
61: (8)                     graph is created.  The data can be any format that is supported
62: (8)                     by the to_networkx_graph() function, currently including edge list,
63: (8)                     dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy
64: (8)                     sparse matrix, or PyGraphviz graph.
65: (4)                 attr : keyword arguments, optional (default= no attributes)
66: (8)                     Attributes to add to graph as key=value pairs.
67: (4)                 See Also
68: (4)                 --------
69: (4)                 DiGraph
70: (4)                 MultiGraph
71: (4)                 MultiDiGraph
72: (4)                 Examples
73: (4)                 --------
74: (4)                 Create an empty graph structure (a "null graph") with no nodes and
75: (4)                 no edges.
76: (4)                 >>> G = nx.Graph()
77: (4)                 G can be grown in several ways.
78: (4)                 **Nodes:**
79: (4)                 Add one node at a time:
80: (4)                 >>> G.add_node(1)
81: (4)                 Add the nodes from any container (a list, dict, set or
82: (4)                 even the lines from a file or the nodes from another graph).
83: (4)                 >>> G.add_nodes_from([2, 3])
84: (4)                 >>> G.add_nodes_from(range(100, 110))
85: (4)                 >>> H = nx.path_graph(10)
86: (4)                 >>> G.add_nodes_from(H)
87: (4)                 In addition to strings and integers any hashable Python object
88: (4)                 (except None) can represent a node, e.g. a customized node object,
89: (4)                 or even another Graph.
90: (4)                 >>> G.add_node(H)
91: (4)                 **Edges:**
92: (4)                 G can also be grown by adding edges.
93: (4)                 Add one edge,
94: (4)                 >>> G.add_edge(1, 2)
95: (4)                 a list of edges,
96: (4)                 >>> G.add_edges_from([(1, 2), (1, 3)])
97: (4)                 or a collection of edges,
98: (4)                 >>> G.add_edges_from(H.edges)
```

```
99: (4)                 If some edges connect nodes not yet in the graph, the nodes
100: (4)                are added automatically.  There are no errors when adding
101: (4)                nodes or edges that already exist.
102: (4)                **Attributes:**
103: (4)                Each graph, node, and edge can hold key/value attribute pairs
104: (4)                in an associated attribute dictionary (the keys must be hashable).
105: (4)                By default these are empty, but can be added or changed using
106: (4)                add_edge, add_node or direct manipulation of the attribute
107: (4)                dictionaries named graph, node and edge respectively.
108: (4)                >>> G = nx.Graph(day="Friday")
109: (4)                >>> G.graph
110: (4)                {'day': 'Friday'}
111: (4)                Add node attributes using add_node(), add_nodes_from() or G.nodes
112: (4)                >>> G.add_node(1, time="5pm")
113: (4)                >>> G.add_nodes_from([3], time="2pm")
114: (4)                >>> G.nodes[1]
115: (4)                {'time': '5pm'}
116: (4)                >>> G.nodes[1]["room"] = 714  # node must exist already to use G.nodes
117: (4)                >>> del G.nodes[1]["room"]  # remove attribute
118: (4)                >>> list(G.nodes(data=True))
119: (4)                [(1, {'time': '5pm'}), (3, {'time': '2pm'})]
120: (4)                Add edge attributes using add_edge(), add_edges_from(), subscript
121: (4)                notation, or G.edges.
122: (4)                >>> G.add_edge(1, 2, weight=4.7)
123: (4)                >>> G.add_edges_from([(3, 4), (4, 5)], color="red")
124: (4)                >>> G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
125: (4)                >>> G[1][2]["weight"] = 4.7
126: (4)                >>> G.edges[1, 2]["weight"] = 4
127: (4)                Warning: we protect the graph data structure by making `G.edges` a
128: (4)                read-only dict-like structure. However, you can assign to attributes
129: (4)                in e.g. `G.edges[1, 2]`. Thus, use 2 sets of brackets to add/change
130: (4)                data attributes: `G.edges[1, 2]['weight'] = 4`
131: (4)                (For multigraphs: `MG.edges[u, v, key][name] = value`).
132: (4)                **Shortcuts:**
133: (4)                Many common graph features allow python syntax to speed reporting.
134: (4)                >>> 1 in G  # check if node in graph
135: (4)                True
136: (4)                >>> [n for n in G if n < 3]  # iterate through nodes
137: (4)                [1, 2]
138: (4)                >>> len(G)  # number of nodes in graph
139: (4)                5
140: (4)                Often the best way to traverse all edges of a graph is via the neighbors.
141: (4)                The neighbors are reported as an adjacency-dict `G.adj` or `G.adjacency()`
142: (4)                >>> for n, nbrsdict in G.adjacency():
143: (4)                ...     for nbr, eattr in nbrsdict.items():
144: (4)                ...         if "weight" in eattr:
145: (4)                ...             # Do something useful with the edges
146: (4)                ...             pass
147: (4)                But the edges() method is often more convenient:
148: (4)                >>> for u, v, weight in G.edges.data("weight"):
149: (4)                ...     if weight is not None:
150: (4)                ...         # Do something useful with the edges
151: (4)                ...         pass
152: (4)                **Reporting:**
153: (4)                Simple graph information is obtained using object-attributes and methods.
154: (4)                Reporting typically provides views instead of containers to reduce memory
155: (4)                usage. The views update as the graph is updated similarly to dict-views.
156: (4)                The objects `nodes`, `edges` and `adj` provide access to data attributes
157: (4)                via lookup (e.g. `nodes[n]`, `edges[u, v]`, `adj[u][v]`) and iteration
158: (4)                (e.g. `nodes.items()`, `nodes.data('color')`,
159: (4)                `nodes.data('color', default='blue')` and similarly for `edges`)
160: (4)                Views exist for `nodes`, `edges`, `neighbors()`/`adj` and `degree`.
161: (4)                For details on these and other miscellaneous methods, see below.
162: (4)                **Subclasses (Advanced):**
163: (4)                The Graph class uses a dict-of-dict-of-dict data structure.
164: (4)                The outer dict (node_dict) holds adjacency information keyed by node.
165: (4)                The next dict (adjlist_dict) represents the adjacency information and
holds
166: (4)                edge data keyed by neighbor.  The inner dict (edge_attr_dict) represents
```

```
    the edge data and holds edge attribute values keyed by attribute names.
    Each of these three dicts can be replaced in a subclass by a user defined
    dict-like object. In general, the dict-like features should be
    maintained but extra features can be added. To replace one of the
    dicts create a new graph class by changing the class(!) variable
    holding the factory for that dict-like structure.
    node_dict_factory : function, (default: dict)
        Factory function to be used to create the dict containing node
        attributes, keyed by node id.
        It should require no arguments and return a dict-like object
    node_attr_dict_factory: function, (default: dict)
        Factory function to be used to create the node attribute
        dict which holds attribute values keyed by attribute name.
        It should require no arguments and return a dict-like object
    adjlist_outer_dict_factory : function, (default: dict)
        Factory function to be used to create the outer-most dict
        in the data structure that holds adjacency info keyed by node.
        It should require no arguments and return a dict-like object.
    adjlist_inner_dict_factory : function, (default: dict)
        Factory function to be used to create the adjacency list
        dict which holds edge data keyed by neighbor.
        It should require no arguments and return a dict-like object
    edge_attr_dict_factory : function, (default: dict)
        Factory function to be used to create the edge attribute
        dict which holds attribute values keyed by attribute name.
        It should require no arguments and return a dict-like object.
    graph_attr_dict_factory : function, (default: dict)
        Factory function to be used to create the graph attribute
        dict which holds attribute values keyed by attribute name.
        It should require no arguments and return a dict-like object.
    Typically, if your extension doesn't impact the data structure all
    methods will inherit without issue except: `to_directed/to_undirected`.
    By default these methods create a DiGraph/Graph class and you probably
    want them to create your extension of a DiGraph/Graph. To facilitate
    this we define two class variables that you can set in your subclass.
    to_directed_class : callable, (default: DiGraph or MultiDiGraph)
        Class to create a new graph structure in the `to_directed` method.
        If `None`, a NetworkX class (DiGraph or MultiDiGraph) is used.
    to_undirected_class : callable, (default: Graph or MultiGraph)
        Class to create a new graph structure in the `to_undirected` method.
        If `None`, a NetworkX class (Graph or MultiGraph) is used.
    **Subclassing Example**
    Create a low memory graph class that effectively disallows edge
    attributes by using a single attribute dict for all edges.
    This reduces the memory used, but you lose edge attributes.
    >>> class ThinGraph(nx.Graph):
    ...     all_edge_dict = {"weight": 1}
    ...
    ...     def single_edge_dict(self):
    ...         return self.all_edge_dict
    ...
    ...     edge_attr_dict_factory = single_edge_dict
    >>> G = ThinGraph()
    >>> G.add_edge(2, 1)
    >>> G[2][1]
    {'weight': 1}
    >>> G.add_edge(2, 2)
    >>> G[2][1] is G[2][2]
    True
    """
    _adj = _CachedPropertyResetterAdj()
    _node = _CachedPropertyResetterNode()
    node_dict_factory = dict
    node_attr_dict_factory = dict
    adjlist_outer_dict_factory = dict
    adjlist_inner_dict_factory = dict
    edge_attr_dict_factory = dict
    graph_attr_dict_factory = dict
    def to_directed_class(self):
```

```
236: (8)                    """Returns the class to use for empty directed copies.
237: (8)                    If you subclass the base classes, use this to designate
238: (8)                    what directed class to use for `to_directed()` copies.
239: (8)                    """
240: (8)                    return nx.DiGraph
241: (4)                def to_undirected_class(self):
242: (8)                    """Returns the class to use for empty undirected copies.
243: (8)                    If you subclass the base classes, use this to designate
244: (8)                    what directed class to use for `to_directed()` copies.
245: (8)                    """
246: (8)                    return Graph
247: (4)                def __init__(self, incoming_graph_data=None, **attr):
248: (8)                    """Initialize a graph with edges, name, or graph attributes.
249: (8)                    Parameters
250: (8)                    ----------
251: (8)                    incoming_graph_data : input graph (optional, default: None)
252: (12)                       Data to initialize graph. If None (default) an empty
253: (12)                       graph is created.  The data can be an edge list, or any
254: (12)                       NetworkX graph object.  If the corresponding optional Python
255: (12)                       packages are installed the data can also be a 2D NumPy array, a
256: (12)                       SciPy sparse array, or a PyGraphviz graph.
257: (8)                    attr : keyword arguments, optional (default= no attributes)
258: (12)                       Attributes to add to graph as key=value pairs.
259: (8)                    See Also
260: (8)                    --------
261: (8)                    convert
262: (8)                    Examples
263: (8)                    --------
264: (8)                    >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
265: (8)                    >>> G = nx.Graph(name="my graph")
266: (8)                    >>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
267: (8)                    >>> G = nx.Graph(e)
268: (8)                    Arbitrary graph attribute pairs (key=value) may be assigned
269: (8)                    >>> G = nx.Graph(e, day="Friday")
270: (8)                    >>> G.graph
271: (8)                    {'day': 'Friday'}
272: (8)                    """
273: (8)                    self.graph = self.graph_attr_dict_factory()  # dictionary for graph
attributes
274: (8)                    self._node = self.node_dict_factory()  # empty node attribute dict
275: (8)                    self._adj = self.adjlist_outer_dict_factory()  # empty adjacency dict
276: (8)                    self.__networkx_cache__ = {}
277: (8)                    if incoming_graph_data is not None:
278: (12)                       convert.to_networkx_graph(incoming_graph_data, create_using=self)
279: (8)                    self.graph.update(attr)
280: (4)                @cached_property
281: (4)                def adj(self):
282: (8)                    """Graph adjacency object holding the neighbors of each node.
283: (8)                    This object is a read-only dict-like structure with node keys
284: (8)                    and neighbor-dict values.  The neighbor-dict is keyed by neighbor
285: (8)                    to the edge-data-dict.  So `G.adj[3][2]['color'] = 'blue'` sets
286: (8)                    the color of the edge `(3, 2)` to `"blue"`.
287: (8)                    Iterating over G.adj behaves like a dict. Useful idioms include
288: (8)                    `for nbr, datadict in G.adj[n].items():`.
289: (8)                    The neighbor information is also provided by subscripting the graph.
290: (8)                    So `for nbr, foovalue in G[node].data('foo', default=1):` works.
291: (8)                    For directed graphs, `G.adj` holds outgoing (successor) info.
292: (8)                    """
293: (8)                    return AdjacencyView(self._adj)
294: (4)                @property
295: (4)                def name(self):
296: (8)                    """String identifier of the graph.
297: (8)                    This graph attribute appears in the attribute dict G.graph
298: (8)                    keyed by the string `"name"`. as well as an attribute (technically
299: (8)                    a property) `G.name`. This is entirely user controlled.
300: (8)                    """
301: (8)                    return self.graph.get("name", "")
302: (4)                @name.setter
303: (4)                def name(self, s):
```

```
304: (8)                    self.graph["name"] = s
305: (8)                    nx._clear_cache(self)
306: (4)                def __str__(self):
307: (8)                    """Returns a short summary of the graph.
308: (8)                    Returns
309: (8)                    -------
310: (8)                    info : string
311: (12)                       Graph information including the graph name (if any), graph type,
and the
312: (12)                       number of nodes and edges.
313: (8)                    Examples
314: (8)                    --------
315: (8)                    >>> G = nx.Graph(name="foo")
316: (8)                    >>> str(G)
317: (8)                    "Graph named 'foo' with 0 nodes and 0 edges"
318: (8)                    >>> G = nx.path_graph(3)
319: (8)                    >>> str(G)
320: (8)                    'Graph with 3 nodes and 2 edges'
321: (8)                    """
322: (8)                    return "".join(
323: (12)                       [
324: (16)                           type(self).__name__,
325: (16)                           f" named {self.name!r}" if self.name else "",
326: (16)                           f" with {self.number_of_nodes()} nodes and
{self.number_of_edges()} edges",
327: (12)                       ]
328: (8)                    )
329: (4)                def __iter__(self):
330: (8)                    """Iterate over the nodes. Use: 'for n in G'.
331: (8)                    Returns
332: (8)                    -------
333: (8)                    niter : iterator
334: (12)                       An iterator over all nodes in the graph.
335: (8)                    Examples
336: (8)                    --------
337: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
338: (8)                    >>> [n for n in G]
339: (8)                    [0, 1, 2, 3]
340: (8)                    >>> list(G)
341: (8)                    [0, 1, 2, 3]
342: (8)                    """
343: (8)                    return iter(self._node)
344: (4)                def __contains__(self, n):
345: (8)                    """Returns True if n is a node, False otherwise. Use: 'n in G'.
346: (8)                    Examples
347: (8)                    --------
348: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
349: (8)                    >>> 1 in G
350: (8)                    True
351: (8)                    """
352: (8)                    try:
353: (12)                       return n in self._node
354: (8)                    except TypeError:
355: (12)                       return False
356: (4)                def __len__(self):
357: (8)                    """Returns the number of nodes in the graph. Use: 'len(G)'.
358: (8)                    Returns
359: (8)                    -------
360: (8)                    nnodes : int
361: (12)                       The number of nodes in the graph.
362: (8)                    See Also
363: (8)                    --------
364: (8)                    number_of_nodes: identical method
365: (8)                    order: identical method
366: (8)                    Examples
367: (8)                    --------
368: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
369: (8)                    >>> len(G)
370: (8)                    4
```

```
(8)                 """
(8)                 return len(self._node)
(4)             def __getitem__(self, n):
(8)                 """Returns a dict of neighbors of node n.  Use: 'G[n]'.
(8)                 Parameters
(8)                 ----------
(8)                 n : node
(11)                   A node in the graph.
(8)                 Returns
(8)                 -------
(8)                 adj_dict : dictionary
(11)                   The adjacency dictionary for nodes connected to n.
(8)                 Notes
(8)                 -----
(8)                 G[n] is the same as G.adj[n] and similar to G.neighbors(n)
(8)                 (which is an iterator over G.adj[n])
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                 >>> G[0]
(8)                 AtlasView({1: {}})
(8)                 """
(8)                 return self.adj[n]
(4)             def add_node(self, node_for_adding, **attr):
(8)                 """Add a single node `node_for_adding` and update node attributes.
(8)                 Parameters
(8)                 ----------
(8)                 node_for_adding : node
(12)                    A node can be any hashable Python object except None.
(8)                 attr : keyword arguments, optional
(12)                    Set or change node attributes using key=value.
(8)                 See Also
(8)                 --------
(8)                 add_nodes_from
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                 >>> G.add_node(1)
(8)                 >>> G.add_node("Hello")
(8)                 >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
(8)                 >>> G.add_node(K3)
(8)                 >>> G.number_of_nodes()
(8)                 3
(8)                 Use keywords set/change node attributes:
(8)                 >>> G.add_node(1, size=10)
(8)                 >>> G.add_node(3, weight=0.4, UTM=("13S", 382871, 3972649))
(8)                 Notes
(8)                 -----
(8)                 A hashable object is one that can be used as a key in a Python
(8)                 dictionary. This includes strings, numbers, tuples of strings
(8)                 and numbers, etc.
(8)                 On many platforms hashable items also include mutables such as
(8)                 NetworkX Graphs, though one should be careful that the hash
(8)                 doesn't change on mutables.
(8)                 """
(8)                 if node_for_adding not in self._node:
(12)                    if node_for_adding is None:
(16)                        raise ValueError("None cannot be a node")
(12)                    self._adj[node_for_adding] = self.adjlist_inner_dict_factory()
(12)                    attr_dict = self._node[node_for_adding] =
self.node_attr_dict_factory()
(12)                    attr_dict.update(attr)
(8)                 else:  # update attr even if node already exists
(12)                    self._node[node_for_adding].update(attr)
(8)                 nx._clear_cache(self)
(4)             def add_nodes_from(self, nodes_for_adding, **attr):
(8)                 """Add multiple nodes.
(8)                 Parameters
(8)                 ----------
```

```
        nodes_for_adding : iterable container
            A container of nodes (list, dict, set, etc.).
            OR
            A container of (node, attribute dict) tuples.
            Node attributes are updated using the attribute dict.
        attr : keyword arguments, optional (default= no attributes)
            Update attributes for all nodes in nodes.
            Node attributes specified in nodes as a tuple take
            precedence over attributes specified via keyword arguments.
        See Also
        --------
        add_node
        Notes
        -----
        When adding nodes from an iterator over the graph you are changing,
        a `RuntimeError` can be raised with message:
        `RuntimeError: dictionary changed size during iteration`. This
        happens when the graph's underlying dictionary is modified during
        iteration. To avoid this error, evaluate the iterator into a separate
        object, e.g. by using `list(iterator_of_nodes)`, and pass this
        object to `G.add_nodes_from`.
        Examples
        --------
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.add_nodes_from("Hello")
        >>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
        >>> G.add_nodes_from(K3)
        >>> sorted(G.nodes(), key=str)
        [0, 1, 2, 'H', 'e', 'l', 'o']
        Use keywords to update specific node attributes for every node.
        >>> G.add_nodes_from([1, 2], size=10)
        >>> G.add_nodes_from([3, 4], weight=0.4)
        Use (node, attrdict) tuples to update attributes for specific nodes.
        >>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
        >>> G.nodes[1]["size"]
        11
        >>> H = nx.Graph()
        >>> H.add_nodes_from(G.nodes(data=True))
        >>> H.nodes[1]["size"]
        11
        Evaluate an iterator over a graph if using it to modify the same graph
        >>> G = nx.Graph([(0, 1), (1, 2), (3, 4)])
        >>> # wrong way - will raise RuntimeError
        >>> # G.add_nodes_from(n + 1 for n in G.nodes)
        >>> # correct way
        >>> G.add_nodes_from(list(n + 1 for n in G.nodes))
        """
        for n in nodes_for_adding:
            try:
                newnode = n not in self._node
                newdict = attr
            except TypeError:
                n, ndict = n
                newnode = n not in self._node
                newdict = attr.copy()
                newdict.update(ndict)
            if newnode:
                if n is None:
                    raise ValueError("None cannot be a node")
                self._adj[n] = self.adjlist_inner_dict_factory()
                self._node[n] = self.node_attr_dict_factory()
            self._node[n].update(newdict)
        nx._clear_cache(self)
    def remove_node(self, n):
        """Remove node n.
        Removes the node n and all adjacent edges.
        Attempting to remove a nonexistent node will raise an exception.
        Parameters
        ----------
```

```
508: (8)                    n : node
509: (11)                      A node in the graph
510: (8)                    Raises
511: (8)                    ------
512: (8)                    NetworkXError
513: (11)                      If n is not in the graph.
514: (8)                    See Also
515: (8)                    --------
516: (8)                    remove_nodes_from
517: (8)                    Examples
518: (8)                    --------
519: (8)                    >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
520: (8)                    >>> list(G.edges)
521: (8)                    [(0, 1), (1, 2)]
522: (8)                    >>> G.remove_node(1)
523: (8)                    >>> list(G.edges)
524: (8)                    []
525: (8)                    """
526: (8)                    adj = self._adj
527: (8)                    try:
528: (12)                       nbrs = list(adj[n])  # list handles self-loops (allows mutation)
529: (12)                       del self._node[n]
530: (8)                    except KeyError as err:  # NetworkXError if n not in self
531: (12)                       raise NetworkXError(f"The node {n} is not in the graph.") from err
532: (8)                    for u in nbrs:
533: (12)                       del adj[u][n]  # remove all edges n-u in graph
534: (8)                    del adj[n]  # now remove node
535: (8)                    nx._clear_cache(self)
536: (4)                def remove_nodes_from(self, nodes):
537: (8)                    """Remove multiple nodes.
538: (8)                    Parameters
539: (8)                    ----------
540: (8)                    nodes : iterable container
541: (12)                       A container of nodes (list, dict, set, etc.).  If a node
542: (12)                       in the container is not in the graph it is silently
543: (12)                       ignored.
544: (8)                    See Also
545: (8)                    --------
546: (8)                    remove_node
547: (8)                    Notes
548: (8)                    -----
549: (8)                    When removing nodes from an iterator over the graph you are changing,
550: (8)                    a `RuntimeError` will be raised with message:
551: (8)                    `RuntimeError: dictionary changed size during iteration`. This
552: (8)                    happens when the graph's underlying dictionary is modified during
553: (8)                    iteration. To avoid this error, evaluate the iterator into a separate
554: (8)                    object, e.g. by using `list(iterator_of_nodes)`, and pass this
555: (8)                    object to `G.remove_nodes_from`.
556: (8)                    Examples
557: (8)                    --------
558: (8)                    >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
559: (8)                    >>> e = list(G.nodes)
560: (8)                    >>> e
561: (8)                    [0, 1, 2]
562: (8)                    >>> G.remove_nodes_from(e)
563: (8)                    >>> list(G.nodes)
564: (8)                    []
565: (8)                    Evaluate an iterator over a graph if using it to modify the same graph
566: (8)                    >>> G = nx.Graph([(0, 1), (1, 2), (3, 4)])
567: (8)                    >>> # this command will fail, as the graph's dict is modified during
iteration
568: (8)                    >>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
569: (8)                    >>> # this command will work, since the dictionary underlying graph is
not modified
570: (8)                    >>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
571: (8)                    """
572: (8)                    adj = self._adj
573: (8)                    for n in nodes:
574: (12)                       try:
```

```
575: (16)                            del self._node[n]
576: (16)                            for u in list(adj[n]):  # list handles self-loops
577: (20)                                del adj[u][n]  # (allows mutation of dict in loop)
578: (16)                            del adj[n]
579: (12)                        except KeyError:
580: (16)                            pass
581: (8)                    nx._clear_cache(self)
582: (4)                @cached_property
583: (4)                def nodes(self):
584: (8)                    """A NodeView of the Graph as G.nodes or G.nodes().
585: (8)                    Can be used as `G.nodes` for data lookup and for set-like operations.
586: (8)                    Can also be used as `G.nodes(data='color', default=None)` to return a
587: (8)                    NodeDataView which reports specific node data but no set operations.
588: (8)                    It presents a dict-like interface as well with `G.nodes.items()`
589: (8)                    iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']`
590: (8)                    providing the value of the `foo` attribute for node `3`. In addition,
591: (8)                    a view `G.nodes.data('foo')` provides a dict-like interface to the
592: (8)                    `foo` attribute of each node. `G.nodes.data('foo', default=1)`
593: (8)                    provides a default for nodes that do not have attribute `foo`.
594: (8)                    Parameters
595: (8)                    ----------
596: (8)                    data : string or bool, optional (default=False)
597: (12)                       The node attribute returned in 2-tuple (n, ddict[data]).
598: (12)                       If True, return entire node attribute dict as (n, ddict).
599: (12)                       If False, return just the nodes n.
600: (8)                    default : value, optional (default=None)
601: (12)                       Value used for nodes that don't have the requested attribute.
602: (12)                       Only relevant if data is not True or False.
603: (8)                    Returns
604: (8)                    -------
605: (8)                    NodeView
606: (12)                       Allows set-like operations over the nodes as well as node
607: (12)                       attribute dict lookup and calling to get a NodeDataView.
608: (12)                       A NodeDataView iterates over `(n, data)` and has no set
operations.
609: (12)                       A NodeView iterates over `n` and includes set operations.
610: (12)                       When called, if data is False, an iterator over nodes.
611: (12)                       Otherwise an iterator of 2-tuples (node, attribute value)
612: (12)                       where the attribute is specified in `data`.
613: (12)                       If data is True then the attribute becomes the
614: (12)                       entire data dictionary.
615: (8)                    Notes
616: (8)                    -----
617: (8)                    If your node data is not needed, it is simpler and equivalent
618: (8)                    to use the expression ``for n in G``, or ``list(G)``.
619: (8)                    Examples
620: (8)                    --------
621: (8)                    There are two simple ways of getting a list of all nodes in the graph:
622: (8)                    >>> G = nx.path_graph(3)
623: (8)                    >>> list(G.nodes)
624: (8)                    [0, 1, 2]
625: (8)                    >>> list(G)
626: (8)                    [0, 1, 2]
627: (8)                    To get the node data along with the nodes:
628: (8)                    >>> G.add_node(1, time="5pm")
629: (8)                    >>> G.nodes[0]["foo"] = "bar"
630: (8)                    >>> list(G.nodes(data=True))
631: (8)                    [(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
632: (8)                    >>> list(G.nodes.data())
633: (8)                    [(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
634: (8)                    >>> list(G.nodes(data="foo"))
635: (8)                    [(0, 'bar'), (1, None), (2, None)]
636: (8)                    >>> list(G.nodes.data("foo"))
637: (8)                    [(0, 'bar'), (1, None), (2, None)]
638: (8)                    >>> list(G.nodes(data="time"))
639: (8)                    [(0, None), (1, '5pm'), (2, None)]
640: (8)                    >>> list(G.nodes.data("time"))
641: (8)                    [(0, None), (1, '5pm'), (2, None)]
642: (8)                    >>> list(G.nodes(data="time", default="Not Available"))
```

```
(8)                     [(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
(8)                     >>> list(G.nodes.data("time", default="Not Available"))
(8)                     [(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
(8)                     If some of your nodes have an attribute and the rest are assumed
(8)                     to have a default attribute value you can create a dictionary
(8)                     from node/attribute pairs using the `default` keyword argument
(8)                     to guarantee the value is never None::
(12)                        >>> G = nx.Graph()
(12)                        >>> G.add_node(0)
(12)                        >>> G.add_node(1, weight=2)
(12)                        >>> G.add_node(2, weight=3)
(12)                        >>> dict(G.nodes(data="weight", default=1))
(12)                        {0: 1, 1: 2, 2: 3}
(8)                     """
(8)                     return NodeView(self)
(4)                 def number_of_nodes(self):
(8)                     """Returns the number of nodes in the graph.
(8)                     Returns
(8)                     -------
(8)                     nnodes : int
(12)                        The number of nodes in the graph.
(8)                     See Also
(8)                     --------
(8)                     order: identical method
(8)                     __len__: identical method
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.number_of_nodes()
(8)                     3
(8)                     """
(8)                     return len(self._node)
(4)                 def order(self):
(8)                     """Returns the number of nodes in the graph.
(8)                     Returns
(8)                     -------
(8)                     nnodes : int
(12)                        The number of nodes in the graph.
(8)                     See Also
(8)                     --------
(8)                     number_of_nodes: identical method
(8)                     __len__: identical method
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.order()
(8)                     3
(8)                     """
(8)                     return len(self._node)
(4)                 def has_node(self, n):
(8)                     """Returns True if the graph contains the node n.
(8)                     Identical to `n in G`
(8)                     Parameters
(8)                     ----------
(8)                     n : node
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.has_node(0)
(8)                     True
(8)                     It is more readable and simpler to use
(8)                     >>> 0 in G
(8)                     True
(8)                     """
(8)                     try:
(12)                        return n in self._node
(8)                     except TypeError:
(12)                        return False
(4)                 def add_edge(self, u_of_edge, v_of_edge, **attr):
```

```
        """Add an edge between u and v.
        The nodes u and v will be automatically added if they are
        not already in the graph.
        Edge attributes can be specified with keywords or by directly
        accessing the edge's attribute dictionary. See examples below.
        Parameters
        ----------
        u_of_edge, v_of_edge : nodes
            Nodes can be, for example, strings or numbers.
            Nodes must be hashable (and not None) Python objects.
        attr : keyword arguments, optional
            Edge data (or labels or objects) can be assigned using
            keyword arguments.
        See Also
        --------
        add_edges_from : add a collection of edges
        Notes
        -----
        Adding an edge that already exists updates the edge data.
        Many NetworkX algorithms designed for weighted graphs use
        an edge attribute (by default `weight`) to hold a numerical value.
        Examples
        --------
        The following all add the edge e=(1, 2) to graph G:
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> e = (1, 2)
        >>> G.add_edge(1, 2)  # explicit two-node form
        >>> G.add_edge(*e)  # single edge as tuple of two nodes
        >>> G.add_edges_from([(1, 2)])  # add edges from iterable container
        Associate data to edges using keywords:
        >>> G.add_edge(1, 2, weight=3)
        >>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
        For non-string attribute keys, use subscript notation.
        >>> G.add_edge(1, 2)
        >>> G[1][2].update({0: 5})
        >>> G.edges[1, 2].update({0: 5})
        """
        u, v = u_of_edge, v_of_edge
        if u not in self._node:
            if u is None:
                raise ValueError("None cannot be a node")
            self._adj[u] = self.adjlist_inner_dict_factory()
            self._node[u] = self.node_attr_dict_factory()
        if v not in self._node:
            if v is None:
                raise ValueError("None cannot be a node")
            self._adj[v] = self.adjlist_inner_dict_factory()
            self._node[v] = self.node_attr_dict_factory()
        datadict = self._adj[u].get(v, self.edge_attr_dict_factory())
        datadict.update(attr)
        self._adj[u][v] = datadict
        self._adj[v][u] = datadict
        nx._clear_cache(self)
    def add_edges_from(self, ebunch_to_add, **attr):
        """Add all the edges in ebunch_to_add.
        Parameters
        ----------
        ebunch_to_add : container of edges
            Each edge given in the container will be added to the
            graph. The edges must be given as 2-tuples (u, v) or
            3-tuples (u, v, d) where d is a dictionary containing edge data.
        attr : keyword arguments, optional
            Edge data (or labels or objects) can be assigned using
            keyword arguments.
        See Also
        --------
        add_edge : add a single edge
        add_weighted_edges_from : convenient way to add weighted edges
        Notes
```

```
(8)                     -----
(8)                     Adding the same edge twice has no effect but any edge data
(8)                     will be updated when each duplicate edge is added.
(8)                     Edge attributes specified in an ebunch take precedence over
(8)                     attributes specified via keyword arguments.
(8)                     When adding edges from an iterator over the graph you are changing,
(8)                     a `RuntimeError` can be raised with message:
(8)                     `RuntimeError: dictionary changed size during iteration`. This
(8)                     happens when the graph's underlying dictionary is modified during
(8)                     iteration. To avoid this error, evaluate the iterator into a separate
(8)                     object, e.g. by using `list(iterator_of_edges)`, and pass this
(8)                     object to `G.add_edges_from`.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
(8)                     >>> e = zip(range(0, 3), range(1, 4))
(8)                     >>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
(8)                     Associate data to edges
(8)                     >>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
(8)                     >>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
(8)                     Evaluate an iterator over a graph if using it to modify the same graph
(8)                     >>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
(8)                     >>> # Grow graph by one new node, adding edges to all existing nodes.
(8)                     >>> # wrong way - will raise RuntimeError
(8)                     >>> # G.add_edges_from(((5, n) for n in G.nodes))
(8)                     >>> # correct way - note that there will be no self-edge for node 5
(8)                     >>> G.add_edges_from(list((5, n) for n in G.nodes))
(8)                     """
(8)                     for e in ebunch_to_add:
(12)                        ne = len(e)
(12)                        if ne == 3:
(16)                            u, v, dd = e
(12)                        elif ne == 2:
(16)                            u, v = e
(16)                            dd = {}  # doesn't need edge_attr_dict_factory
(12)                        else:
(16)                            raise NetworkXError(f"Edge tuple {e} must be a 2-tuple or 3-
tuple.")
(12)                        if u not in self._node:
(16)                            if u is None:
(20)                                raise ValueError("None cannot be a node")
(16)                            self._adj[u] = self.adjlist_inner_dict_factory()
(16)                            self._node[u] = self.node_attr_dict_factory()
(12)                        if v not in self._node:
(16)                            if v is None:
(20)                                raise ValueError("None cannot be a node")
(16)                            self._adj[v] = self.adjlist_inner_dict_factory()
(16)                            self._node[v] = self.node_attr_dict_factory()
(12)                        datadict = self._adj[u].get(v, self.edge_attr_dict_factory())
(12)                        datadict.update(attr)
(12)                        datadict.update(dd)
(12)                        self._adj[u][v] = datadict
(12)                        self._adj[v][u] = datadict
(8)                     nx._clear_cache(self)
(4)                 def add_weighted_edges_from(self, ebunch_to_add, weight="weight", **attr):
(8)                     """Add weighted edges in `ebunch_to_add` with specified weight attr
(8)                     Parameters
(8)                     ----------
(8)                     ebunch_to_add : container of edges
(12)                        Each edge given in the list or container will be added
(12)                        to the graph. The edges must be given as 3-tuples (u, v, w)
(12)                        where w is a number.
(8)                     weight : string, optional (default= 'weight')
(12)                        The attribute name for the edge weights to be added.
(8)                     attr : keyword arguments, optional (default= no attributes)
(12)                        Edge attributes to add/update for all edges.
(8)                     See Also
(8)                     --------
```

```
(8)                     add_edge : add a single edge
(8)                     add_edges_from : add multiple edges
(8)                     Notes
(8)                     -----
(8)                     Adding the same edge twice for Graph/DiGraph simply updates
(8)                     the edge data. For MultiGraph/MultiDiGraph, duplicate edges
(8)                     are stored.
(8)                     When adding edges from an iterator over the graph you are changing,
(8)                     a `RuntimeError` can be raised with message:
(8)                     `RuntimeError: dictionary changed size during iteration`. This
(8)                     happens when the graph's underlying dictionary is modified during
(8)                     iteration. To avoid this error, evaluate the iterator into a separate
(8)                     object, e.g. by using `list(iterator_of_edges)`, and pass this
(8)                     object to `G.add_weighted_edges_from`.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
(8)                     Evaluate an iterator over edges before passing it
(8)                     >>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
(8)                     >>> weight = 0.1
(8)                     >>> # Grow graph by one new node, adding edges to all existing nodes.
(8)                     >>> # wrong way - will raise RuntimeError
(8)                     >>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
(8)                     >>> # correct way - note that there will be no self-edge for node 5
(8)                     >>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
(8)                     """
(8)                     self.add_edges_from(((u, v, {weight: d}) for u, v, d in
ebunch_to_add), **attr)
(8)                     nx._clear_cache(self)
(4)                 def remove_edge(self, u, v):
(8)                     """Remove the edge between u and v.
(8)                     Parameters
(8)                     ----------
(8)                     u, v : nodes
(12)                        Remove the edge between nodes u and v.
(8)                     Raises
(8)                     ------
(8)                     NetworkXError
(12)                        If there is not an edge between u and v.
(8)                     See Also
(8)                     --------
(8)                     remove_edges_from : remove a collection of edges
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(4)  # or DiGraph, etc
(8)                     >>> G.remove_edge(0, 1)
(8)                     >>> e = (1, 2)
(8)                     >>> G.remove_edge(*e)  # unpacks e from an edge tuple
(8)                     >>> e = (2, 3, {"weight": 7})  # an edge with attribute data
(8)                     >>> G.remove_edge(*e[:2])  # select first part of edge tuple
(8)                     """
(8)                     try:
(12)                        del self._adj[u][v]
(12)                        if u != v:  # self-loop needs only one entry removed
(16)                            del self._adj[v][u]
(8)                     except KeyError as err:
(12)                        raise NetworkXError(f"The edge {u}-{v} is not in the graph") from
err
(8)                     nx._clear_cache(self)
(4)                 def remove_edges_from(self, ebunch):
(8)                     """Remove all edges specified in ebunch.
(8)                     Parameters
(8)                     ----------
(8)                     ebunch: list or container of edge tuples
(12)                        Each edge given in the list or container will be removed
(12)                        from the graph. The edges can be:
(16)                            - 2-tuples (u, v) edge between u and v.
(16)                            - 3-tuples (u, v, k) where k is ignored.
```

```
        See Also
        --------
        remove_edge : remove a single edge
        Notes
        -----
        Will fail silently if an edge in ebunch is not in the graph.
        Examples
        --------
        >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> ebunch = [(1, 2), (2, 3)]
        >>> G.remove_edges_from(ebunch)
        """
        adj = self._adj
        for e in ebunch:
            u, v = e[:2]  # ignore edge data if present
            if u in adj and v in adj[u]:
                del adj[u][v]
                if u != v:  # self loop needs only one entry removed
                    del adj[v][u]
        nx._clear_cache(self)
    def update(self, edges=None, nodes=None):
        """Update the graph using nodes/edges/graphs as input.
        Like dict.update, this method takes a graph as input, adding the
        graph's nodes and edges to this graph. It can also take two inputs:
        edges and nodes. Finally it can take either edges or nodes.
        To specify only nodes the keyword `nodes` must be used.
        The collections of edges and nodes are treated similarly to
        the add_edges_from/add_nodes_from methods. When iterated, they
        should yield 2-tuples (u, v) or 3-tuples (u, v, datadict).
        Parameters
        ----------
        edges : Graph object, collection of edges, or None
            The first parameter can be a graph or some edges. If it has
            attributes `nodes` and `edges`, then it is taken to be a
            Graph-like object and those attributes are used as collections
            of nodes and edges to be added to the graph.
            If the first parameter does not have those attributes, it is
            treated as a collection of edges and added to the graph.
            If the first argument is None, no edges are added.
        nodes : collection of nodes, or None
            The second parameter is treated as a collection of nodes
            to be added to the graph unless it is None.
            If `edges is None` and `nodes is None` an exception is raised.
            If the first parameter is a Graph, then `nodes` is ignored.
        Examples
        --------
        >>> G = nx.path_graph(5)
        >>> G.update(nx.complete_graph(range(4, 10)))
        >>> from itertools import combinations
        >>> edges = (
        ...     (u, v, {"power": u * v})
        ...     for u, v in combinations(range(10, 20), 2)
        ...     if u * v < 225
        ... )
        >>> nodes = [1000]  # for singleton, use a container
        >>> G.update(edges, nodes)
        Notes
        -----
        It you want to update the graph using an adjacency structure
        it is straightforward to obtain the edges/nodes from adjacency.
        The following examples provide common cases, your adjacency may
        be slightly different and require tweaks of these examples::
        >>> # dict-of-set/list/tuple
        >>> adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
        >>> e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
        >>> G.update(edges=e, nodes=adj)
        >>> DG = nx.DiGraph()
        >>> # dict-of-dict-of-attribute
        >>> adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
```

```
985: (8)                    >>> e = [(u, v, {"weight": d}) for u, nbrs in adj.items() for v, d in
nbrs.items()]
986: (8)                    >>> DG.update(edges=e, nodes=adj)
987: (8)                    >>> # dict-of-dict-of-dict
988: (8)                    >>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
989: (8)                    >>> e = [(u, v, {"weight": d}) for u, nbrs in adj.items() for v, d in
nbrs.items()]
990: (8)                    >>> DG.update(edges=e, nodes=adj)
991: (8)                    >>> # predecessor adjacency (dict-of-set)
992: (8)                    >>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
993: (8)                    >>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
994: (8)                    >>> # MultiGraph dict-of-dict-of-dict-of-attribute
995: (8)                    >>> MDG = nx.MultiDiGraph()
996: (8)                    >>> adj = {
997: (8)                    ...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
998: (8)                    ...     3: {2: {0: {"weight": 0.7}}},
999: (8)                    ... }
1000: (8)                   >>> e = [
1001: (8)                   ...     (u, v, ekey, d)
1002: (8)                   ...     for u, nbrs in adj.items()
1003: (8)                   ...     for v, keydict in nbrs.items()
1004: (8)                   ...     for ekey, d in keydict.items()
1005: (8)                   ... ]
1006: (8)                   >>> MDG.update(edges=e)
1007: (8)                   See Also
1008: (8)                   --------
1009: (8)                   add_edges_from: add multiple edges to a graph
1010: (8)                   add_nodes_from: add multiple nodes to a graph
1011: (8)                   """
1012: (8)                   if edges is not None:
1013: (12)                      if nodes is not None:
1014: (16)                          self.add_nodes_from(nodes)
1015: (16)                          self.add_edges_from(edges)
1016: (12)                      else:
1017: (16)                          try:
1018: (20)                              graph_nodes = edges.nodes
1019: (20)                              graph_edges = edges.edges
1020: (16)                          except AttributeError:
1021: (20)                              self.add_edges_from(edges)
1022: (16)                          else:  # edges is Graph-like
1023: (20)                              self.add_nodes_from(graph_nodes.data())
1024: (20)                              self.add_edges_from(graph_edges.data())
1025: (20)                              self.graph.update(edges.graph)
1026: (8)                   elif nodes is not None:
1027: (12)                      self.add_nodes_from(nodes)
1028: (8)                   else:
1029: (12)                      raise NetworkXError("update needs nodes or edges input")
1030: (4)               def has_edge(self, u, v):
1031: (8)                   """Returns True if the edge (u, v) is in the graph.
1032: (8)                   This is the same as `v in G[u]` without KeyError exceptions.
1033: (8)                   Parameters
1034: (8)                   ----------
1035: (8)                   u, v : nodes
1036: (12)                      Nodes can be, for example, strings or numbers.
1037: (12)                      Nodes must be hashable (and not None) Python objects.
1038: (8)                   Returns
1039: (8)                   -------
1040: (8)                   edge_ind : bool
1041: (12)                      True if edge is in the graph, False otherwise.
1042: (8)                   Examples
1043: (8)                   --------
1044: (8)                   >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
1045: (8)                   >>> G.has_edge(0, 1)  # using two nodes
1046: (8)                   True
1047: (8)                   >>> e = (0, 1)
1048: (8)                   >>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
1049: (8)                   True
1050: (8)                   >>> e = (0, 1, {"weight": 7})
1051: (8)                   >>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
```

```
(8)                 True
(8)                 The following syntax are equivalent:
(8)                 >>> G.has_edge(0, 1)
(8)                 True
(8)                 >>> 1 in G[0]  # though this gives KeyError if 0 not in G
(8)                 True
(8)                 """
(8)                 try:
(12)                    return v in self._adj[u]
(8)                 except KeyError:
(12)                    return False
(4)             def neighbors(self, n):
(8)                 """Returns an iterator over all neighbors of node n.
(8)                 This is identical to `iter(G[n])`
(8)                 Parameters
(8)                 ----------
(8)                 n : node
(11)                   A node in the graph
(8)                 Returns
(8)                 -------
(8)                 neighbors : iterator
(12)                    An iterator over all neighbors of node n
(8)                 Raises
(8)                 ------
(8)                 NetworkXError
(12)                    If the node n is not in the graph.
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                 >>> [n for n in G.neighbors(0)]
(8)                 [1]
(8)                 Notes
(8)                 -----
(8)                 Alternate ways to access the neighbors are ``G.adj[n]`` or ``G[n]``:
(8)                 >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                 >>> G.add_edge("a", "b", weight=7)
(8)                 >>> G["a"]
(8)                 AtlasView({'b': {'weight': 7}})
(8)                 >>> G = nx.path_graph(4)
(8)                 >>> [n for n in G[0]]
(8)                 [1]
(8)                 """
(8)                 try:
(12)                    return iter(self._adj[n])
(8)                 except KeyError as err:
(12)                    raise NetworkXError(f"The node {n} is not in the graph.") from err
(4)             @cached_property
(4)             def edges(self):
(8)                 """An EdgeView of the Graph as G.edges or G.edges().
(8)                 edges(self, nbunch=None, data=False, default=None)
(8)                 The EdgeView provides set-like operations on the edge-tuples
(8)                 as well as edge attribute lookup. When called, it also provides
(8)                 an EdgeDataView object which allows control of access to edge
(8)                 attributes (but does not provide set-like operations).
(8)                 Hence, `G.edges[u, v]['color']` provides the value of the color
(8)                 attribute for edge `(u, v)` while
(8)                 `for (u, v, c) in G.edges.data('color', default='red'):`
(8)                 iterates through all the edges yielding the color attribute
(8)                 with default `'red'` if no color attribute exists.
(8)                 Parameters
(8)                 ----------
(8)                 nbunch : single node, container, or all nodes (default= all nodes)
(12)                    The view will only report edges from these nodes.
(8)                 data : string or bool, optional (default=False)
(12)                    The edge attribute returned in 3-tuple (u, v, ddict[data]).
(12)                    If True, return edge attribute dict in 3-tuple (u, v, ddict).
(12)                    If False, return 2-tuple (u, v).
(8)                 default : value, optional (default=None)
(12)                    Value used for edges that don't have the requested attribute.
```

```
            Only relevant if data is not True or False.
        Returns
        -------
        edges : EdgeView
            A view of edge attributes, usually it iterates over (u, v)
            or (u, v, d) tuples of edges, but can also be used for
            attribute lookup as `edges[u, v]['foo']`.
        Notes
        -----
        Nodes in nbunch that are not in the graph will be (quietly) ignored.
        For directed graphs this returns the out-edges.
        Examples
        --------
        >>> G = nx.path_graph(3)  # or MultiGraph, etc
        >>> G.add_edge(2, 3, weight=5)
        >>> [e for e in G.edges]
        [(0, 1), (1, 2), (2, 3)]
        >>> G.edges.data()  # default data is {} (empty dict)
        EdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
        >>> G.edges.data("weight", default=1)
        EdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
        >>> G.edges([0, 3])  # only edges from these nodes
        EdgeDataView([(0, 1), (3, 2)])
        >>> G.edges(0)  # only edges from node 0
        EdgeDataView([(0, 1)])
        """
        return EdgeView(self)
    def get_edge_data(self, u, v, default=None):
        """Returns the attribute dictionary associated with edge (u, v).
        This is identical to `G[u][v]` except the default is returned
        instead of an exception if the edge doesn't exist.
        Parameters
        ----------
        u, v : nodes
        default:  any Python object (default=None)
            Value to return if the edge (u, v) is not found.
        Returns
        -------
        edge_dict : dictionary
            The edge attribute dictionary.
        Examples
        --------
        >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G[0][1]
        {}
        Warning: Assigning to `G[u][v]` is not permitted.
        But it is safe to assign attributes `G[u][v]['foo']`
        >>> G[0][1]["weight"] = 7
        >>> G[0][1]["weight"]
        7
        >>> G[1][0]["weight"]
        7
        >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.get_edge_data(0, 1)  # default edge data is {}
        {}
        >>> e = (0, 1)
        >>> G.get_edge_data(*e)  # tuple form
        {}
        >>> G.get_edge_data("a", "b", default=0)  # edge not in graph, return 0
        0
        """
        try:
            return self._adj[u][v]
        except KeyError:
            return default
    def adjacency(self):
        """Returns an iterator over (node, adjacency dict) tuples for all nodes.
```

```
1188: (8)                    For directed graphs, only outgoing neighbors/adjacencies are included.
1189: (8)                    Returns
1190: (8)                    -------
1191: (8)                    adj_iter : iterator
1192: (11)                      An iterator over (node, adjacency dictionary) for all nodes in
1193: (11)                      the graph.
1194: (8)                    Examples
1195: (8)                    --------
1196: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
1197: (8)                    >>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
1198: (8)                    [(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
1199: (8)                    """
1200: (8)                    return iter(self._adj.items())
1201: (4)                @cached_property
1202: (4)                def degree(self):
1203: (8)                    """A DegreeView for the Graph as G.degree or G.degree().
1204: (8)                    The node degree is the number of edges adjacent to the node.
1205: (8)                    The weighted node degree is the sum of the edge weights for
1206: (8)                    edges incident to that node.
1207: (8)                    This object provides an iterator for (node, degree) as well as
1208: (8)                    lookup for the degree for a single node.
1209: (8)                    Parameters
1210: (8)                    ----------
1211: (8)                    nbunch : single node, container, or all nodes (default= all nodes)
1212: (12)                       The view will only report edges incident to these nodes.
1213: (8)                    weight : string or None, optional (default=None)
1214: (11)                      The name of an edge attribute that holds the numerical value used
1215: (11)                      as a weight.  If None, then each edge has weight 1.
1216: (11)                      The degree is the sum of the edge weights adjacent to the node.
1217: (8)                    Returns
1218: (8)                    -------
1219: (8)                    DegreeView or int
1220: (12)                       If multiple nodes are requested (the default), returns a
`DegreeView`
1221: (12)                       mapping nodes to their degree.
1222: (12)                       If a single node is requested, returns the degree of the node as
an integer.
1223: (8)                    Examples
1224: (8)                    --------
1225: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
1226: (8)                    >>> G.degree[0]  # node 0 has degree 1
1227: (8)                    1
1228: (8)                    >>> list(G.degree([0, 1, 2]))
1229: (8)                    [(0, 1), (1, 2), (2, 2)]
1230: (8)                    """
1231: (8)                    return DegreeView(self)
1232: (4)                def clear(self):
1233: (8)                    """Remove all nodes and edges from the graph.
1234: (8)                    This also removes the name, and all graph, node, and edge attributes.
1235: (8)                    Examples
1236: (8)                    --------
1237: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
1238: (8)                    >>> G.clear()
1239: (8)                    >>> list(G.nodes)
1240: (8)                    []
1241: (8)                    >>> list(G.edges)
1242: (8)                    []
1243: (8)                    """
1244: (8)                    self._adj.clear()
1245: (8)                    self._node.clear()
1246: (8)                    self.graph.clear()
1247: (8)                    nx._clear_cache(self)
1248: (4)                def clear_edges(self):
1249: (8)                    """Remove all edges from the graph without altering nodes.
1250: (8)                    Examples
1251: (8)                    --------
1252: (8)                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
1253: (8)                    >>> G.clear_edges()
1254: (8)                    >>> list(G.nodes)
```

```
(8)                 [0, 1, 2, 3]
(8)                 >>> list(G.edges)
(8)                 []
(8)                 """
(8)                 for nbr_dict in self._adj.values():
(12)                    nbr_dict.clear()
(8)                 nx._clear_cache(self)
(4)             def is_multigraph(self):
(8)                 """Returns True if graph is a multigraph, False otherwise."""
(8)                 return False
(4)             def is_directed(self):
(8)                 """Returns True if graph is directed, False otherwise."""
(8)                 return False
(4)             def copy(self, as_view=False):
(8)                 """Returns a copy of the graph.
(8)                 The copy method by default returns an independent shallow copy
(8)                 of the graph and attributes. That is, if an attribute is a
(8)                 container, that container is shared by the original an the copy.
(8)                 Use Python's `copy.deepcopy` for new containers.
(8)                 If `as_view` is True then a view is returned instead of a copy.
(8)                 Notes
(8)                 -----
(8)                 All copies reproduce the graph structure, but data attributes
(8)                 may be handled in different ways. There are four types of copies
(8)                 of a graph that people might want.
(8)                 Deepcopy -- A "deepcopy" copies the graph structure as well as
(8)                 all data attributes and any objects they might contain.
(8)                 The entire graph object is new so that changes in the copy
(8)                 do not affect the original object. (see Python's copy.deepcopy)
(8)                 Data Reference (Shallow) -- For a shallow copy the graph structure
(8)                 is copied but the edge, node and graph attribute dicts are
(8)                 references to those in the original graph. This saves
(8)                 time and memory but could cause confusion if you change an attribute
(8)                 in one graph and it changes the attribute in the other.
(8)                 NetworkX does not provide this level of shallow copy.
(8)                 Independent Shallow -- This copy creates new independent attribute
(8)                 dicts and then does a shallow copy of the attributes. That is, any
(8)                 attributes that are containers are shared between the new graph
(8)                 and the original. This is exactly what `dict.copy()` provides.
(8)                 You can obtain this style copy using:
(12)                    >>> G = nx.path_graph(5)
(12)                    >>> H = G.copy()
(12)                    >>> H = G.copy(as_view=False)
(12)                    >>> H = nx.Graph(G)
(12)                    >>> H = G.__class__(G)
(8)                 Fresh Data -- For fresh data, the graph structure is copied while
(8)                 new empty data attribute dicts are created. The resulting graph
(8)                 is independent of the original and it has no edge, node or graph
(8)                 attributes. Fresh copies are not enabled. Instead use:
(12)                    >>> H = G.__class__()
(12)                    >>> H.add_nodes_from(G)
(12)                    >>> H.add_edges_from(G.edges)
(8)                 View -- Inspired by dict-views, graph-views act like read-only
(8)                 versions of the original graph, providing a copy of the original
(8)                 structure without requiring any memory for copying the information.
(8)                 See the Python copy module for more information on shallow
(8)                 and deep copies, https://docs.python.org/3/library/copy.html.
(8)                 Parameters
(8)                 ----------
(8)                 as_view : bool, optional (default=False)
(12)                    If True, the returned graph-view provides a read-only view
(12)                    of the original graph without actually copying any data.
(8)                 Returns
(8)                 -------
(8)                 G : Graph
(12)                    A copy of the graph.
(8)                 See Also
(8)                 --------
(8)                 to_directed: return a directed copy of the graph.
```

```
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                 >>> H = G.copy()
(8)                 """
(8)                 if as_view is True:
(12)                    return nx.graphviews.generic_graph_view(self)
(8)                 G = self.__class__()
(8)                 G.graph.update(self.graph)
(8)                 G.add_nodes_from((n, d.copy()) for n, d in self._node.items())
(8)                 G.add_edges_from(
(12)                    (u, v, datadict.copy())
(12)                    for u, nbrs in self._adj.items()
(12)                    for v, datadict in nbrs.items()
(8)                 )
(8)                 return G
(4)             def to_directed(self, as_view=False):
(8)                 """Returns a directed representation of the graph.
(8)                 Returns
(8)                 -------
(8)                 G : DiGraph
(12)                    A directed graph with the same name, same nodes, and with
(12)                    each edge (u, v, data) replaced by two directed edges
(12)                    (u, v, data) and (v, u, data).
(8)                 Notes
(8)                 -----
(8)                 This returns a "deepcopy" of the edge, node, and
(8)                 graph attributes which attempts to completely copy
(8)                 all of the data and references.
(8)                 This is in contrast to the similar D=DiGraph(G) which returns a
(8)                 shallow copy of the data.
(8)                 See the Python copy module for more information on shallow
(8)                 and deep copies, https://docs.python.org/3/library/copy.html.
(8)                 Warning: If you have subclassed Graph to use dict-like objects
(8)                 in the data structure, those changes do not transfer to the
(8)                 DiGraph created by this method.
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.Graph()  # or MultiGraph, etc
(8)                 >>> G.add_edge(0, 1)
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1), (1, 0)]
(8)                 If already directed, return a (deep) copy
(8)                 >>> G = nx.DiGraph()  # or MultiDiGraph, etc
(8)                 >>> G.add_edge(0, 1)
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1)]
(8)                 """
(8)                 graph_class = self.to_directed_class()
(8)                 if as_view is True:
(12)                    return nx.graphviews.generic_graph_view(self, graph_class)
(8)                 G = graph_class()
(8)                 G.graph.update(deepcopy(self.graph))
(8)                 G.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
(8)                 G.add_edges_from(
(12)                    (u, v, deepcopy(data))
(12)                    for u, nbrs in self._adj.items()
(12)                    for v, data in nbrs.items()
(8)                 )
(8)                 return G
(4)             def to_undirected(self, as_view=False):
(8)                 """Returns an undirected copy of the graph.
(8)                 Parameters
(8)                 ----------
(8)                 as_view : bool (optional, default=False)
(10)                  If True return a view of the original undirected graph.
(8)                 Returns
```

```
(8)                 -------
(8)                 G : Graph/MultiGraph
(12)                    A deepcopy of the graph.
(8)                 See Also
(8)                 --------
(8)                 Graph, copy, add_edge, add_edges_from
(8)                 Notes
(8)                 -----
(8)                 This returns a "deepcopy" of the edge, node, and
(8)                 graph attributes which attempts to completely copy
(8)                 all of the data and references.
(8)                 This is in contrast to the similar `G = nx.DiGraph(D)` which returns a
(8)                 shallow copy of the data.
(8)                 See the Python copy module for more information on shallow
(8)                 and deep copies, https://docs.python.org/3/library/copy.html.
(8)                 Warning: If you have subclassed DiGraph to use dict-like objects
(8)                 in the data structure, those changes do not transfer to the
(8)                 Graph created by this method.
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.path_graph(2)  # or MultiGraph, etc
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1), (1, 0)]
(8)                 >>> G2 = H.to_undirected()
(8)                 >>> list(G2.edges)
(8)                 [(0, 1)]
(8)                 """
(8)                 graph_class = self.to_undirected_class()
(8)                 if as_view is True:
(12)                    return nx.graphviews.generic_graph_view(self, graph_class)
(8)                 G = graph_class()
(8)                 G.graph.update(deepcopy(self.graph))
(8)                 G.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
(8)                 G.add_edges_from(
(12)                    (u, v, deepcopy(d))
(12)                    for u, nbrs in self._adj.items()
(12)                    for v, d in nbrs.items()
(8)                 )
(8)                 return G
(4)             def subgraph(self, nodes):
(8)                 """Returns a SubGraph view of the subgraph induced on `nodes`.
(8)                 The induced subgraph of the graph contains the nodes in `nodes`
(8)                 and the edges between those nodes.
(8)                 Parameters
(8)                 ----------
(8)                 nodes : list, iterable
(12)                    A container of nodes which will be iterated through once.
(8)                 Returns
(8)                 -------
(8)                 G : SubGraph View
(12)                    A subgraph view of the graph. The graph structure cannot be
(12)                    changed but node/edge attributes can and are shared with the
(12)                    original graph.
(8)                 Notes
(8)                 -----
(8)                 The graph, edge and node attributes are shared with the original
graph.
(8)                 Changes to the graph structure is ruled out by the view, but changes
(8)                 to attributes are reflected in the original graph.
(8)                 To create a subgraph with its own copy of the edge/node attributes
use:
(8)                 G.subgraph(nodes).copy()
(8)                 For an inplace reduction of a graph to a subgraph you can remove
nodes:
(8)                 G.remove_nodes_from([n for n in G if n not in set(nodes)])
(8)                 Subgraph views are sometimes NOT what you want. In most cases where
(8)                 you want to do more than simply look at the induced edges, it makes
(8)                 more sense to just create the subgraph as its own graph with code
```

```
like:
                    ::
                        SG = G.__class__()
                        SG.add_nodes_from((n, G.nodes[n]) for n in largest_wcc)
                        if SG.is_multigraph():
                            SG.add_edges_from(
                                (n, nbr, key, d)
                                for n, nbrs in G.adj.items()
                                if n in largest_wcc
                                for nbr, keydict in nbrs.items()
                                if nbr in largest_wcc
                                for key, d in keydict.items()
                            )
                        else:
                            SG.add_edges_from(
                                (n, nbr, d)
                                for n, nbrs in G.adj.items()
                                if n in largest_wcc
                                for nbr, d in nbrs.items()
                                if nbr in largest_wcc
                            )
                        SG.graph.update(G.graph)
                    Examples
                    --------
                    >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
                    >>> H = G.subgraph([0, 1, 2])
                    >>> list(H.edges)
                    [(0, 1), (1, 2)]
                    """
                    induced_nodes = nx.filters.show_nodes(self.nbunch_iter(nodes))
                    subgraph = nx.subgraph_view
                    if hasattr(self, "_NODE_OK"):
                        return subgraph(
                            self._graph, filter_node=induced_nodes,
filter_edge=self._EDGE_OK
                        )
                    return subgraph(self, filter_node=induced_nodes)
                def edge_subgraph(self, edges):
                    """Returns the subgraph induced by the specified edges.
                    The induced subgraph contains each edge in `edges` and each
                    node incident to any one of those edges.
                    Parameters
                    ----------
                    edges : iterable
                        An iterable of edges in this graph.
                    Returns
                    -------
                    G : Graph
                        An edge-induced subgraph of this graph with the same edge
                        attributes.
                    Notes
                    -----
                    The graph, edge, and node attributes in the returned subgraph
                    view are references to the corresponding attributes in the original
                    graph. The view is read-only.
                    To create a full graph version of the subgraph with its own copy
                    of the edge or node attributes, use::
                        G.edge_subgraph(edges).copy()
                    Examples
                    --------
                    >>> G = nx.path_graph(5)
                    >>> H = G.edge_subgraph([(0, 1), (3, 4)])
                    >>> list(H.nodes)
                    [0, 1, 3, 4]
                    >>> list(H.edges)
                    [(0, 1), (3, 4)]
                    """
                    return nx.edge_subgraph(self, edges)
                def size(self, weight=None):
```

```
        """Returns the number of edges or total of all edge weights.
        Parameters
        ----------
        weight : string or None, optional (default=None)
            The edge attribute that holds the numerical value used
            as a weight. If None, then each edge has weight 1.
        Returns
        -------
        size : numeric
            The number of edges or
            (if weight keyword is provided) the total weight sum.
            If weight is None, returns an int. Otherwise a float
            (or more general numeric if the weights are more general).
        See Also
        --------
        number_of_edges
        Examples
        --------
        >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.size()
        3
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.add_edge("a", "b", weight=2)
        >>> G.add_edge("b", "c", weight=4)
        >>> G.size()
        2
        >>> G.size(weight="weight")
        6.0
        """
        s = sum(d for v, d in self.degree(weight=weight))
        return s // 2 if weight is None else s / 2
    def number_of_edges(self, u=None, v=None):
        """Returns the number of edges between two nodes.
        Parameters
        ----------
        u, v : nodes, optional (default=all edges)
            If u and v are specified, return the number of edges between
            u and v. Otherwise return the total number of all edges.
        Returns
        -------
        nedges : int
            The number of edges in the graph.  If nodes `u` and `v` are
            specified return the number of edges between those nodes. If
            the graph is directed, this only returns the number of edges
            from `u` to `v`.
        See Also
        --------
        size
        Examples
        --------
        For undirected graphs, this method counts the total number of
        edges in the graph:
        >>> G = nx.path_graph(4)
        >>> G.number_of_edges()
        3
        If you specify two nodes, this counts the total number of edges
        joining the two nodes:
        >>> G.number_of_edges(0, 1)
        1
        For directed graphs, this method can count the total number of
        directed edges from `u` to `v`:
        >>> G = nx.DiGraph()
        >>> G.add_edge(0, 1)
        >>> G.add_edge(1, 0)
        >>> G.number_of_edges(0, 1)
        1
        """
        if u is None:
            return int(self.size())
```

```
1595: (8)                    if v in self._adj[u]:
1596: (12)                       return 1
1597: (8)                    return 0
1598: (4)                def nbunch_iter(self, nbunch=None):
1599: (8)                    """Returns an iterator over nodes contained in nbunch that are
1600: (8)                    also in the graph.
1601: (8)                    The nodes in nbunch are checked for membership in the graph
1602: (8)                    and if not are silently ignored.
1603: (8)                    Parameters
1604: (8)                    ----------
1605: (8)                    nbunch : single node, container, or all nodes (default= all nodes)
1606: (12)                       The view will only report edges incident to these nodes.
1607: (8)                    Returns
1608: (8)                    -------
1609: (8)                    niter : iterator
1610: (12)                       An iterator over nodes in nbunch that are also in the graph.
1611: (12)                       If nbunch is None, iterate over all nodes in the graph.
1612: (8)                    Raises
1613: (8)                    ------
1614: (8)                    NetworkXError
1615: (12)                       If nbunch is not a node or sequence of nodes.
1616: (12)                       If a node in nbunch is not hashable.
1617: (8)                    See Also
1618: (8)                    --------
1619: (8)                    Graph.__iter__
1620: (8)                    Notes
1621: (8)                    -----
1622: (8)                    When nbunch is an iterator, the returned iterator yields values
1623: (8)                    directly from nbunch, becoming exhausted when nbunch is exhausted.
1624: (8)                    To test whether nbunch is a single node, one can use
1625: (8)                    "if nbunch in self:", even after processing with this routine.
1626: (8)                    If nbunch is not a node or a (possibly empty) sequence/iterator
1627: (8)                    or None, a :exc:`NetworkXError` is raised.  Also, if any object in
1628: (8)                    nbunch is not hashable, a :exc:`NetworkXError` is raised.
1629: (8)                    """
1630: (8)                    if nbunch is None:  # include all nodes via iterator
1631: (12)                       bunch = iter(self._adj)
1632: (8)                    elif nbunch in self:  # if nbunch is a single node
1633: (12)                       bunch = iter([nbunch])
1634: (8)                    else:  # if nbunch is a sequence of nodes
1635: (12)                       def bunch_iter(nlist, adj):
1636: (16)                           try:
1637: (20)                               for n in nlist:
1638: (24)                                   if n in adj:
1639: (28)                                       yield n
1640: (16)                           except TypeError as err:
1641: (20)                               exc, message = err, err.args[0]
1642: (20)                               if "iter" in message:
1643: (24)                                   exc = NetworkXError(
1644: (28)                                       "nbunch is not a node or a sequence of nodes."
1645: (24)                                   )
1646: (20)                               if "hashable" in message:
1647: (24)                                   exc = NetworkXError(
1648: (28)                                       f"Node {n} in sequence nbunch is not a valid
node."
1649: (24)                                   )
1650: (20)                               raise exc
1651: (12)                       bunch = bunch_iter(nbunch, self._adj)
1652: (8)                    return bunch
```

File 394 - graphviews.py:

```
1: (0)              """View of Graphs as SubGraph, Reverse, Directed, Undirected.
2: (0)              In some algorithms it is convenient to temporarily morph
3: (0)              a graph to exclude some nodes or edges. It should be better
4: (0)              to do that via a view than to remove and then re-add.
5: (0)              In other algorithms it is convenient to temporarily morph
```

```
(0)             a graph to reverse directed edges, or treat a directed graph
(0)             as undirected, etc. This module provides those graph views.
(0)             The resulting views are essentially read-only graphs that
(0)             report data from the original graph object. We provide an
(0)             attribute G._graph which points to the underlying graph object.
(0)             Note: Since graphviews look like graphs, one can end up with
(0)             view-of-view-of-view chains. Be careful with chains because
(0)             they become very slow with about 15 nested views.
(0)             For the common simple case of node induced subgraphs created
(0)             from the graph class, we short-cut the chain by returning a
(0)             subgraph of the original graph directly rather than a subgraph
(0)             of a subgraph. We are careful not to disrupt any edge filter in
(0)             the middle subgraph. In general, determining how to short-cut
(0)             the chain is tricky and much harder with restricted_views than
(0)             with induced subgraphs.
(0)             Often it is easiest to use .copy() to avoid chains.
(0)             """
(0)             import networkx as nx
(0)             from networkx.classes.coreviews import (
(4)                 FilterAdjacency,
(4)                 FilterAtlas,
(4)                 FilterMultiAdjacency,
(4)                 UnionAdjacency,
(4)                 UnionMultiAdjacency,
(0)             )
(0)             from networkx.classes.filters import no_filter
(0)             from networkx.exception import NetworkXError
(0)             from networkx.utils import deprecate_positional_args, not_implemented_for
(0)             __all__ = ["generic_graph_view", "subgraph_view", "reverse_view"]
(0)             def generic_graph_view(G, create_using=None):
(4)                 """Returns a read-only view of `G`.
(4)                 The graph `G` and its attributes are not copied but viewed through the new
graph object
(4)                 of the same class as `G` (or of the class specified in `create_using`).
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A directed/undirected graph/multigraph.
(4)                 create_using : NetworkX graph constructor, optional (default=None)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(7)                    If `None`, then the appropriate Graph type is inferred from `G`.
(4)                 Returns
(4)                 -------
(4)                 newG : graph
(8)                     A view of the input graph `G` and its attributes as viewed through
(8)                     the `create_using` class.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If `G` is a multigraph (or multidigraph) but `create_using` is not, or
vice versa.
(4)                 Notes
(4)                 -----
(4)                 The returned graph view is read-only (cannot modify the graph).
(4)                 Yet the view reflects any changes in `G`. The intent is to mimic dict
views.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edge(1, 2, weight=0.3)
(4)                 >>> G.add_edge(2, 3, weight=0.5)
(4)                 >>> G.edges(data=True)
(4)                 EdgeDataView([(1, 2, {'weight': 0.3}), (2, 3, {'weight': 0.5})])
(4)                 The view exposes the attributes from the original graph.
(4)                 >>> viewG = nx.graphviews.generic_graph_view(G)
(4)                 >>> viewG.edges(data=True)
(4)                 EdgeDataView([(1, 2, {'weight': 0.3}), (2, 3, {'weight': 0.5})])
(4)                 Changes to `G` are reflected in `viewG`.
(4)                 >>> G.remove_edge(2, 3)
```

```
(4)                 >>> G.edges(data=True)
(4)                 EdgeDataView([(1, 2, {'weight': 0.3})])
(4)                 >>> viewG.edges(data=True)
(4)                 EdgeDataView([(1, 2, {'weight': 0.3})])
(4)                 We can change the graph type with the `create_using` parameter.
(4)                 >>> type(G)
(4)                 <class 'networkx.classes.graph.Graph'>
(4)                 >>> viewDG = nx.graphviews.generic_graph_view(G, create_using=nx.DiGraph)
(4)                 >>> type(viewDG)
(4)                 <class 'networkx.classes.digraph.DiGraph'>
(4)                 """
(4)                 if create_using is None:
(8)                     newG = G.__class__()
(4)                 else:
(8)                     newG = nx.empty_graph(0, create_using)
(4)                 if G.is_multigraph() != newG.is_multigraph():
(8)                     raise NetworkXError("Multigraph for G must agree with create_using")
(4)                 newG = nx.freeze(newG)
(4)                 newG._graph = G
(4)                 newG.graph = G.graph
(4)                 newG._node = G._node
(4)                 if newG.is_directed():
(8)                     if G.is_directed():
(12)                        newG._succ = G._succ
(12)                        newG._pred = G._pred
(8)                     else:
(12)                        newG._succ = G._adj
(12)                        newG._pred = G._adj
(4)                 elif G.is_directed():
(8)                     if G.is_multigraph():
(12)                        newG._adj = UnionMultiAdjacency(G._succ, G._pred)
(8)                     else:
(12)                        newG._adj = UnionAdjacency(G._succ, G._pred)
(4)                 else:
(8)                     newG._adj = G._adj
(4)                 return newG
(0)             @deprecate_positional_args(version="3.4")
(0)             def subgraph_view(G, *, filter_node=no_filter, filter_edge=no_filter):
(4)                 """View of `G` applying a filter on nodes and edges.
(4)                 `subgraph_view` provides a read-only view of the input graph that excludes
(4)                 nodes and edges based on the outcome of two filter functions `filter_node`
(4)                 and `filter_edge`.
(4)                 The `filter_node` function takes one argument --- the node --- and returns
(4)                 `True` if the node should be included in the subgraph, and `False` if it
(4)                 should not be included.
(4)                 The `filter_edge` function takes two (or three arguments if `G` is a
(4)                 multi-graph) --- the nodes describing an edge, plus the edge-key if
(4)                 parallel edges are possible --- and returns `True` if the edge should be
(4)                 included in the subgraph, and `False` if it should not be included.
(4)                 Both node and edge filter functions are called on graph elements as they
(4)                 are queried, meaning there is no up-front cost to creating the view.
(4)                 Parameters
(4)                 ----------
(4)                 G : networkx.Graph
(8)                     A directed/undirected graph/multigraph
(4)                 filter_node : callable, optional
(8)                     A function taking a node as input, which returns `True` if the node
(8)                     should appear in the view.
(4)                 filter_edge : callable, optional
(8)                     A function taking as input the two nodes describing an edge (plus the
(8)                     edge-key if `G` is a multi-graph), which returns `True` if the edge
(8)                     should appear in the view.
(4)                 Returns
(4)                 -------
(4)                 graph : networkx.Graph
(8)                     A read-only graph view of the input graph.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(6)
```

```
(4)                 Filter functions operate on the node, and return `True` if the node should
(4)                 appear in the view:
(4)                 >>> def filter_node(n1):
(4)                 ...     return n1 != 5
(4)                 >>> view = nx.subgraph_view(G, filter_node=filter_node)
(4)                 >>> view.nodes()
(4)                 NodeView((0, 1, 2, 3, 4))
(4)                 We can use a closure pattern to filter graph elements based on additional
(4)                 data --- for example, filtering on edge data attached to the graph:
(4)                 >>> G[3][4]["cross_me"] = False
(4)                 >>> def filter_edge(n1, n2):
(4)                 ...     return G[n1][n2].get("cross_me", True)
(4)                 >>> view = nx.subgraph_view(G, filter_edge=filter_edge)
(4)                 >>> view.edges()
(4)                 EdgeView([(0, 1), (1, 2), (2, 3), (4, 5)])
(4)                 >>> view = nx.subgraph_view(
(4)                 ...     G,
(4)                 ...     filter_node=filter_node,
(4)                 ...     filter_edge=filter_edge,
(4)                 ... )
(4)                 >>> view.nodes()
(4)                 NodeView((0, 1, 2, 3, 4))
(4)                 >>> view.edges()
(4)                 EdgeView([(0, 1), (1, 2), (2, 3)])
(4)                 """
(4)                 newG = nx.freeze(G.__class__())
(4)                 newG._NODE_OK = filter_node
(4)                 newG._EDGE_OK = filter_edge
(4)                 newG._graph = G
(4)                 newG.graph = G.graph
(4)                 newG._node = FilterAtlas(G._node, filter_node)
(4)                 if G.is_multigraph():
(8)                     Adj = FilterMultiAdjacency
(8)                     def reverse_edge(u, v, k=None):
(12)                        return filter_edge(v, u, k)
(4)                 else:
(8)                     Adj = FilterAdjacency
(8)                     def reverse_edge(u, v, k=None):
(12)                        return filter_edge(v, u)
(4)                 if G.is_directed():
(8)                     newG._succ = Adj(G._succ, filter_node, filter_edge)
(8)                     newG._pred = Adj(G._pred, filter_node, reverse_edge)
(4)                 else:
(8)                     newG._adj = Adj(G._adj, filter_node, filter_edge)
(4)                 return newG
(0)             @not_implemented_for("undirected")
(0)             def reverse_view(G):
(4)                 """View of `G` with edge directions reversed
(4)                 `reverse_view` returns a read-only view of the input graph where
(4)                 edge directions are reversed.
(4)                 Identical to digraph.reverse(copy=False)
(4)                 Parameters
(4)                 ----------
(4)                 G : networkx.DiGraph
(4)                 Returns
(4)                 -------
(4)                 graph : networkx.DiGraph
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.DiGraph()
(4)                 >>> G.add_edge(1, 2)
(4)                 >>> G.add_edge(2, 3)
(4)                 >>> G.edges()
(4)                 OutEdgeView([(1, 2), (2, 3)])
(4)                 >>> view = nx.reverse_view(G)
(4)                 >>> view.edges()
(4)                 OutEdgeView([(2, 1), (3, 2)])
(4)                 """
(4)                 newG = generic_graph_view(G)
```

```
210: (4)                newG._succ, newG._pred = G._pred, G._succ
211: (4)                return newG
```

File 395 - multidigraph.py:

```
1: (0)              """Base class for MultiDiGraph."""
2: (0)              from copy import deepcopy
3: (0)              from functools import cached_property
4: (0)              import networkx as nx
5: (0)              from networkx import convert
6: (0)              from networkx.classes.coreviews import MultiAdjacencyView
7: (0)              from networkx.classes.digraph import DiGraph
8: (0)              from networkx.classes.multigraph import MultiGraph
9: (0)              from networkx.classes.reportviews import (
10: (4)                 DiMultiDegreeView,
11: (4)                 InMultiDegreeView,
12: (4)                 InMultiEdgeView,
13: (4)                 OutMultiDegreeView,
14: (4)                 OutMultiEdgeView,
15: (0)             )
16: (0)             from networkx.exception import NetworkXError
17: (0)             __all__ = ["MultiDiGraph"]
18: (0)             class MultiDiGraph(MultiGraph, DiGraph):
19: (4)                 """A directed graph class that can store multiedges.
20: (4)                 Multiedges are multiple edges between two nodes.  Each edge
21: (4)                 can hold optional data or attributes.
22: (4)                 A MultiDiGraph holds directed edges.  Self loops are allowed.
23: (4)                 Nodes can be arbitrary (hashable) Python objects with optional
24: (4)                 key/value attributes. By convention `None` is not used as a node.
25: (4)                 Edges are represented as links between nodes with optional
26: (4)                 key/value attributes.
27: (4)                 Parameters
28: (4)                 ----------
29: (4)                 incoming_graph_data : input graph (optional, default: None)
30: (8)                     Data to initialize graph. If None (default) an empty
31: (8)                     graph is created.  The data can be any format that is supported
32: (8)                     by the to_networkx_graph() function, currently including edge list,
33: (8)                     dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy
34: (8)                     sparse matrix, or PyGraphviz graph.
35: (4)                 multigraph_input : bool or None (default None)
36: (8)                     Note: Only used when `incoming_graph_data` is a dict.
37: (8)                     If True, `incoming_graph_data` is assumed to be a
38: (8)                     dict-of-dict-of-dict-of-dict structure keyed by
39: (8)                     node to neighbor to edge keys to edge data for multi-edges.
40: (8)                     A NetworkXError is raised if this is not the case.
41: (8)                     If False, :func:`to_networkx_graph` is used to try to determine
42: (8)                     the dict's graph data structure as either a dict-of-dict-of-dict
43: (8)                     keyed by node to neighbor to edge data, or a dict-of-iterable
44: (8)                     keyed by node to neighbors.
45: (8)                     If None, the treatment for True is tried, but if it fails,
46: (8)                     the treatment for False is tried.
47: (4)                 attr : keyword arguments, optional (default= no attributes)
48: (8)                     Attributes to add to graph as key=value pairs.
49: (4)                 See Also
50: (4)                 --------
51: (4)                 Graph
52: (4)                 DiGraph
53: (4)                 MultiGraph
54: (4)                 Examples
55: (4)                 --------
56: (4)                 Create an empty graph structure (a "null graph") with no nodes and
57: (4)                 no edges.
58: (4)                 >>> G = nx.MultiDiGraph()
59: (4)                 G can be grown in several ways.
60: (4)                 **Nodes:**
61: (4)                 Add one node at a time:
62: (4)                 >>> G.add_node(1)
```

```
(4)                 Add the nodes from any container (a list, dict, set or
(4)                 even the lines from a file or the nodes from another graph).
(4)                 >>> G.add_nodes_from([2, 3])
(4)                 >>> G.add_nodes_from(range(100, 110))
(4)                 >>> H = nx.path_graph(10)
(4)                 >>> G.add_nodes_from(H)
(4)                 In addition to strings and integers any hashable Python object
(4)                 (except None) can represent a node, e.g. a customized node object,
(4)                 or even another Graph.
(4)                 >>> G.add_node(H)
(4)                 **Edges:**
(4)                 G can also be grown by adding edges.
(4)                 Add one edge,
(4)                 >>> key = G.add_edge(1, 2)
(4)                 a list of edges,
(4)                 >>> keys = G.add_edges_from([(1, 2), (1, 3)])
(4)                 or a collection of edges,
(4)                 >>> keys = G.add_edges_from(H.edges)
(4)                 If some edges connect nodes not yet in the graph, the nodes
(4)                 are added automatically.  If an edge already exists, an additional
(4)                 edge is created and stored using a key to identify the edge.
(4)                 By default the key is the lowest unused integer.
(4)                 >>> keys = G.add_edges_from([(4, 5, dict(route=282)), (4, 5,
dict(route=37))])
(4)                 >>> G[4]
(4)                 AdjacencyView({5: {0: {}, 1: {'route': 282}, 2: {'route': 37}}})
(4)                 **Attributes:**
(4)                 Each graph, node, and edge can hold key/value attribute pairs
(4)                 in an associated attribute dictionary (the keys must be hashable).
(4)                 By default these are empty, but can be added or changed using
(4)                 add_edge, add_node or direct manipulation of the attribute
(4)                 dictionaries named graph, node and edge respectively.
(4)                 >>> G = nx.MultiDiGraph(day="Friday")
(4)                 >>> G.graph
(4)                 {'day': 'Friday'}
(4)                 Add node attributes using add_node(), add_nodes_from() or G.nodes
(4)                 >>> G.add_node(1, time="5pm")
(4)                 >>> G.add_nodes_from([3], time="2pm")
(4)                 >>> G.nodes[1]
(4)                 {'time': '5pm'}
(4)                 >>> G.nodes[1]["room"] = 714
(4)                 >>> del G.nodes[1]["room"]  # remove attribute
(4)                 >>> list(G.nodes(data=True))
(4)                 [(1, {'time': '5pm'}), (3, {'time': '2pm'})]
(4)                 Add edge attributes using add_edge(), add_edges_from(), subscript
(4)                 notation, or G.edges.
(4)                 >>> key = G.add_edge(1, 2, weight=4.7)
(4)                 >>> keys = G.add_edges_from([(3, 4), (4, 5)], color="red")
(4)                 >>> keys = G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight":
8})])
(4)                 >>> G[1][2][0]["weight"] = 4.7
(4)                 >>> G.edges[1, 2, 0]["weight"] = 4
(4)                 Warning: we protect the graph data structure by making `G.edges[1,
(4)                 2, 0]` a read-only dict-like structure. However, you can assign to
(4)                 attributes in e.g. `G.edges[1, 2, 0]`. Thus, use 2 sets of brackets
(4)                 to add/change data attributes: `G.edges[1, 2, 0]['weight'] = 4`
(4)                 (for multigraphs the edge key is required: `MG.edges[u, v,
(4)                 key][name] = value`).
(4)                 **Shortcuts:**
(4)                 Many common graph features allow python syntax to speed reporting.
(4)                 >>> 1 in G  # check if node in graph
(4)                 True
(4)                 >>> [n for n in G if n < 3]  # iterate through nodes
(4)                 [1, 2]
(4)                 >>> len(G)  # number of nodes in graph
(4)                 5
(4)                 >>> G[1]  # adjacency dict-like view mapping neighbor -> edge key -> edge
attributes
(4)                 AdjacencyView({2: {0: {'weight': 4}, 1: {'color': 'blue'}}})
```

```
(4)                 Often the best way to traverse all edges of a graph is via the neighbors.
(4)                 The neighbors are available as an adjacency-view `G.adj` object or via
(4)                 the method `G.adjacency()`.
(4)                 >>> for n, nbrsdict in G.adjacency():
(4)                 ...     for nbr, keydict in nbrsdict.items():
(4)                 ...         for key, eattr in keydict.items():
(4)                 ...             if "weight" in eattr:
(4)                 ...                 # Do something useful with the edges
(4)                 ...                 pass
(4)                 But the edges() method is often more convenient:
(4)                 >>> for u, v, keys, weight in G.edges(data="weight", keys=True):
(4)                 ...     if weight is not None:
(4)                 ...         # Do something useful with the edges
(4)                 ...         pass
(4)                 **Reporting:**
(4)                 Simple graph information is obtained using methods and object-attributes.
(4)                 Reporting usually provides views instead of containers to reduce memory
(4)                 usage. The views update as the graph is updated similarly to dict-views.
(4)                 The objects `nodes`, `edges` and `adj` provide access to data attributes
(4)                 via lookup (e.g. `nodes[n]`, `edges[u, v, k]`, `adj[u][v]`) and iteration
(4)                 (e.g. `nodes.items()`, `nodes.data('color')`,
(4)                 `nodes.data('color', default='blue')` and similarly for `edges`)
(4)                 Views exist for `nodes`, `edges`, `neighbors()`/`adj` and `degree`.
(4)                 For details on these and other miscellaneous methods, see below.
(4)                 **Subclasses (Advanced):**
(4)                 The MultiDiGraph class uses a dict-of-dict-of-dict-of-dict structure.
(4)                 The outer dict (node_dict) holds adjacency information keyed by node.
(4)                 The next dict (adjlist_dict) represents the adjacency information
(4)                 and holds edge_key dicts keyed by neighbor. The edge_key dict holds
(4)                 each edge_attr dict keyed by edge key. The inner dict
(4)                 (edge_attr_dict) represents the edge data and holds edge attribute
(4)                 values keyed by attribute names.
(4)                 Each of these four dicts in the dict-of-dict-of-dict-of-dict
(4)                 structure can be replaced by a user defined dict-like object.
(4)                 In general, the dict-like features should be maintained but
(4)                 extra features can be added. To replace one of the dicts create
(4)                 a new graph class by changing the class(!) variable holding the
(4)                 factory for that dict-like structure. The variable names are
(4)                 node_dict_factory, node_attr_dict_factory, adjlist_inner_dict_factory,
(4)                 adjlist_outer_dict_factory, edge_key_dict_factory, edge_attr_dict_factory
(4)                 and graph_attr_dict_factory.
(4)                 node_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the dict containing node
(8)                     attributes, keyed by node id.
(8)                     It should require no arguments and return a dict-like object
(4)                 node_attr_dict_factory: function, (default: dict)
(8)                     Factory function to be used to create the node attribute
(8)                     dict which holds attribute values keyed by attribute name.
(8)                     It should require no arguments and return a dict-like object
(4)                 adjlist_outer_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the outer-most dict
(8)                     in the data structure that holds adjacency info keyed by node.
(8)                     It should require no arguments and return a dict-like object.
(4)                 adjlist_inner_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the adjacency list
(8)                     dict which holds multiedge key dicts keyed by neighbor.
(8)                     It should require no arguments and return a dict-like object.
(4)                 edge_key_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the edge key dict
(8)                     which holds edge data keyed by edge key.
(8)                     It should require no arguments and return a dict-like object.
(4)                 edge_attr_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the edge attribute
(8)                     dict which holds attribute values keyed by attribute name.
(8)                     It should require no arguments and return a dict-like object.
(4)                 graph_attr_dict_factory : function, (default: dict)
(8)                     Factory function to be used to create the graph attribute
(8)                     dict which holds attribute values keyed by attribute name.
(8)                     It should require no arguments and return a dict-like object.
```

```
    Typically, if your extension doesn't impact the data structure all
    methods will inherited without issue except: `to_directed/to_undirected`.
    By default these methods create a DiGraph/Graph class and you probably
    want them to create your extension of a DiGraph/Graph. To facilitate
    this we define two class variables that you can set in your subclass.
    to_directed_class : callable, (default: DiGraph or MultiDiGraph)
        Class to create a new graph structure in the `to_directed` method.
        If `None`, a NetworkX class (DiGraph or MultiDiGraph) is used.
    to_undirected_class : callable, (default: Graph or MultiGraph)
        Class to create a new graph structure in the `to_undirected` method.
        If `None`, a NetworkX class (Graph or MultiGraph) is used.
    **Subclassing Example**
    Create a low memory graph class that effectively disallows edge
    attributes by using a single attribute dict for all edges.
    This reduces the memory used, but you lose edge attributes.
    >>> class ThinGraph(nx.Graph):
    ...     all_edge_dict = {"weight": 1}
    ...
    ...     def single_edge_dict(self):
    ...         return self.all_edge_dict
    ...
    ...     edge_attr_dict_factory = single_edge_dict
    >>> G = ThinGraph()
    >>> G.add_edge(2, 1)
    >>> G[2][1]
    {'weight': 1}
    >>> G.add_edge(2, 2)
    >>> G[2][1] is G[2][2]
    True
    """
    edge_key_dict_factory = dict
    def __init__(self, incoming_graph_data=None, multigraph_input=None, **attr):
        """Initialize a graph with edges, name, or graph attributes.
        Parameters
        ----------
        incoming_graph_data : input graph
            Data to initialize graph.  If incoming_graph_data=None (default)
            an empty graph is created.  The data can be an edge list, or any
            NetworkX graph object.  If the corresponding optional Python
            packages are installed the data can also be a 2D NumPy array, a
            SciPy sparse array, or a PyGraphviz graph.
        multigraph_input : bool or None (default None)
            Note: Only used when `incoming_graph_data` is a dict.
            If True, `incoming_graph_data` is assumed to be a
            dict-of-dict-of-dict-of-dict structure keyed by
            node to neighbor to edge keys to edge data for multi-edges.
            A NetworkXError is raised if this is not the case.
            If False, :func:`to_networkx_graph` is used to try to determine
            the dict's graph data structure as either a dict-of-dict-of-dict
            keyed by node to neighbor to edge data, or a dict-of-iterable
            keyed by node to neighbors.
            If None, the treatment for True is tried, but if it fails,
            the treatment for False is tried.
        attr : keyword arguments, optional (default= no attributes)
            Attributes to add to graph as key=value pairs.
        See Also
        --------
        convert
        Examples
        --------
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G = nx.Graph(name="my graph")
        >>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
        >>> G = nx.Graph(e)
        Arbitrary graph attribute pairs (key=value) may be assigned
        >>> G = nx.Graph(e, day="Friday")
        >>> G.graph
        {'day': 'Friday'}
```

```
266: (8)                    """
267: (8)                    if isinstance(incoming_graph_data, dict) and multigraph_input is not
False:
268: (12)                       DiGraph.__init__(self)
269: (12)                       try:
270: (16)                           convert.from_dict_of_dicts(
271: (20)                               incoming_graph_data, create_using=self,
multigraph_input=True
272: (16)                           )
273: (16)                           self.graph.update(attr)
274: (12)                       except Exception as err:
275: (16)                           if multigraph_input is True:
276: (20)                               raise nx.NetworkXError(
277: (24)                                   f"converting multigraph_input raised:\n{type(err)}:
{err}"
278: (20)                               )
279: (16)                           DiGraph.__init__(self, incoming_graph_data, **attr)
280: (8)                    else:
281: (12)                       DiGraph.__init__(self, incoming_graph_data, **attr)
282: (4)                @cached_property
283: (4)                def adj(self):
284: (8)                    """Graph adjacency object holding the neighbors of each node.
285: (8)                    This object is a read-only dict-like structure with node keys
286: (8)                    and neighbor-dict values.  The neighbor-dict is keyed by neighbor
287: (8)                    to the edgekey-dict.  So `G.adj[3][2][0]['color'] = 'blue'` sets
288: (8)                    the color of the edge `(3, 2, 0)` to `"blue"`.
289: (8)                    Iterating over G.adj behaves like a dict. Useful idioms include
290: (8)                    `for nbr, datadict in G.adj[n].items():`.
291: (8)                    The neighbor information is also provided by subscripting the graph.
292: (8)                    So `for nbr, foovalue in G[node].data('foo', default=1):` works.
293: (8)                    For directed graphs, `G.adj` holds outgoing (successor) info.
294: (8)                    """
295: (8)                    return MultiAdjacencyView(self._succ)
296: (4)                @cached_property
297: (4)                def succ(self):
298: (8)                    """Graph adjacency object holding the successors of each node.
299: (8)                    This object is a read-only dict-like structure with node keys
300: (8)                    and neighbor-dict values.  The neighbor-dict is keyed by neighbor
301: (8)                    to the edgekey-dict.  So `G.adj[3][2][0]['color'] = 'blue'` sets
302: (8)                    the color of the edge `(3, 2, 0)` to `"blue"`.
303: (8)                    Iterating over G.adj behaves like a dict. Useful idioms include
304: (8)                    `for nbr, datadict in G.adj[n].items():`.
305: (8)                    The neighbor information is also provided by subscripting the graph.
306: (8)                    So `for nbr, foovalue in G[node].data('foo', default=1):` works.
307: (8)                    For directed graphs, `G.succ` is identical to `G.adj`.
308: (8)                    """
309: (8)                    return MultiAdjacencyView(self._succ)
310: (4)                @cached_property
311: (4)                def pred(self):
312: (8)                    """Graph adjacency object holding the predecessors of each node.
313: (8)                    This object is a read-only dict-like structure with node keys
314: (8)                    and neighbor-dict values.  The neighbor-dict is keyed by neighbor
315: (8)                    to the edgekey-dict.  So `G.adj[3][2][0]['color'] = 'blue'` sets
316: (8)                    the color of the edge `(3, 2, 0)` to `"blue"`.
317: (8)                    Iterating over G.adj behaves like a dict. Useful idioms include
318: (8)                    `for nbr, datadict in G.adj[n].items():`.
319: (8)                    """
320: (8)                    return MultiAdjacencyView(self._pred)
321: (4)                def add_edge(self, u_for_edge, v_for_edge, key=None, **attr):
322: (8)                    """Add an edge between u and v.
323: (8)                    The nodes u and v will be automatically added if they are
324: (8)                    not already in the graph.
325: (8)                    Edge attributes can be specified with keywords or by directly
326: (8)                    accessing the edge's attribute dictionary. See examples below.
327: (8)                    Parameters
328: (8)                    ----------
329: (8)                    u_for_edge, v_for_edge : nodes
330: (12)                       Nodes can be, for example, strings or numbers.
331: (12)                       Nodes must be hashable (and not None) Python objects.
```

```
        key : hashable identifier, optional (default=lowest unused integer)
            Used to distinguish multiedges between a pair of nodes.
        attr : keyword arguments, optional
            Edge data (or labels or objects) can be assigned using
            keyword arguments.
        Returns
        -------
        The edge key assigned to the edge.
        See Also
        --------
        add_edges_from : add a collection of edges
        Notes
        -----
        To replace/update edge data, use the optional key argument
        to identify a unique edge.  Otherwise a new edge will be created.
        NetworkX algorithms designed for weighted graphs cannot use
        multigraphs directly because it is not clear how to handle
        multiedge weights.  Convert to Graph using edge attribute
        'weight' to enable weighted graph algorithms.
        Default keys are generated using the method `new_edge_key()`.
        This method can be overridden by subclassing the base class and
        providing a custom `new_edge_key()` method.
        Examples
        --------
        The following all add the edge e=(1, 2) to graph G:
        >>> G = nx.MultiDiGraph()
        >>> e = (1, 2)
        >>> key = G.add_edge(1, 2)  # explicit two-node form
        >>> G.add_edge(*e)  # single edge as tuple of two nodes
        1
        >>> G.add_edges_from([(1, 2)])  # add edges from iterable container
        [2]
        Associate data to edges using keywords:
        >>> key = G.add_edge(1, 2, weight=3)
        >>> key = G.add_edge(1, 2, key=0, weight=4)  # update data for key=0
        >>> key = G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
        For non-string attribute keys, use subscript notation.
        >>> ekey = G.add_edge(1, 2)
        >>> G[1][2][0].update({0: 5})
        >>> G.edges[1, 2, 0].update({0: 5})
        """
        u, v = u_for_edge, v_for_edge
        if u not in self._succ:
            if u is None:
                raise ValueError("None cannot be a node")
            self._succ[u] = self.adjlist_inner_dict_factory()
            self._pred[u] = self.adjlist_inner_dict_factory()
            self._node[u] = self.node_attr_dict_factory()
        if v not in self._succ:
            if v is None:
                raise ValueError("None cannot be a node")
            self._succ[v] = self.adjlist_inner_dict_factory()
            self._pred[v] = self.adjlist_inner_dict_factory()
            self._node[v] = self.node_attr_dict_factory()
        if key is None:
            key = self.new_edge_key(u, v)
        if v in self._succ[u]:
            keydict = self._adj[u][v]
            datadict = keydict.get(key, self.edge_attr_dict_factory())
            datadict.update(attr)
            keydict[key] = datadict
        else:
            datadict = self.edge_attr_dict_factory()
            datadict.update(attr)
            keydict = self.edge_key_dict_factory()
            keydict[key] = datadict
            self._succ[u][v] = keydict
            self._pred[v][u] = keydict
        nx._clear_cache(self)
```

```
(8)                 return key
(4)             def remove_edge(self, u, v, key=None):
(8)                 """Remove an edge between u and v.
(8)                 Parameters
(8)                 ----------
(8)                 u, v : nodes
(12)                    Remove an edge between nodes u and v.
(8)                 key : hashable identifier, optional (default=None)
(12)                    Used to distinguish multiple edges between a pair of nodes.
(12)                    If None, remove a single edge between u and v. If there are
(12)                    multiple edges, removes the last edge added in terms of
(12)                    insertion order.
(8)                 Raises
(8)                 ------
(8)                 NetworkXError
(12)                    If there is not an edge between u and v, or
(12)                    if there is no edge with the specified key.
(8)                 See Also
(8)                 --------
(8)                 remove_edges_from : remove a collection of edges
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.MultiDiGraph()
(8)                 >>> nx.add_path(G, [0, 1, 2, 3])
(8)                 >>> G.remove_edge(0, 1)
(8)                 >>> e = (1, 2)
(8)                 >>> G.remove_edge(*e)  # unpacks e from an edge tuple
(8)                 For multiple edges
(8)                 >>> G = nx.MultiDiGraph()
(8)                 >>> G.add_edges_from([(1, 2), (1, 2), (1, 2)])  # key_list returned
(8)                 [0, 1, 2]
(8)                 When ``key=None`` (the default), edges are removed in the opposite
(8)                 order that they were added:
(8)                 >>> G.remove_edge(1, 2)
(8)                 >>> G.edges(keys=True)
(8)                 OutMultiEdgeView([(1, 2, 0), (1, 2, 1)])
(8)                 For edges with keys
(8)                 >>> G = nx.MultiDiGraph()
(8)                 >>> G.add_edge(1, 2, key="first")
(8)                 'first'
(8)                 >>> G.add_edge(1, 2, key="second")
(8)                 'second'
(8)                 >>> G.remove_edge(1, 2, key="first")
(8)                 >>> G.edges(keys=True)
(8)                 OutMultiEdgeView([(1, 2, 'second')])
(8)                 """
(8)                 try:
(12)                    d = self._adj[u][v]
(8)                 except KeyError as err:
(12)                    raise NetworkXError(f"The edge {u}-{v} is not in the graph.") from
err
(8)                 if key is None:
(12)                    d.popitem()
(8)                 else:
(12)                    try:
(16)                        del d[key]
(12)                    except KeyError as err:
(16)                        msg = f"The edge {u}-{v} with key {key} is not in the graph."
(16)                        raise NetworkXError(msg) from err
(8)                 if len(d) == 0:
(12)                    del self._succ[u][v]
(12)                    del self._pred[v][u]
(8)                 nx._clear_cache(self)
(4)             @cached_property
(4)             def edges(self):
(8)                 """An OutMultiEdgeView of the Graph as G.edges or G.edges().
(8)                 edges(self, nbunch=None, data=False, keys=False, default=None)
(8)                 The OutMultiEdgeView provides set-like operations on the edge-tuples
(8)                 as well as edge attribute lookup. When called, it also provides
```

```
(8)                     an EdgeDataView object which allows control of access to edge
(8)                     attributes (but does not provide set-like operations).
(8)                     Hence, ``G.edges[u, v, k]['color']`` provides the value of the color
(8)                     attribute for the edge from ``u`` to ``v`` with key ``k`` while
(8)                     ``for (u, v, k, c) in G.edges(data='color', default='red',
keys=True):``
(8)                     iterates through all the edges yielding the color attribute with
(8)                     default `'red'` if no color attribute exists.
(8)                     Edges are returned as tuples with optional data and keys
(8)                     in the order (node, neighbor, key, data). If ``keys=True`` is not
(8)                     provided, the tuples will just be (node, neighbor, data), but
(8)                     multiple tuples with the same node and neighbor will be
(8)                     generated when multiple edges between two nodes exist.
(8)                     Parameters
(8)                     ----------
(8)                     nbunch : single node, container, or all nodes (default= all nodes)
(12)                        The view will only report edges from these nodes.
(8)                     data : string or bool, optional (default=False)
(12)                        The edge attribute returned in 3-tuple (u, v, ddict[data]).
(12)                        If True, return edge attribute dict in 3-tuple (u, v, ddict).
(12)                        If False, return 2-tuple (u, v).
(8)                     keys : bool, optional (default=False)
(12)                        If True, return edge keys with each edge, creating (u, v, k,
(12)                        d) tuples when data is also requested (the default) and (u,
(12)                        v, k) tuples when data is not requested.
(8)                     default : value, optional (default=None)
(12)                        Value used for edges that don't have the requested attribute.
(12)                        Only relevant if data is not True or False.
(8)                     Returns
(8)                     -------
(8)                     edges : OutMultiEdgeView
(12)                        A view of edge attributes, usually it iterates over (u, v)
(12)                        (u, v, k) or (u, v, k, d) tuples of edges, but can also be
(12)                        used for attribute lookup as ``edges[u, v, k]['foo']``.
(8)                     Notes
(8)                     -----
(8)                     Nodes in nbunch that are not in the graph will be (quietly) ignored.
(8)                     For directed graphs this returns the out-edges.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.MultiDiGraph()
(8)                     >>> nx.add_path(G, [0, 1, 2])
(8)                     >>> key = G.add_edge(2, 3, weight=5)
(8)                     >>> key2 = G.add_edge(1, 2)  # second edge between these nodes
(8)                     >>> [e for e in G.edges()]
(8)                     [(0, 1), (1, 2), (1, 2), (2, 3)]
(8)                     >>> list(G.edges(data=True))  # default data is {} (empty dict)
(8)                     [(0, 1, {}), (1, 2, {}), (1, 2, {}), (2, 3, {'weight': 5})]
(8)                     >>> list(G.edges(data="weight", default=1))
(8)                     [(0, 1, 1), (1, 2, 1), (1, 2, 1), (2, 3, 5)]
(8)                     >>> list(G.edges(keys=True))  # default keys are integers
(8)                     [(0, 1, 0), (1, 2, 0), (1, 2, 1), (2, 3, 0)]
(8)                     >>> list(G.edges(data=True, keys=True))
(8)                     [(0, 1, 0, {}), (1, 2, 0, {}), (1, 2, 1, {}), (2, 3, 0, {'weight':
5})]
(8)                     >>> list(G.edges(data="weight", default=1, keys=True))
(8)                     [(0, 1, 0, 1), (1, 2, 0, 1), (1, 2, 1, 1), (2, 3, 0, 5)]
(8)                     >>> list(G.edges([0, 2]))
(8)                     [(0, 1), (2, 3)]
(8)                     >>> list(G.edges(0))
(8)                     [(0, 1)]
(8)                     >>> list(G.edges(1))
(8)                     [(1, 2), (1, 2)]
(8)                     See Also
(8)                     --------
(8)                     in_edges, out_edges
(8)                     """
(8)                     return OutMultiEdgeView(self)
(4)                 @cached_property
```

```
    def out_edges(self):
        return OutMultiEdgeView(self)
    out_edges.__doc__ = edges.__doc__
    @cached_property
    def in_edges(self):
        """A view of the in edges of the graph as G.in_edges or G.in_edges().
        in_edges(self, nbunch=None, data=False, keys=False, default=None)
        Parameters
        ----------
        nbunch : single node, container, or all nodes (default= all nodes)
            The view will only report edges incident to these nodes.
        data : string or bool, optional (default=False)
            The edge attribute returned in 3-tuple (u, v, ddict[data]).
            If True, return edge attribute dict in 3-tuple (u, v, ddict).
            If False, return 2-tuple (u, v).
        keys : bool, optional (default=False)
            If True, return edge keys with each edge, creating 3-tuples
            (u, v, k) or with data, 4-tuples (u, v, k, d).
        default : value, optional (default=None)
            Value used for edges that don't have the requested attribute.
            Only relevant if data is not True or False.
        Returns
        -------
        in_edges : InMultiEdgeView or InMultiEdgeDataView
            A view of edge attributes, usually it iterates over (u, v)
            or (u, v, k) or (u, v, k, d) tuples of edges, but can also be
            used for attribute lookup as `edges[u, v, k]['foo']`.
        See Also
        --------
        edges
        """
        return InMultiEdgeView(self)
    @cached_property
    def degree(self):
        """A DegreeView for the Graph as G.degree or G.degree().
        The node degree is the number of edges adjacent to the node.
        The weighted node degree is the sum of the edge weights for
        edges incident to that node.
        This object provides an iterator for (node, degree) as well as
        lookup for the degree for a single node.
        Parameters
        ----------
        nbunch : single node, container, or all nodes (default= all nodes)
            The view will only report edges incident to these nodes.
        weight : string or None, optional (default=None)
           The name of an edge attribute that holds the numerical value used
           as a weight.  If None, then each edge has weight 1.
           The degree is the sum of the edge weights adjacent to the node.
        Returns
        -------
        DiMultiDegreeView or int
            If multiple nodes are requested (the default), returns a `DiMultiDegreeView`
            mapping nodes to their degree.
            If a single node is requested, returns the degree of the node as an integer.
        See Also
        --------
        out_degree, in_degree
        Examples
        --------
        >>> G = nx.MultiDiGraph()
        >>> nx.add_path(G, [0, 1, 2, 3])
        >>> G.degree(0)  # node 0 with degree 1
        1
        >>> list(G.degree([0, 1, 2]))
        [(0, 1), (1, 2), (2, 2)]
        >>> G.add_edge(0, 1)  # parallel edge
        1
```

```
603: (8)                >>> list(G.degree([0, 1, 2]))  # parallel edges are counted
604: (8)                [(0, 2), (1, 3), (2, 2)]
605: (8)                """
606: (8)                return DiMultiDegreeView(self)
607: (4)            @cached_property
608: (4)            def in_degree(self):
609: (8)                """A DegreeView for (node, in_degree) or in_degree for single node.
610: (8)                The node in-degree is the number of edges pointing into the node.
611: (8)                The weighted node degree is the sum of the edge weights for
612: (8)                edges incident to that node.
613: (8)                This object provides an iterator for (node, degree) as well as
614: (8)                lookup for the degree for a single node.
615: (8)                Parameters
616: (8)                ----------
617: (8)                nbunch : single node, container, or all nodes (default= all nodes)
618: (12)                   The view will only report edges incident to these nodes.
619: (8)                weight : string or None, optional (default=None)
620: (11)                  The edge attribute that holds the numerical value used
621: (11)                  as a weight.  If None, then each edge has weight 1.
622: (11)                  The degree is the sum of the edge weights adjacent to the node.
623: (8)                Returns
624: (8)                -------
625: (8)                If a single node is requested
626: (8)                deg : int
627: (12)                   Degree of the node
628: (8)                OR if multiple nodes are requested
629: (8)                nd_iter : iterator
630: (12)                   The iterator returns two-tuples of (node, in-degree).
631: (8)                See Also
632: (8)                --------
633: (8)                degree, out_degree
634: (8)                Examples
635: (8)                --------
636: (8)                >>> G = nx.MultiDiGraph()
637: (8)                >>> nx.add_path(G, [0, 1, 2, 3])
638: (8)                >>> G.in_degree(0)  # node 0 with degree 0
639: (8)                0
640: (8)                >>> list(G.in_degree([0, 1, 2]))
641: (8)                [(0, 0), (1, 1), (2, 1)]
642: (8)                >>> G.add_edge(0, 1)  # parallel edge
643: (8)                1
644: (8)                >>> list(G.in_degree([0, 1, 2]))  # parallel edges counted
645: (8)                [(0, 0), (1, 2), (2, 1)]
646: (8)                """
647: (8)                return InMultiDegreeView(self)
648: (4)            @cached_property
649: (4)            def out_degree(self):
650: (8)                """Returns an iterator for (node, out-degree) or out-degree for single
node.
651: (8)                out_degree(self, nbunch=None, weight=None)
652: (8)                The node out-degree is the number of edges pointing out of the node.
653: (8)                This function returns the out-degree for a single node or an iterator
654: (8)                for a bunch of nodes or if nothing is passed as argument.
655: (8)                Parameters
656: (8)                ----------
657: (8)                nbunch : single node, container, or all nodes (default= all nodes)
658: (12)                   The view will only report edges incident to these nodes.
659: (8)                weight : string or None, optional (default=None)
660: (11)                  The edge attribute that holds the numerical value used
661: (11)                  as a weight.  If None, then each edge has weight 1.
662: (11)                  The degree is the sum of the edge weights.
663: (8)                Returns
664: (8)                -------
665: (8)                If a single node is requested
666: (8)                deg : int
667: (12)                   Degree of the node
668: (8)                OR if multiple nodes are requested
669: (8)                nd_iter : iterator
670: (12)                   The iterator returns two-tuples of (node, out-degree).
```

```
(8)                 See Also
(8)                 --------
(8)                 degree, in_degree
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.MultiDiGraph()
(8)                 >>> nx.add_path(G, [0, 1, 2, 3])
(8)                 >>> G.out_degree(0)  # node 0 with degree 1
(8)                 1
(8)                 >>> list(G.out_degree([0, 1, 2]))
(8)                 [(0, 1), (1, 1), (2, 1)]
(8)                 >>> G.add_edge(0, 1)  # parallel edge
(8)                 1
(8)                 >>> list(G.out_degree([0, 1, 2]))  # counts parallel edges
(8)                 [(0, 2), (1, 1), (2, 1)]
(8)                 """
(8)                 return OutMultiDegreeView(self)
(4)             def is_multigraph(self):
(8)                 """Returns True if graph is a multigraph, False otherwise."""
(8)                 return True
(4)             def is_directed(self):
(8)                 """Returns True if graph is directed, False otherwise."""
(8)                 return True
(4)             def to_undirected(self, reciprocal=False, as_view=False):
(8)                 """Returns an undirected representation of the digraph.
(8)                 Parameters
(8)                 ----------
(8)                 reciprocal : bool (optional)
(10)                  If True only keep edges that appear in both directions
(10)                  in the original digraph.
(8)                 as_view : bool (optional, default=False)
(10)                  If True return an undirected view of the original directed graph.
(8)                 Returns
(8)                 -------
(8)                 G : MultiGraph
(12)                    An undirected graph with the same name and nodes and
(12)                    with edge (u, v, data) if either (u, v, data) or (v, u, data)
(12)                    is in the digraph.  If both edges exist in digraph and
(12)                    their edge data is different, only one edge is created
(12)                    with an arbitrary choice of which edge data to use.
(12)                    You must check and correct for this manually if desired.
(8)                 See Also
(8)                 --------
(8)                 MultiGraph, copy, add_edge, add_edges_from
(8)                 Notes
(8)                 -----
(8)                 This returns a "deepcopy" of the edge, node, and
(8)                 graph attributes which attempts to completely copy
(8)                 all of the data and references.
(8)                 This is in contrast to the similar D=MultiDiGraph(G) which
(8)                 returns a shallow copy of the data.
(8)                 See the Python copy module for more information on shallow
(8)                 and deep copies, https://docs.python.org/3/library/copy.html.
(8)                 Warning: If you have subclassed MultiDiGraph to use dict-like
(8)                 objects in the data structure, those changes do not transfer
(8)                 to the MultiGraph created by this method.
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.path_graph(2)  # or MultiGraph, etc
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1), (1, 0)]
(8)                 >>> G2 = H.to_undirected()
(8)                 >>> list(G2.edges)
(8)                 [(0, 1)]
(8)                 """
(8)                 graph_class = self.to_undirected_class()
(8)                 if as_view is True:
(12)                    return nx.graphviews.generic_graph_view(self, graph_class)
```

```
740: (8)                    G = graph_class()
741: (8)                    G.graph.update(deepcopy(self.graph))
742: (8)                    G.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
743: (8)                    if reciprocal is True:
744: (12)                       G.add_edges_from(
745: (16)                           (u, v, key, deepcopy(data))
746: (16)                           for u, nbrs in self._adj.items()
747: (16)                           for v, keydict in nbrs.items()
748: (16)                           for key, data in keydict.items()
749: (16)                           if v in self._pred[u] and key in self._pred[u][v]
750: (12)                       )
751: (8)                    else:
752: (12)                       G.add_edges_from(
753: (16)                           (u, v, key, deepcopy(data))
754: (16)                           for u, nbrs in self._adj.items()
755: (16)                           for v, keydict in nbrs.items()
756: (16)                           for key, data in keydict.items()
757: (12)                       )
758: (8)                    return G
759: (4)                def reverse(self, copy=True):
760: (8)                    """Returns the reverse of the graph.
761: (8)                    The reverse is a graph with the same nodes and edges
762: (8)                    but with the directions of the edges reversed.
763: (8)                    Parameters
764: (8)                    ----------
765: (8)                    copy : bool optional (default=True)
766: (12)                       If True, return a new DiGraph holding the reversed edges.
767: (12)                       If False, the reverse graph is created using a view of
768: (12)                       the original graph.
769: (8)                    """
770: (8)                    if copy:
771: (12)                       H = self.__class__()
772: (12)                       H.graph.update(deepcopy(self.graph))
773: (12)                       H.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
774: (12)                       H.add_edges_from(
775: (16)                           (v, u, k, deepcopy(d))
776: (16)                           for u, v, k, d in self.edges(keys=True, data=True)
777: (12)                       )
778: (12)                       return H
779: (8)                    return nx.reverse_view(self)
```

File 396 - multigraph.py:

```
1: (0)              """Base class for MultiGraph."""
2: (0)              from copy import deepcopy
3: (0)              from functools import cached_property
4: (0)              import networkx as nx
5: (0)              from networkx import NetworkXError, convert
6: (0)              from networkx.classes.coreviews import MultiAdjacencyView
7: (0)              from networkx.classes.graph import Graph
8: (0)              from networkx.classes.reportviews import MultiDegreeView, MultiEdgeView
9: (0)              __all__ = ["MultiGraph"]
10: (0)             class MultiGraph(Graph):
11: (4)                 """
12: (4)                 An undirected graph class that can store multiedges.
13: (4)                 Multiedges are multiple edges between two nodes.  Each edge
14: (4)                 can hold optional data or attributes.
15: (4)                 A MultiGraph holds undirected edges.  Self loops are allowed.
16: (4)                 Nodes can be arbitrary (hashable) Python objects with optional
17: (4)                 key/value attributes. By convention `None` is not used as a node.
18: (4)                 Edges are represented as links between nodes with optional
19: (4)                 key/value attributes, in a MultiGraph each edge has a key to
20: (4)                 distinguish between multiple edges that have the same source and
21: (4)                 destination nodes.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 incoming_graph_data : input graph (optional, default: None)
```

```
(8)                     Data to initialize graph. If None (default) an empty
(8)                     graph is created.  The data can be any format that is supported
(8)                     by the to_networkx_graph() function, currently including edge list,
(8)                     dict of dicts, dict of lists, NetworkX graph, 2D NumPy array,
(8)                     SciPy sparse array, or PyGraphviz graph.
(4)                 multigraph_input : bool or None (default None)
(8)                     Note: Only used when `incoming_graph_data` is a dict.
(8)                     If True, `incoming_graph_data` is assumed to be a
(8)                     dict-of-dict-of-dict-of-dict structure keyed by
(8)                     node to neighbor to edge keys to edge data for multi-edges.
(8)                     A NetworkXError is raised if this is not the case.
(8)                     If False, :func:`to_networkx_graph` is used to try to determine
(8)                     the dict's graph data structure as either a dict-of-dict-of-dict
(8)                     keyed by node to neighbor to edge data, or a dict-of-iterable
(8)                     keyed by node to neighbors.
(8)                     If None, the treatment for True is tried, but if it fails,
(8)                     the treatment for False is tried.
(4)                 attr : keyword arguments, optional (default= no attributes)
(8)                     Attributes to add to graph as key=value pairs.
(4)                 See Also
(4)                 --------
(4)                 Graph
(4)                 DiGraph
(4)                 MultiDiGraph
(4)                 Examples
(4)                 --------
(4)                 Create an empty graph structure (a "null graph") with no nodes and
(4)                 no edges.
(4)                 >>> G = nx.MultiGraph()
(4)                 G can be grown in several ways.
(4)                 **Nodes:**
(4)                 Add one node at a time:
(4)                 >>> G.add_node(1)
(4)                 Add the nodes from any container (a list, dict, set or
(4)                 even the lines from a file or the nodes from another graph).
(4)                 >>> G.add_nodes_from([2, 3])
(4)                 >>> G.add_nodes_from(range(100, 110))
(4)                 >>> H = nx.path_graph(10)
(4)                 >>> G.add_nodes_from(H)
(4)                 In addition to strings and integers any hashable Python object
(4)                 (except None) can represent a node, e.g. a customized node object,
(4)                 or even another Graph.
(4)                 >>> G.add_node(H)
(4)                 **Edges:**
(4)                 G can also be grown by adding edges.
(4)                 Add one edge,
(4)                 >>> key = G.add_edge(1, 2)
(4)                 a list of edges,
(4)                 >>> keys = G.add_edges_from([(1, 2), (1, 3)])
(4)                 or a collection of edges,
(4)                 >>> keys = G.add_edges_from(H.edges)
(4)                 If some edges connect nodes not yet in the graph, the nodes
(4)                 are added automatically.  If an edge already exists, an additional
(4)                 edge is created and stored using a key to identify the edge.
(4)                 By default the key is the lowest unused integer.
(4)                 >>> keys = G.add_edges_from([(4, 5, {"route": 28}), (4, 5, {"route":
37})])
(4)                 >>> G[4]
(4)                 AdjacencyView({3: {0: {}}, 5: {0: {}, 1: {'route': 28}, 2: {'route':
37}}})
(4)                 **Attributes:**
(4)                 Each graph, node, and edge can hold key/value attribute pairs
(4)                 in an associated attribute dictionary (the keys must be hashable).
(4)                 By default these are empty, but can be added or changed using
(4)                 add_edge, add_node or direct manipulation of the attribute
(4)                 dictionaries named graph, node and edge respectively.
(4)                 >>> G = nx.MultiGraph(day="Friday")
(4)                 >>> G.graph
(4)                 {'day': 'Friday'}
```

```
(4)                 Add node attributes using add_node(), add_nodes_from() or G.nodes
(4)                 >>> G.add_node(1, time="5pm")
(4)                 >>> G.add_nodes_from([3], time="2pm")
(4)                 >>> G.nodes[1]
(4)                 {'time': '5pm'}
(4)                 >>> G.nodes[1]["room"] = 714
(4)                 >>> del G.nodes[1]["room"]  # remove attribute
(4)                 >>> list(G.nodes(data=True))
(4)                 [(1, {'time': '5pm'}), (3, {'time': '2pm'})]
(4)                 Add edge attributes using add_edge(), add_edges_from(), subscript
(4)                 notation, or G.edges.
(4)                 >>> key = G.add_edge(1, 2, weight=4.7)
(4)                 >>> keys = G.add_edges_from([(3, 4), (4, 5)], color="red")
(4)                 >>> keys = G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight":
8})])
(4)                 >>> G[1][2][0]["weight"] = 4.7
(4)                 >>> G.edges[1, 2, 0]["weight"] = 4
(4)                 Warning: we protect the graph data structure by making `G.edges[1,
(4)                 2, 0]` a read-only dict-like structure. However, you can assign to
(4)                 attributes in e.g. `G.edges[1, 2, 0]`. Thus, use 2 sets of brackets
(4)                 to add/change data attributes: `G.edges[1, 2, 0]['weight'] = 4`.
(4)                 **Shortcuts:**
(4)                 Many common graph features allow python syntax to speed reporting.
(4)                 >>> 1 in G  # check if node in graph
(4)                 True
(4)                 >>> [n for n in G if n < 3]  # iterate through nodes
(4)                 [1, 2]
(4)                 >>> len(G)  # number of nodes in graph
(4)                 5
(4)                 >>> G[1]  # adjacency dict-like view mapping neighbor -> edge key -> edge
attributes
(4)                 AdjacencyView({2: {0: {'weight': 4}, 1: {'color': 'blue'}}})
(4)                 Often the best way to traverse all edges of a graph is via the neighbors.
(4)                 The neighbors are reported as an adjacency-dict `G.adj` or
`G.adjacency()`.
(4)                 >>> for n, nbrsdict in G.adjacency():
(4)                 ...     for nbr, keydict in nbrsdict.items():
(4)                 ...         for key, eattr in keydict.items():
(4)                 ...             if "weight" in eattr:
(4)                 ...                 # Do something useful with the edges
(4)                 ...                 pass
(4)                 But the edges() method is often more convenient:
(4)                 >>> for u, v, keys, weight in G.edges(data="weight", keys=True):
(4)                 ...     if weight is not None:
(4)                 ...         # Do something useful with the edges
(4)                 ...         pass
(4)                 **Reporting:**
(4)                 Simple graph information is obtained using methods and object-attributes.
(4)                 Reporting usually provides views instead of containers to reduce memory
(4)                 usage. The views update as the graph is updated similarly to dict-views.
(4)                 The objects `nodes`, `edges` and `adj` provide access to data attributes
(4)                 via lookup (e.g. `nodes[n]`, `edges[u, v, k]`, `adj[u][v]`) and iteration
(4)                 (e.g. `nodes.items()`, `nodes.data('color')`,
(4)                 `nodes.data('color', default='blue')` and similarly for `edges`)
(4)                 Views exist for `nodes`, `edges`, `neighbors()`/`adj` and `degree`.
(4)                 For details on these and other miscellaneous methods, see below.
(4)                 **Subclasses (Advanced):**
(4)                 The MultiGraph class uses a dict-of-dict-of-dict-of-dict data structure.
(4)                 The outer dict (node_dict) holds adjacency information keyed by node.
(4)                 The next dict (adjlist_dict) represents the adjacency information
(4)                 and holds edge_key dicts keyed by neighbor. The edge_key dict holds
(4)                 each edge_attr dict keyed by edge key. The inner dict
(4)                 (edge_attr_dict) represents the edge data and holds edge attribute
(4)                 values keyed by attribute names.
(4)                 Each of these four dicts in the dict-of-dict-of-dict-of-dict
(4)                 structure can be replaced by a user defined dict-like object.
(4)                 In general, the dict-like features should be maintained but
(4)                 extra features can be added. To replace one of the dicts create
(4)                 a new graph class by changing the class(!) variable holding the
```

```
    factory for that dict-like structure. The variable names are
    node_dict_factory, node_attr_dict_factory, adjlist_inner_dict_factory,
    adjlist_outer_dict_factory, edge_key_dict_factory, edge_attr_dict_factory
    and graph_attr_dict_factory.
    node_dict_factory : function, (default: dict)
        Factory function to be used to create the dict containing node
        attributes, keyed by node id.
        It should require no arguments and return a dict-like object
    node_attr_dict_factory: function, (default: dict)
        Factory function to be used to create the node attribute
        dict which holds attribute values keyed by attribute name.
        It should require no arguments and return a dict-like object
    adjlist_outer_dict_factory : function, (default: dict)
        Factory function to be used to create the outer-most dict
        in the data structure that holds adjacency info keyed by node.
        It should require no arguments and return a dict-like object.
    adjlist_inner_dict_factory : function, (default: dict)
        Factory function to be used to create the adjacency list
        dict which holds multiedge key dicts keyed by neighbor.
        It should require no arguments and return a dict-like object.
    edge_key_dict_factory : function, (default: dict)
        Factory function to be used to create the edge key dict
        which holds edge data keyed by edge key.
        It should require no arguments and return a dict-like object.
    edge_attr_dict_factory : function, (default: dict)
        Factory function to be used to create the edge attribute
        dict which holds attribute values keyed by attribute name.
        It should require no arguments and return a dict-like object.
    graph_attr_dict_factory : function, (default: dict)
        Factory function to be used to create the graph attribute
        dict which holds attribute values keyed by attribute name.
        It should require no arguments and return a dict-like object.
    Typically, if your extension doesn't impact the data structure all
    methods will inherited without issue except: `to_directed/to_undirected`.
    By default these methods create a DiGraph/Graph class and you probably
    want them to create your extension of a DiGraph/Graph. To facilitate
    this we define two class variables that you can set in your subclass.
    to_directed_class : callable, (default: DiGraph or MultiDiGraph)
        Class to create a new graph structure in the `to_directed` method.
        If `None`, a NetworkX class (DiGraph or MultiDiGraph) is used.
    to_undirected_class : callable, (default: Graph or MultiGraph)
        Class to create a new graph structure in the `to_undirected` method.
        If `None`, a NetworkX class (Graph or MultiGraph) is used.
    **Subclassing Example**
    Create a low memory graph class that effectively disallows edge
    attributes by using a single attribute dict for all edges.
    This reduces the memory used, but you lose edge attributes.
    >>> class ThinGraph(nx.Graph):
    ...     all_edge_dict = {"weight": 1}
    ...
    ...     def single_edge_dict(self):
    ...         return self.all_edge_dict
    ...
    ...     edge_attr_dict_factory = single_edge_dict
    >>> G = ThinGraph()
    >>> G.add_edge(2, 1)
    >>> G[2][1]
    {'weight': 1}
    >>> G.add_edge(2, 2)
    >>> G[2][1] is G[2][2]
    True
    """
    edge_key_dict_factory = dict
    def to_directed_class(self):
        """Returns the class to use for empty directed copies.
        If you subclass the base classes, use this to designate
        what directed class to use for `to_directed()` copies.
        """
        return nx.MultiDiGraph
```

```
    def to_undirected_class(self):
        """Returns the class to use for empty undirected copies.
        If you subclass the base classes, use this to designate
        what directed class to use for `to_directed()` copies.
        """
        return MultiGraph
    def __init__(self, incoming_graph_data=None, multigraph_input=None, **attr):
        """Initialize a graph with edges, name, or graph attributes.
        Parameters
        ----------
        incoming_graph_data : input graph
            Data to initialize graph.  If incoming_graph_data=None (default)
            an empty graph is created.  The data can be an edge list, or any
            NetworkX graph object.  If the corresponding optional Python
            packages are installed the data can also be a 2D NumPy array, a
            SciPy sparse array, or a PyGraphviz graph.
        multigraph_input : bool or None (default None)
            Note: Only used when `incoming_graph_data` is a dict.
            If True, `incoming_graph_data` is assumed to be a
            dict-of-dict-of-dict-of-dict structure keyed by
            node to neighbor to edge keys to edge data for multi-edges.
            A NetworkXError is raised if this is not the case.
            If False, :func:`to_networkx_graph` is used to try to determine
            the dict's graph data structure as either a dict-of-dict-of-dict
            keyed by node to neighbor to edge data, or a dict-of-iterable
            keyed by node to neighbors.
            If None, the treatment for True is tried, but if it fails,
            the treatment for False is tried.
        attr : keyword arguments, optional (default= no attributes)
            Attributes to add to graph as key=value pairs.
        See Also
        --------
        convert
        Examples
        --------
        >>> G = nx.MultiGraph()
        >>> G = nx.MultiGraph(name="my graph")
        >>> e = [(1, 2), (1, 2), (2, 3), (3, 4)]  # list of edges
        >>> G = nx.MultiGraph(e)
        Arbitrary graph attribute pairs (key=value) may be assigned
        >>> G = nx.MultiGraph(e, day="Friday")
        >>> G.graph
        {'day': 'Friday'}
        """
        if isinstance(incoming_graph_data, dict) and multigraph_input is not False:
            Graph.__init__(self)
            try:
                convert.from_dict_of_dicts(
                    incoming_graph_data, create_using=self, multigraph_input=True
                )
                self.graph.update(attr)
            except Exception as err:
                if multigraph_input is True:
                    raise nx.NetworkXError(
                        f"converting multigraph_input raised:\n{type(err)}: {err}"
                    )
                Graph.__init__(self, incoming_graph_data, **attr)
        else:
            Graph.__init__(self, incoming_graph_data, **attr)
    @cached_property
    def adj(self):
        """Graph adjacency object holding the neighbors of each node.
        This object is a read-only dict-like structure with node keys
        and neighbor-dict values.  The neighbor-dict is keyed by neighbor
        to the edgekey-data-dict.  So `G.adj[3][2][0]['color'] = 'blue'` sets
```

```
(8)                 the color of the edge `(3, 2, 0)` to `"blue"`.
(8)                 Iterating over G.adj behaves like a dict. Useful idioms include
(8)                 `for nbr, edgesdict in G.adj[n].items():`.
(8)                 The neighbor information is also provided by subscripting the graph.
(8)                 Examples
(8)                 --------
(8)                 >>> e = [(1, 2), (1, 2), (1, 3), (3, 4)]  # list of edges
(8)                 >>> G = nx.MultiGraph(e)
(8)                 >>> G.edges[1, 2, 0]["weight"] = 3
(8)                 >>> result = set()
(8)                 >>> for edgekey, data in G[1][2].items():
(8)                 ...     result.add(data.get("weight", 1))
(8)                 >>> result
(8)                 {1, 3}
(8)                 For directed graphs, `G.adj` holds outgoing (successor) info.
(8)                 """
(8)                 return MultiAdjacencyView(self._adj)
(4)             def new_edge_key(self, u, v):
(8)                 """Returns an unused key for edges between nodes `u` and `v`.
(8)                 The nodes `u` and `v` do not need to be already in the graph.
(8)                 Notes
(8)                 -----
(8)                 In the standard MultiGraph class the new key is the number of existing
(8)                 edges between `u` and `v` (increased if necessary to ensure unused).
(8)                 The first edge will have key 0, then 1, etc. If an edge is removed
(8)                 further new_edge_keys may not be in this order.
(8)                 Parameters
(8)                 ----------
(8)                 u, v : nodes
(8)                 Returns
(8)                 -------
(8)                 key : int
(8)                 """
(8)                 try:
(12)                    keydict = self._adj[u][v]
(8)                 except KeyError:
(12)                    return 0
(8)                 key = len(keydict)
(8)                 while key in keydict:
(12)                    key += 1
(8)                 return key
(4)             def add_edge(self, u_for_edge, v_for_edge, key=None, **attr):
(8)                 """Add an edge between u and v.
(8)                 The nodes u and v will be automatically added if they are
(8)                 not already in the graph.
(8)                 Edge attributes can be specified with keywords or by directly
(8)                 accessing the edge's attribute dictionary. See examples below.
(8)                 Parameters
(8)                 ----------
(8)                 u_for_edge, v_for_edge : nodes
(12)                    Nodes can be, for example, strings or numbers.
(12)                    Nodes must be hashable (and not None) Python objects.
(8)                 key : hashable identifier, optional (default=lowest unused integer)
(12)                    Used to distinguish multiedges between a pair of nodes.
(8)                 attr : keyword arguments, optional
(12)                    Edge data (or labels or objects) can be assigned using
(12)                    keyword arguments.
(8)                 Returns
(8)                 -------
(8)                 The edge key assigned to the edge.
(8)                 See Also
(8)                 --------
(8)                 add_edges_from : add a collection of edges
(8)                 Notes
(8)                 -----
(8)                 To replace/update edge data, use the optional key argument
(8)                 to identify a unique edge.  Otherwise a new edge will be created.
(8)                 NetworkX algorithms designed for weighted graphs cannot use
(8)                 multigraphs directly because it is not clear how to handle
```

```
(8)                     multiedge weights.  Convert to Graph using edge attribute
(8)                     'weight' to enable weighted graph algorithms.
(8)                     Default keys are generated using the method `new_edge_key()`.
(8)                     This method can be overridden by subclassing the base class and
(8)                     providing a custom `new_edge_key()` method.
(8)                     Examples
(8)                     --------
(8)                     The following each add an additional edge e=(1, 2) to graph G:
(8)                     >>> G = nx.MultiGraph()
(8)                     >>> e = (1, 2)
(8)                     >>> ekey = G.add_edge(1, 2)  # explicit two-node form
(8)                     >>> G.add_edge(*e)  # single edge as tuple of two nodes
(8)                     1
(8)                     >>> G.add_edges_from([(1, 2)])  # add edges from iterable container
(8)                     [2]
(8)                     Associate data to edges using keywords:
(8)                     >>> ekey = G.add_edge(1, 2, weight=3)
(8)                     >>> ekey = G.add_edge(1, 2, key=0, weight=4)  # update data for key=0
(8)                     >>> ekey = G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
(8)                     For non-string attribute keys, use subscript notation.
(8)                     >>> ekey = G.add_edge(1, 2)
(8)                     >>> G[1][2][0].update({0: 5})
(8)                     >>> G.edges[1, 2, 0].update({0: 5})
(8)                     """
(8)                     u, v = u_for_edge, v_for_edge
(8)                     if u not in self._adj:
(12)                        if u is None:
(16)                            raise ValueError("None cannot be a node")
(12)                        self._adj[u] = self.adjlist_inner_dict_factory()
(12)                        self._node[u] = self.node_attr_dict_factory()
(8)                     if v not in self._adj:
(12)                        if v is None:
(16)                            raise ValueError("None cannot be a node")
(12)                        self._adj[v] = self.adjlist_inner_dict_factory()
(12)                        self._node[v] = self.node_attr_dict_factory()
(8)                     if key is None:
(12)                        key = self.new_edge_key(u, v)
(8)                     if v in self._adj[u]:
(12)                        keydict = self._adj[u][v]
(12)                        datadict = keydict.get(key, self.edge_attr_dict_factory())
(12)                        datadict.update(attr)
(12)                        keydict[key] = datadict
(8)                     else:
(12)                        datadict = self.edge_attr_dict_factory()
(12)                        datadict.update(attr)
(12)                        keydict = self.edge_key_dict_factory()
(12)                        keydict[key] = datadict
(12)                        self._adj[u][v] = keydict
(12)                        self._adj[v][u] = keydict
(8)                     nx._clear_cache(self)
(8)                     return key
(4)                 def add_edges_from(self, ebunch_to_add, **attr):
(8)                     """Add all the edges in ebunch_to_add.
(8)                     Parameters
(8)                     ----------
(8)                     ebunch_to_add : container of edges
(12)                        Each edge given in the container will be added to the
(12)                        graph. The edges can be:
(16)                            - 2-tuples (u, v) or
(16)                            - 3-tuples (u, v, d) for an edge data dict d, or
(16)                            - 3-tuples (u, v, k) for not iterable key k, or
(16)                            - 4-tuples (u, v, k, d) for an edge with data and key k
(8)                     attr : keyword arguments, optional
(12)                        Edge data (or labels or objects) can be assigned using
(12)                        keyword arguments.
(8)                     Returns
(8)                     -------
(8)                     A list of edge keys assigned to the edges in `ebunch`.
(8)                     See Also
```

```
        --------
        add_edge : add a single edge
        add_weighted_edges_from : convenient way to add weighted edges
        Notes
        -----
        Adding the same edge twice has no effect but any edge data
        will be updated when each duplicate edge is added.
        Edge attributes specified in an ebunch take precedence over
        attributes specified via keyword arguments.
        Default keys are generated using the method ``new_edge_key()``.
        This method can be overridden by subclassing the base class and
        providing a custom ``new_edge_key()`` method.
        When adding edges from an iterator over the graph you are changing,
        a `RuntimeError` can be raised with message:
        `RuntimeError: dictionary changed size during iteration`. This
        happens when the graph's underlying dictionary is modified during
        iteration. To avoid this error, evaluate the iterator into a separate
        object, e.g. by using `list(iterator_of_edges)`, and pass this
        object to `G.add_edges_from`.
        Examples
        --------
        >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
        >>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
        >>> e = zip(range(0, 3), range(1, 4))
        >>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
        Associate data to edges
        >>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
        >>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
        Evaluate an iterator over a graph if using it to modify the same graph
        >>> G = nx.MultiGraph([(1, 2), (2, 3), (3, 4)])
        >>> # Grow graph by one new node, adding edges to all existing nodes.
        >>> # wrong way - will raise RuntimeError
        >>> # G.add_edges_from(((5, n) for n in G.nodes))
        >>> # right way - note that there will be no self-edge for node 5
        >>> assigned_keys = G.add_edges_from(list((5, n) for n in G.nodes))
        """
        keylist = []
        for e in ebunch_to_add:
            ne = len(e)
            if ne == 4:
                u, v, key, dd = e
            elif ne == 3:
                u, v, dd = e
                key = None
            elif ne == 2:
                u, v = e
                dd = {}
                key = None
            else:
                msg = f"Edge tuple {e} must be a 2-tuple, 3-tuple or 4-tuple."
                raise NetworkXError(msg)
            ddd = {}
            ddd.update(attr)
            try:
                ddd.update(dd)
            except (TypeError, ValueError):
                if ne != 3:
                    raise
                key = dd  # ne == 3 with 3rd value not dict, must be a key
            key = self.add_edge(u, v, key)
            self[u][v][key].update(ddd)
            keylist.append(key)
        nx._clear_cache(self)
        return keylist
    def remove_edge(self, u, v, key=None):
        """Remove an edge between u and v.
        Parameters
        ----------
        u, v : nodes
```

```
(12)                        Remove an edge between nodes u and v.
(8)                     key : hashable identifier, optional (default=None)
(12)                        Used to distinguish multiple edges between a pair of nodes.
(12)                        If None, remove a single edge between u and v. If there are
(12)                        multiple edges, removes the last edge added in terms of
(12)                        insertion order.
(8)                     Raises
(8)                     ------
(8)                     NetworkXError
(12)                        If there is not an edge between u and v, or
(12)                        if there is no edge with the specified key.
(8)                     See Also
(8)                     --------
(8)                     remove_edges_from : remove a collection of edges
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.MultiGraph()
(8)                     >>> nx.add_path(G, [0, 1, 2, 3])
(8)                     >>> G.remove_edge(0, 1)
(8)                     >>> e = (1, 2)
(8)                     >>> G.remove_edge(*e)  # unpacks e from an edge tuple
(8)                     For multiple edges
(8)                     >>> G = nx.MultiGraph()  # or MultiDiGraph, etc
(8)                     >>> G.add_edges_from([(1, 2), (1, 2), (1, 2)])  # key_list returned
(8)                     [0, 1, 2]
(8)                     When ``key=None`` (the default), edges are removed in the opposite
(8)                     order that they were added:
(8)                     >>> G.remove_edge(1, 2)
(8)                     >>> G.edges(keys=True)
(8)                     MultiEdgeView([(1, 2, 0), (1, 2, 1)])
(8)                     >>> G.remove_edge(2, 1)  # edges are not directed
(8)                     >>> G.edges(keys=True)
(8)                     MultiEdgeView([(1, 2, 0)])
(8)                     For edges with keys
(8)                     >>> G = nx.MultiGraph()
(8)                     >>> G.add_edge(1, 2, key="first")
(8)                     'first'
(8)                     >>> G.add_edge(1, 2, key="second")
(8)                     'second'
(8)                     >>> G.remove_edge(1, 2, key="first")
(8)                     >>> G.edges(keys=True)
(8)                     MultiEdgeView([(1, 2, 'second')])
(8)                     """
(8)                     try:
(12)                        d = self._adj[u][v]
(8)                     except KeyError as err:
(12)                        raise NetworkXError(f"The edge {u}-{v} is not in the graph.") from
err
(8)                     if key is None:
(12)                        d.popitem()
(8)                     else:
(12)                        try:
(16)                            del d[key]
(12)                        except KeyError as err:
(16)                            msg = f"The edge {u}-{v} with key {key} is not in the graph."
(16)                            raise NetworkXError(msg) from err
(8)                     if len(d) == 0:
(12)                        del self._adj[u][v]
(12)                        if u != v:  # check for selfloop
(16)                            del self._adj[v][u]
(8)                     nx._clear_cache(self)
(4)                 def remove_edges_from(self, ebunch):
(8)                     """Remove all edges specified in ebunch.
(8)                     Parameters
(8)                     ----------
(8)                     ebunch: list or container of edge tuples
(12)                        Each edge given in the list or container will be removed
(12)                        from the graph. The edges can be:
(16)                            - 2-tuples (u, v) A single edge between u and v is removed.
```

```
(16)                            - 3-tuples (u, v, key) The edge identified by key is removed.
(16)                            - 4-tuples (u, v, key, data) where data is ignored.
(8)                     See Also
(8)                     --------
(8)                     remove_edge : remove a single edge
(8)                     Notes
(8)                     -----
(8)                     Will fail silently if an edge in ebunch is not in the graph.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> ebunch = [(1, 2), (2, 3)]
(8)                     >>> G.remove_edges_from(ebunch)
(8)                     Removing multiple copies of edges
(8)                     >>> G = nx.MultiGraph()
(8)                     >>> keys = G.add_edges_from([(1, 2), (1, 2), (1, 2)])
(8)                     >>> G.remove_edges_from([(1, 2), (2, 1)])  # edges aren't directed
(8)                     >>> list(G.edges())
(8)                     [(1, 2)]
(8)                     >>> G.remove_edges_from([(1, 2), (1, 2)])  # silently ignore extra
copy
(8)                     >>> list(G.edges)  # now empty graph
(8)                     []
(8)                     When the edge is a 2-tuple ``(u, v)`` but there are multiple edges
between
(8)                     u and v in the graph, the most recent edge (in terms of insertion
(8)                     order) is removed.
(8)                     >>> G = nx.MultiGraph()
(8)                     >>> for key in ("x", "y", "a"):
(8)                     ...     k = G.add_edge(0, 1, key=key)
(8)                     >>> G.edges(keys=True)
(8)                     MultiEdgeView([(0, 1, 'x'), (0, 1, 'y'), (0, 1, 'a')])
(8)                     >>> G.remove_edges_from([(0, 1)])
(8)                     >>> G.edges(keys=True)
(8)                     MultiEdgeView([(0, 1, 'x'), (0, 1, 'y')])
(8)                     """
(8)                     for e in ebunch:
(12)                        try:
(16)                            self.remove_edge(*e[:3])
(12)                        except NetworkXError:
(16)                            pass
(8)                     nx._clear_cache(self)
(4)                 def has_edge(self, u, v, key=None):
(8)                     """Returns True if the graph has an edge between nodes u and v.
(8)                     This is the same as `v in G[u] or key in G[u][v]`
(8)                     without KeyError exceptions.
(8)                     Parameters
(8)                     ----------
(8)                     u, v : nodes
(12)                        Nodes can be, for example, strings or numbers.
(8)                     key : hashable identifier, optional (default=None)
(12)                        If specified return True only if the edge with
(12)                        key is found.
(8)                     Returns
(8)                     -------
(8)                     edge_ind : bool
(12)                        True if edge is in the graph, False otherwise.
(8)                     Examples
(8)                     --------
(8)                     Can be called either using two nodes u, v, an edge tuple (u, v),
(8)                     or an edge tuple (u, v, key).
(8)                     >>> G = nx.MultiGraph()  # or MultiDiGraph
(8)                     >>> nx.add_path(G, [0, 1, 2, 3])
(8)                     >>> G.has_edge(0, 1)  # using two nodes
(8)                     True
(8)                     >>> e = (0, 1)
(8)                     >>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
(8)                     True
(8)                     >>> G.add_edge(0, 1, key="a")
```

```
634: (8)                    'a'
635: (8)                    >>> G.has_edge(0, 1, key="a")  # specify key
636: (8)                    True
637: (8)                    >>> G.has_edge(1, 0, key="a")  # edges aren't directed
638: (8)                    True
639: (8)                    >>> e = (0, 1, "a")
640: (8)                    >>> G.has_edge(*e)  # e is a 3-tuple (u, v, 'a')
641: (8)                    True
642: (8)                    The following syntax are equivalent:
643: (8)                    >>> G.has_edge(0, 1)
644: (8)                    True
645: (8)                    >>> 1 in G[0]  # though this gives :exc:`KeyError` if 0 not in G
646: (8)                    True
647: (8)                    >>> 0 in G[1]  # other order; also gives :exc:`KeyError` if 0 not in G
648: (8)                    True
649: (8)                    """
650: (8)                    try:
651: (12)                       if key is None:
652: (16)                           return v in self._adj[u]
653: (12)                       else:
654: (16)                           return key in self._adj[u][v]
655: (8)                    except KeyError:
656: (12)                       return False
657: (4)                @cached_property
658: (4)                def edges(self):
659: (8)                    """Returns an iterator over the edges.
660: (8)                    edges(self, nbunch=None, data=False, keys=False, default=None)
661: (8)                    The MultiEdgeView provides set-like operations on the edge-tuples
662: (8)                    as well as edge attribute lookup. When called, it also provides
663: (8)                    an EdgeDataView object which allows control of access to edge
664: (8)                    attributes (but does not provide set-like operations).
665: (8)                    Hence, ``G.edges[u, v, k]['color']`` provides the value of the color
666: (8)                    attribute for the edge from ``u`` to ``v`` with key ``k`` while
667: (8)                    ``for (u, v, k, c) in G.edges(data='color', keys=True,
default="red"):``
668: (8)                    iterates through all the edges yielding the color attribute with
669: (8)                    default `'red'` if no color attribute exists.
670: (8)                    Edges are returned as tuples with optional data and keys
671: (8)                    in the order (node, neighbor, key, data). If ``keys=True`` is not
672: (8)                    provided, the tuples will just be (node, neighbor, data), but
673: (8)                    multiple tuples with the same node and neighbor will be generated
674: (8)                    when multiple edges exist between two nodes.
675: (8)                    Parameters
676: (8)                    ----------
677: (8)                    nbunch : single node, container, or all nodes (default= all nodes)
678: (12)                       The view will only report edges from these nodes.
679: (8)                    data : string or bool, optional (default=False)
680: (12)                       The edge attribute returned in 3-tuple (u, v, ddict[data]).
681: (12)                       If True, return edge attribute dict in 3-tuple (u, v, ddict).
682: (12)                       If False, return 2-tuple (u, v).
683: (8)                    keys : bool, optional (default=False)
684: (12)                       If True, return edge keys with each edge, creating (u, v, k)
685: (12)                       tuples or (u, v, k, d) tuples if data is also requested.
686: (8)                    default : value, optional (default=None)
687: (12)                       Value used for edges that don't have the requested attribute.
688: (12)                       Only relevant if data is not True or False.
689: (8)                    Returns
690: (8)                    -------
691: (8)                    edges : MultiEdgeView
692: (12)                       A view of edge attributes, usually it iterates over (u, v)
693: (12)                       (u, v, k) or (u, v, k, d) tuples of edges, but can also be
694: (12)                       used for attribute lookup as ``edges[u, v, k]['foo']``.
695: (8)                    Notes
696: (8)                    -----
697: (8)                    Nodes in nbunch that are not in the graph will be (quietly) ignored.
698: (8)                    For directed graphs this returns the out-edges.
699: (8)                    Examples
700: (8)                    --------
701: (8)                    >>> G = nx.MultiGraph()
```

```
(8)                     >>> nx.add_path(G, [0, 1, 2])
(8)                     >>> key = G.add_edge(2, 3, weight=5)
(8)                     >>> key2 = G.add_edge(2, 1, weight=2)  # multi-edge
(8)                     >>> [e for e in G.edges()]
(8)                     [(0, 1), (1, 2), (1, 2), (2, 3)]
(8)                     >>> G.edges.data()  # default data is {} (empty dict)
(8)                     MultiEdgeDataView([(0, 1, {}), (1, 2, {}), (1, 2, {'weight': 2}), (2,
3, {'weight': 5})])
(8)                     >>> G.edges.data("weight", default=1)
(8)                     MultiEdgeDataView([(0, 1, 1), (1, 2, 1), (1, 2, 2), (2, 3, 5)])
(8)                     >>> G.edges(keys=True)  # default keys are integers
(8)                     MultiEdgeView([(0, 1, 0), (1, 2, 0), (1, 2, 1), (2, 3, 0)])
(8)                     >>> G.edges.data(keys=True)
(8)                     MultiEdgeDataView([(0, 1, 0, {}), (1, 2, 0, {}), (1, 2, 1, {'weight':
2}), (2, 3, 0, {'weight': 5})])
(8)                     >>> G.edges.data("weight", default=1, keys=True)
(8)                     MultiEdgeDataView([(0, 1, 0, 1), (1, 2, 0, 1), (1, 2, 1, 2), (2, 3, 0,
5)])
(8)                     >>> G.edges([0, 3])  # Note ordering of tuples from listed sources
(8)                     MultiEdgeDataView([(0, 1), (3, 2)])
(8)                     >>> G.edges([0, 3, 2, 1])  # Note ordering of tuples
(8)                     MultiEdgeDataView([(0, 1), (3, 2), (2, 1), (2, 1)])
(8)                     >>> G.edges(0)
(8)                     MultiEdgeDataView([(0, 1)])
(8)                     """
(8)                     return MultiEdgeView(self)
(4)                 def get_edge_data(self, u, v, key=None, default=None):
(8)                     """Returns the attribute dictionary associated with edge (u, v,
(8)                     key).
(8)                     If a key is not provided, returns a dictionary mapping edge keys
(8)                     to attribute dictionaries for each edge between u and v.
(8)                     This is identical to `G[u][v][key]` except the default is returned
(8)                     instead of an exception is the edge doesn't exist.
(8)                     Parameters
(8)                     ----------
(8)                     u, v : nodes
(8)                     default :  any Python object (default=None)
(12)                        Value to return if the specific edge (u, v, key) is not
(12)                        found, OR if there are no edges between u and v and no key
(12)                        is specified.
(8)                     key : hashable identifier, optional (default=None)
(12)                        Return data only for the edge with specified key, as an
(12)                        attribute dictionary (rather than a dictionary mapping keys
(12)                        to attribute dictionaries).
(8)                     Returns
(8)                     -------
(8)                     edge_dict : dictionary
(12)                        The edge attribute dictionary, OR a dictionary mapping edge
(12)                        keys to attribute dictionaries for each of those edges if no
(12)                        specific key is provided (even if there's only one edge
(12)                        between u and v).
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.MultiGraph()  # or MultiDiGraph
(8)                     >>> key = G.add_edge(0, 1, key="a", weight=7)
(8)                     >>> G[0][1]["a"]  # key='a'
(8)                     {'weight': 7}
(8)                     >>> G.edges[0, 1, "a"]  # key='a'
(8)                     {'weight': 7}
(8)                     Warning: we protect the graph data structure by making
(8)                     `G.edges` and `G[1][2]` read-only dict-like structures.
(8)                     However, you can assign values to attributes in e.g.
(8)                     `G.edges[1, 2, 'a']` or `G[1][2]['a']` using an additional
(8)                     bracket as shown next. You need to specify all edge info
(8)                     to assign to the edge data associated with an edge.
(8)                     >>> G[0][1]["a"]["weight"] = 10
(8)                     >>> G.edges[0, 1, "a"]["weight"] = 10
(8)                     >>> G[0][1]["a"]["weight"]
(8)                     10
```

```
768: (8)                    >>> G.edges[1, 0, "a"]["weight"]
769: (8)                    10
770: (8)                    >>> G = nx.MultiGraph()  # or MultiDiGraph
771: (8)                    >>> nx.add_path(G, [0, 1, 2, 3])
772: (8)                    >>> G.edges[0, 1, 0]["weight"] = 5
773: (8)                    >>> G.get_edge_data(0, 1)
774: (8)                    {0: {'weight': 5}}
775: (8)                    >>> e = (0, 1)
776: (8)                    >>> G.get_edge_data(*e)  # tuple form
777: (8)                    {0: {'weight': 5}}
778: (8)                    >>> G.get_edge_data(3, 0)  # edge not in graph, returns None
779: (8)                    >>> G.get_edge_data(3, 0, default=0)  # edge not in graph, return
default
780: (8)                    0
781: (8)                    >>> G.get_edge_data(1, 0, 0)  # specific key gives back
782: (8)                    {'weight': 5}
783: (8)                    """
784: (8)                    try:
785: (12)                       if key is None:
786: (16)                           return self._adj[u][v]
787: (12)                       else:
788: (16)                           return self._adj[u][v][key]
789: (8)                    except KeyError:
790: (12)                       return default
791: (4)                @cached_property
792: (4)                def degree(self):
793: (8)                    """A DegreeView for the Graph as G.degree or G.degree().
794: (8)                    The node degree is the number of edges adjacent to the node.
795: (8)                    The weighted node degree is the sum of the edge weights for
796: (8)                    edges incident to that node.
797: (8)                    This object provides an iterator for (node, degree) as well as
798: (8)                    lookup for the degree for a single node.
799: (8)                    Parameters
800: (8)                    ----------
801: (8)                    nbunch : single node, container, or all nodes (default= all nodes)
802: (12)                       The view will only report edges incident to these nodes.
803: (8)                    weight : string or None, optional (default=None)
804: (11)                      The name of an edge attribute that holds the numerical value used
805: (11)                      as a weight.  If None, then each edge has weight 1.
806: (11)                      The degree is the sum of the edge weights adjacent to the node.
807: (8)                    Returns
808: (8)                    -------
809: (8)                    MultiDegreeView or int
810: (12)                       If multiple nodes are requested (the default), returns a
`MultiDegreeView`
811: (12)                       mapping nodes to their degree.
812: (12)                       If a single node is requested, returns the degree of the node as
an integer.
813: (8)                    Examples
814: (8)                    --------
815: (8)                    >>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
816: (8)                    >>> nx.add_path(G, [0, 1, 2, 3])
817: (8)                    >>> G.degree(0)  # node 0 with degree 1
818: (8)                    1
819: (8)                    >>> list(G.degree([0, 1]))
820: (8)                    [(0, 1), (1, 2)]
821: (8)                    """
822: (8)                    return MultiDegreeView(self)
823: (4)                def is_multigraph(self):
824: (8)                    """Returns True if graph is a multigraph, False otherwise."""
825: (8)                    return True
826: (4)                def is_directed(self):
827: (8)                    """Returns True if graph is directed, False otherwise."""
828: (8)                    return False
829: (4)                def copy(self, as_view=False):
830: (8)                    """Returns a copy of the graph.
831: (8)                    The copy method by default returns an independent shallow copy
832: (8)                    of the graph and attributes. That is, if an attribute is a
833: (8)                    container, that container is shared by the original an the copy.
```

```
(8)                     Use Python's `copy.deepcopy` for new containers.
(8)                     If `as_view` is True then a view is returned instead of a copy.
(8)                     Notes
(8)                     -----
(8)                     All copies reproduce the graph structure, but data attributes
(8)                     may be handled in different ways. There are four types of copies
(8)                     of a graph that people might want.
(8)                     Deepcopy -- A "deepcopy" copies the graph structure as well as
(8)                     all data attributes and any objects they might contain.
(8)                     The entire graph object is new so that changes in the copy
(8)                     do not affect the original object. (see Python's copy.deepcopy)
(8)                     Data Reference (Shallow) -- For a shallow copy the graph structure
(8)                     is copied but the edge, node and graph attribute dicts are
(8)                     references to those in the original graph. This saves
(8)                     time and memory but could cause confusion if you change an attribute
(8)                     in one graph and it changes the attribute in the other.
(8)                     NetworkX does not provide this level of shallow copy.
(8)                     Independent Shallow -- This copy creates new independent attribute
(8)                     dicts and then does a shallow copy of the attributes. That is, any
(8)                     attributes that are containers are shared between the new graph
(8)                     and the original. This is exactly what `dict.copy()` provides.
(8)                     You can obtain this style copy using:
(12)                        >>> G = nx.path_graph(5)
(12)                        >>> H = G.copy()
(12)                        >>> H = G.copy(as_view=False)
(12)                        >>> H = nx.Graph(G)
(12)                        >>> H = G.__class__(G)
(8)                     Fresh Data -- For fresh data, the graph structure is copied while
(8)                     new empty data attribute dicts are created. The resulting graph
(8)                     is independent of the original and it has no edge, node or graph
(8)                     attributes. Fresh copies are not enabled. Instead use:
(12)                        >>> H = G.__class__()
(12)                        >>> H.add_nodes_from(G)
(12)                        >>> H.add_edges_from(G.edges)
(8)                     View -- Inspired by dict-views, graph-views act like read-only
(8)                     versions of the original graph, providing a copy of the original
(8)                     structure without requiring any memory for copying the information.
(8)                     See the Python copy module for more information on shallow
(8)                     and deep copies, https://docs.python.org/3/library/copy.html.
(8)                     Parameters
(8)                     ----------
(8)                     as_view : bool, optional (default=False)
(12)                        If True, the returned graph-view provides a read-only view
(12)                        of the original graph without actually copying any data.
(8)                     Returns
(8)                     -------
(8)                     G : Graph
(12)                        A copy of the graph.
(8)                     See Also
(8)                     --------
(8)                     to_directed: return a directed copy of the graph.
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
(8)                     >>> H = G.copy()
(8)                     """
(8)                     if as_view is True:
(12)                        return nx.graphviews.generic_graph_view(self)
(8)                     G = self.__class__()
(8)                     G.graph.update(self.graph)
(8)                     G.add_nodes_from((n, d.copy()) for n, d in self._node.items())
(8)                     G.add_edges_from(
(12)                        (u, v, key, datadict.copy())
(12)                        for u, nbrs in self._adj.items()
(12)                        for v, keydict in nbrs.items()
(12)                        for key, datadict in keydict.items()
(8)                     )
(8)                     return G
(4)                 def to_directed(self, as_view=False):
```

```
(8)                 """Returns a directed representation of the graph.
(8)                 Returns
(8)                 -------
(8)                 G : MultiDiGraph
(12)                    A directed graph with the same name, same nodes, and with
(12)                    each edge (u, v, k, data) replaced by two directed edges
(12)                    (u, v, k, data) and (v, u, k, data).
(8)                 Notes
(8)                 -----
(8)                 This returns a "deepcopy" of the edge, node, and
(8)                 graph attributes which attempts to completely copy
(8)                 all of the data and references.
(8)                 This is in contrast to the similar D=MultiDiGraph(G) which
(8)                 returns a shallow copy of the data.
(8)                 See the Python copy module for more information on shallow
(8)                 and deep copies, https://docs.python.org/3/library/copy.html.
(8)                 Warning: If you have subclassed MultiGraph to use dict-like objects
(8)                 in the data structure, those changes do not transfer to the
(8)                 MultiDiGraph created by this method.
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.MultiGraph()
(8)                 >>> G.add_edge(0, 1)
(8)                 0
(8)                 >>> G.add_edge(0, 1)
(8)                 1
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1)]
(8)                 If already directed, return a (deep) copy
(8)                 >>> G = nx.MultiDiGraph()
(8)                 >>> G.add_edge(0, 1)
(8)                 0
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1, 0)]
(8)                 """
(8)                 graph_class = self.to_directed_class()
(8)                 if as_view is True:
(12)                    return nx.graphviews.generic_graph_view(self, graph_class)
(8)                 G = graph_class()
(8)                 G.graph.update(deepcopy(self.graph))
(8)                 G.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
(8)                 G.add_edges_from(
(12)                    (u, v, key, deepcopy(datadict))
(12)                    for u, nbrs in self.adj.items()
(12)                    for v, keydict in nbrs.items()
(12)                    for key, datadict in keydict.items()
(8)                 )
(8)                 return G
(4)             def to_undirected(self, as_view=False):
(8)                 """Returns an undirected copy of the graph.
(8)                 Returns
(8)                 -------
(8)                 G : Graph/MultiGraph
(12)                    A deepcopy of the graph.
(8)                 See Also
(8)                 --------
(8)                 copy, add_edge, add_edges_from
(8)                 Notes
(8)                 -----
(8)                 This returns a "deepcopy" of the edge, node, and
(8)                 graph attributes which attempts to completely copy
(8)                 all of the data and references.
(8)                 This is in contrast to the similar `G = nx.MultiGraph(D)`
(8)                 which returns a shallow copy of the data.
(8)                 See the Python copy module for more information on shallow
(8)                 and deep copies, https://docs.python.org/3/library/copy.html.
(8)                 Warning: If you have subclassed MultiGraph to use dict-like
```

```
(8)                 objects in the data structure, those changes do not transfer
(8)                 to the MultiGraph created by this method.
(8)                 Examples
(8)                 --------
(8)                 >>> G = nx.MultiGraph([(0, 1), (0, 1), (1, 2)])
(8)                 >>> H = G.to_directed()
(8)                 >>> list(H.edges)
(8)                 [(0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 2, 0), (2, 1, 0)]
(8)                 >>> G2 = H.to_undirected()
(8)                 >>> list(G2.edges)
(8)                 [(0, 1, 0), (0, 1, 1), (1, 2, 0)]
(8)                 """
(8)                 graph_class = self.to_undirected_class()
(8)                 if as_view is True:
(12)                    return nx.graphviews.generic_graph_view(self, graph_class)
(8)                 G = graph_class()
(8)                 G.graph.update(deepcopy(self.graph))
(8)                 G.add_nodes_from((n, deepcopy(d)) for n, d in self._node.items())
(8)                 G.add_edges_from(
(12)                    (u, v, key, deepcopy(datadict))
(12)                    for u, nbrs in self._adj.items()
(12)                    for v, keydict in nbrs.items()
(12)                    for key, datadict in keydict.items()
(8)                 )
(8)                 return G
(4)             def number_of_edges(self, u=None, v=None):
(8)                 """Returns the number of edges between two nodes.
(8)                 Parameters
(8)                 ----------
(8)                 u, v : nodes, optional (Default=all edges)
(12)                    If u and v are specified, return the number of edges between
(12)                    u and v. Otherwise return the total number of all edges.
(8)                 Returns
(8)                 -------
(8)                 nedges : int
(12)                    The number of edges in the graph.  If nodes `u` and `v` are
(12)                    specified return the number of edges between those nodes. If
(12)                    the graph is directed, this only returns the number of edges
(12)                    from `u` to `v`.
(8)                 See Also
(8)                 --------
(8)                 size
(8)                 Examples
(8)                 --------
(8)                 For undirected multigraphs, this method counts the total number
(8)                 of edges in the graph::
(12)                    >>> G = nx.MultiGraph()
(12)                    >>> G.add_edges_from([(0, 1), (0, 1), (1, 2)])
(12)                    [0, 1, 0]
(12)                    >>> G.number_of_edges()
(12)                    3
(8)                 If you specify two nodes, this counts the total number of edges
(8)                 joining the two nodes::
(12)                    >>> G.number_of_edges(0, 1)
(12)                    2
(8)                 For directed multigraphs, this method can count the total number
(8)                 of directed edges from `u` to `v`::
(12)                    >>> G = nx.MultiDiGraph()
(12)                    >>> G.add_edges_from([(0, 1), (0, 1), (1, 0)])
(12)                    [0, 1, 0]
(12)                    >>> G.number_of_edges(0, 1)
(12)                    2
(12)                    >>> G.number_of_edges(1, 0)
(12)                    1
(8)                 """
(8)                 if u is None:
(12)                    return self.size()
(8)                 try:
(12)                    edgedata = self._adj[u][v]
```

```
1041: (8)                   except KeyError:
1042: (12)                      return 0  # no such edge
1043: (8)                   return len(edgedata)
```

File 397 - reportviews.py:

```
1: (0)              """
2: (0)              View Classes provide node, edge and degree "views" of a graph.
3: (0)              Views for nodes, edges and degree are provided for all base graph classes.
4: (0)              A view means a read-only object that is quick to create, automatically
5: (0)              updated when the graph changes, and provides basic access like `n in V`,
6: (0)              `for n in V`, `V[n]` and sometimes set operations.
7: (0)              The views are read-only iterable containers that are updated as the
8: (0)              graph is updated. As with dicts, the graph should not be updated
9: (0)              while iterating through the view. Views can be iterated multiple times.
10: (0)             Edge and Node views also allow data attribute lookup.
11: (0)             The resulting attribute dict is writable as `G.edges[3, 4]['color']='red'`
12: (0)             Degree views allow lookup of degree values for single nodes.
13: (0)             Weighted degree is supported with the `weight` argument.
14: (0)             NodeView
15: (0)             ========
16: (4)                 `V = G.nodes` (or `V = G.nodes()`) allows `len(V)`, `n in V`, set
17: (4)                 operations e.g. "G.nodes & H.nodes", and `dd = G.nodes[n]`, where
18: (4)                 `dd` is the node data dict. Iteration is over the nodes by default.
19: (0)             NodeDataView
20: (0)             ============
21: (4)                 To iterate over (node, data) pairs, use arguments to `G.nodes()`
22: (4)                 to create a DataView e.g. `DV = G.nodes(data='color', default='red')`.
23: (4)                 The DataView iterates as `for n, color in DV` and allows
24: (4)                 `(n, 'red') in DV`. Using `DV = G.nodes(data=True)`, the DataViews
25: (4)                 use the full datadict in writeable form also allowing contain testing as
26: (4)                 `(n, {'color': 'red'}) in VD`. DataViews allow set operations when
27: (4)                 data attributes are hashable.
28: (0)             DegreeView
29: (0)             ==========
30: (4)                 `V = G.degree` allows iteration over (node, degree) pairs as well
31: (4)                 as lookup: `deg=V[n]`. There are many flavors of DegreeView
32: (4)                 for In/Out/Directed/Multi. For Directed Graphs, `G.degree`
33: (4)                 counts both in and out going edges. `G.out_degree` and
34: (4)                 `G.in_degree` count only specific directions.
35: (4)                 Weighted degree using edge data attributes is provide via
36: (4)                 `V = G.degree(weight='attr_name')` where any string with the
37: (4)                 attribute name can be used. `weight=None` is the default.
38: (4)                 No set operations are implemented for degrees, use NodeView.
39: (4)                 The argument `nbunch` restricts iteration to nodes in nbunch.
40: (4)                 The DegreeView can still lookup any node even if nbunch is specified.
41: (0)             EdgeView
42: (0)             ========
43: (4)                 `V = G.edges` or `V = G.edges()` allows iteration over edges as well as
44: (4)                 `e in V`, set operations and edge data lookup `dd = G.edges[2, 3]`.
45: (4)                 Iteration is over 2-tuples `(u, v)` for Graph/DiGraph. For multigraphs
46: (4)                 edges 3-tuples `(u, v, key)` are the default but 2-tuples can be obtained
47: (4)                 via `V = G.edges(keys=False)`.
48: (4)                 Set operations for directed graphs treat the edges as a set of 2-tuples.
49: (4)                 For undirected graphs, 2-tuples are not a unique representation of edges.
50: (4)                 So long as the set being compared to contains unique representations
51: (4)                 of its edges, the set operations will act as expected. If the other
52: (4)                 set contains both `(0, 1)` and `(1, 0)` however, the result of set
53: (4)                 operations may contain both representations of the same edge.
54: (0)             EdgeDataView
55: (0)             ============
56: (4)                 Edge data can be reported using an EdgeDataView typically created
57: (4)                 by calling an EdgeView: `DV = G.edges(data='weight', default=1)`.
58: (4)                 The EdgeDataView allows iteration over edge tuples, membership checking
59: (4)                 but no set operations.
60: (4)                 Iteration depends on `data` and `default` and for multigraph `keys`
61: (4)                 If `data is False` (the default) then iterate over 2-tuples `(u, v)`.
```

```
62: (4)                 If `data is True` iterate over 3-tuples `(u, v, datadict)`.
63: (4)                 Otherwise iterate over `(u, v, datadict.get(data, default))`.
64: (4)                 For Multigraphs, if `keys is True`, replace `u, v` with `u, v, key`
65: (4)                 to create 3-tuples and 4-tuples.
66: (4)                 The argument `nbunch` restricts edges to those incident to nodes in
nbunch.
67: (0)             """
68: (0)             from collections.abc import Mapping, Set
69: (0)             import networkx as nx
70: (0)             __all__ = [
71: (4)                 "NodeView",
72: (4)                 "NodeDataView",
73: (4)                 "EdgeView",
74: (4)                 "OutEdgeView",
75: (4)                 "InEdgeView",
76: (4)                 "EdgeDataView",
77: (4)                 "OutEdgeDataView",
78: (4)                 "InEdgeDataView",
79: (4)                 "MultiEdgeView",
80: (4)                 "OutMultiEdgeView",
81: (4)                 "InMultiEdgeView",
82: (4)                 "MultiEdgeDataView",
83: (4)                 "OutMultiEdgeDataView",
84: (4)                 "InMultiEdgeDataView",
85: (4)                 "DegreeView",
86: (4)                 "DiDegreeView",
87: (4)                 "InDegreeView",
88: (4)                 "OutDegreeView",
89: (4)                 "MultiDegreeView",
90: (4)                 "DiMultiDegreeView",
91: (4)                 "InMultiDegreeView",
92: (4)                 "OutMultiDegreeView",
93: (0)             ]
94: (0)             class NodeView(Mapping, Set):
95: (4)                 """A NodeView class to act as G.nodes for a NetworkX Graph
96: (4)                 Set operations act on the nodes without considering data.
97: (4)                 Iteration is over nodes. Node data can be looked up like a dict.
98: (4)                 Use NodeDataView to iterate over node data or to specify a data
99: (4)                 attribute for lookup. NodeDataView is created by calling the NodeView.
100: (4)                Parameters
101: (4)                ----------
102: (4)                graph : NetworkX graph-like class
103: (4)                Examples
104: (4)                --------
105: (4)                >>> G = nx.path_graph(3)
106: (4)                >>> NV = G.nodes()
107: (4)                >>> 2 in NV
108: (4)                True
109: (4)                >>> for n in NV:
110: (4)                ...     print(n)
111: (4)                0
112: (4)                1
113: (4)                2
114: (4)                >>> assert NV & {1, 2, 3} == {1, 2}
115: (4)                >>> G.add_node(2, color="blue")
116: (4)                >>> NV[2]
117: (4)                {'color': 'blue'}
118: (4)                >>> G.add_node(8, color="red")
119: (4)                >>> NDV = G.nodes(data=True)
120: (4)                >>> (2, NV[2]) in NDV
121: (4)                True
122: (4)                >>> for n, dd in NDV:
123: (4)                ...     print((n, dd.get("color", "aqua")))
124: (4)                (0, 'aqua')
125: (4)                (1, 'aqua')
126: (4)                (2, 'blue')
127: (4)                (8, 'red')
128: (4)                >>> NDV[2] == NV[2]
129: (4)                True
```

```
(4)                 >>> NVdata = G.nodes(data="color", default="aqua")
(4)                 >>> (2, NVdata[2]) in NVdata
(4)                 True
(4)                 >>> for n, dd in NVdata:
(4)                 ...     print((n, dd))
(4)                 (0, 'aqua')
(4)                 (1, 'aqua')
(4)                 (2, 'blue')
(4)                 (8, 'red')
(4)                 >>> NVdata[2] == NV[2]  # NVdata gets 'color', NV gets datadict
(4)                 False
(4)                 """
(4)                 __slots__ = ("_nodes",)
(4)                 def __getstate__(self):
(8)                     return {"_nodes": self._nodes}
(4)                 def __setstate__(self, state):
(8)                     self._nodes = state["_nodes"]
(4)                 def __init__(self, graph):
(8)                     self._nodes = graph._node
(4)                 def __len__(self):
(8)                     return len(self._nodes)
(4)                 def __iter__(self):
(8)                     return iter(self._nodes)
(4)                 def __getitem__(self, n):
(8)                     if isinstance(n, slice):
(12)                        raise nx.NetworkXError(
(16)                            f"{type(self).__name__} does not support slicing, "
(16)                            f"try list(G.nodes)[{n.start}:{n.stop}:{n.step}]"
(12)                        )
(8)                     return self._nodes[n]
(4)                 def __contains__(self, n):
(8)                     return n in self._nodes
(4)                 @classmethod
(4)                 def _from_iterable(cls, it):
(8)                     return set(it)
(4)                 def __call__(self, data=False, default=None):
(8)                     if data is False:
(12)                        return self
(8)                     return NodeDataView(self._nodes, data, default)
(4)                 def data(self, data=True, default=None):
(8)                     """
(8)                     Return a read-only view of node data.
(8)                     Parameters
(8)                     ----------
(8)                     data : bool or node data key, default=True
(12)                        If ``data=True`` (the default), return a `NodeDataView` object
that
(12)                        maps each node to *all* of its attributes. `data` may also be an
(12)                        arbitrary key, in which case the `NodeDataView` maps each node to
(12)                        the value for the keyed attribute. In this case, if a node does
(12)                        not have the `data` attribute, the `default` value is used.
(8)                     default : object, default=None
(12)                        The value used when a node does not have a specific attribute.
(8)                     Returns
(8)                     -------
(8)                     NodeDataView
(12)                        The layout of the returned NodeDataView depends on the value of
the
(12)                        `data` parameter.
(8)                     Notes
(8)                     -----
(8)                     If ``data=False``, returns a `NodeView` object without data.
(8)                     See Also
(8)                     --------
(8)                     NodeDataView
(8)                     Examples
(8)                     --------
(8)                     >>> G = nx.Graph()
(8)                     >>> G.add_nodes_from(
```

```
197: (8)                    ...     [
198: (8)                    ...         (0, {"color": "red", "weight": 10}),
199: (8)                    ...         (1, {"color": "blue"}),
200: (8)                    ...         (2, {"color": "yellow", "weight": 2}),
201: (8)                    ...     ]
202: (8)                    ... )
203: (8)                    Accessing node data with ``data=True`` (the default) returns a
204: (8)                    NodeDataView mapping each node to all of its attributes:
205: (8)                    >>> G.nodes.data()
206: (8)                    NodeDataView({0: {'color': 'red', 'weight': 10}, 1: {'color': 'blue'},
2: {'color': 'yellow', 'weight': 2}})
207: (8)                    If `data` represents  a key in the node attribute dict, a NodeDataView
mapping
208: (8)                    the nodes to the value for that specific key is returned:
209: (8)                    >>> G.nodes.data("color")
210: (8)                    NodeDataView({0: 'red', 1: 'blue', 2: 'yellow'}, data='color')
211: (8)                    If a specific key is not found in an attribute dict, the value
specified
212: (8)                    by `default` is returned:
213: (8)                    >>> G.nodes.data("weight", default=-999)
214: (8)                    NodeDataView({0: 10, 1: -999, 2: 2}, data='weight')
215: (8)                    Note that there is no check that the `data` key is in any of the
216: (8)                    node attribute dictionaries:
217: (8)                    >>> G.nodes.data("height")
218: (8)                    NodeDataView({0: None, 1: None, 2: None}, data='height')
219: (8)                    """
220: (8)                    if data is False:
221: (12)                       return self
222: (8)                    return NodeDataView(self._nodes, data, default)
223: (4)                def __str__(self):
224: (8)                    return str(list(self))
225: (4)                def __repr__(self):
226: (8)                    return f"{self.__class__.__name__}({tuple(self)})"
227: (0)            class NodeDataView(Set):
228: (4)                """A DataView class for nodes of a NetworkX Graph
229: (4)                The main use for this class is to iterate through node-data pairs.
230: (4)                The data can be the entire data-dictionary for each node, or it
231: (4)                can be a specific attribute (with default) for each node.
232: (4)                Set operations are enabled with NodeDataView, but don't work in
233: (4)                cases where the data is not hashable. Use with caution.
234: (4)                Typically, set operations on nodes use NodeView, not NodeDataView.
235: (4)                That is, they use `G.nodes` instead of `G.nodes(data='foo')`.
236: (4)                Parameters
237: (4)                ==========
238: (4)                graph : NetworkX graph-like class
239: (4)                data : bool or string (default=False)
240: (4)                default : object (default=None)
241: (4)                """
242: (4)                __slots__ = ("_nodes", "_data", "_default")
243: (4)                def __getstate__(self):
244: (8)                    return {"_nodes": self._nodes, "_data": self._data, "_default":
self._default}
245: (4)                def __setstate__(self, state):
246: (8)                    self._nodes = state["_nodes"]
247: (8)                    self._data = state["_data"]
248: (8)                    self._default = state["_default"]
249: (4)                def __init__(self, nodedict, data=False, default=None):
250: (8)                    self._nodes = nodedict
251: (8)                    self._data = data
252: (8)                    self._default = default
253: (4)                @classmethod
254: (4)                def _from_iterable(cls, it):
255: (8)                    try:
256: (12)                       return set(it)
257: (8)                    except TypeError as err:
258: (12)                       if "unhashable" in str(err):
259: (16)                           msg = " : Could be b/c data=True or your values are
unhashable"
260: (16)                           raise TypeError(str(err) + msg) from err
```

```
(12)                        raise
(4)                 def __len__(self):
(8)                     return len(self._nodes)
(4)                 def __iter__(self):
(8)                     data = self._data
(8)                     if data is False:
(12)                        return iter(self._nodes)
(8)                     if data is True:
(12)                        return iter(self._nodes.items())
(8)                     return (
(12)                        (n, dd[data] if data in dd else self._default)
(12)                        for n, dd in self._nodes.items()
(8)                     )
(4)                 def __contains__(self, n):
(8)                     try:
(12)                        node_in = n in self._nodes
(8)                     except TypeError:
(12)                        n, d = n
(12)                        return n in self._nodes and self[n] == d
(8)                     if node_in is True:
(12)                        return node_in
(8)                     try:
(12)                        n, d = n
(8)                     except (TypeError, ValueError):
(12)                        return False
(8)                     return n in self._nodes and self[n] == d
(4)                 def __getitem__(self, n):
(8)                     if isinstance(n, slice):
(12)                        raise nx.NetworkXError(
(16)                            f"{type(self).__name__} does not support slicing, "
(16)                            f"try list(G.nodes.data())[{n.start}:{n.stop}:{n.step}]"
(12)                        )
(8)                     ddict = self._nodes[n]
(8)                     data = self._data
(8)                     if data is False or data is True:
(12)                        return ddict
(8)                     return ddict[data] if data in ddict else self._default
(4)                 def __str__(self):
(8)                     return str(list(self))
(4)                 def __repr__(self):
(8)                     name = self.__class__.__name__
(8)                     if self._data is False:
(12)                        return f"{name}({tuple(self)})"
(8)                     if self._data is True:
(12)                        return f"{name}({dict(self)})"
(8)                     return f"{name}({dict(self)}, data={self._data!r})"
(0)             class DiDegreeView:
(4)                 """A View class for degree of nodes in a NetworkX Graph
(4)                 The functionality is like dict.items() with (node, degree) pairs.
(4)                 Additional functionality includes read-only lookup of node degree,
(4)                 and calling with optional features nbunch (for only a subset of nodes)
(4)                 and weight (use edge weights to compute degree).
(4)                 Parameters
(4)                 ==========
(4)                 graph : NetworkX graph-like class
(4)                 nbunch : node, container of nodes, or None meaning all nodes
(default=None)
(4)                 weight : bool or string (default=None)
(4)                 Notes
(4)                 -----
(4)                 DegreeView can still lookup any node even if nbunch is specified.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(3)
(4)                 >>> DV = G.degree()
(4)                 >>> assert DV[2] == 1
(4)                 >>> assert sum(deg for n, deg in DV) == 4
(4)                 >>> DVweight = G.degree(weight="span")
(4)                 >>> G.add_edge(1, 2, span=34)
```

```
329: (4)                >>> DVweight[2]
330: (4)                34
331: (4)                >>> DVweight[0]  #  default edge weight is 1
332: (4)                1
333: (4)                >>> sum(span for n, span in DVweight)  # sum weighted degrees
334: (4)                70
335: (4)                >>> DVnbunch = G.degree(nbunch=(1, 2))
336: (4)                >>> assert len(list(DVnbunch)) == 2  # iteration over nbunch only
337: (4)                """
338: (4)                def __init__(self, G, nbunch=None, weight=None):
339: (8)                    self._graph = G
340: (8)                    self._succ = G._succ if hasattr(G, "_succ") else G._adj
341: (8)                    self._pred = G._pred if hasattr(G, "_pred") else G._adj
342: (8)                    self._nodes = self._succ if nbunch is None else
list(G.nbunch_iter(nbunch))
343: (8)                    self._weight = weight
344: (4)                def __call__(self, nbunch=None, weight=None):
345: (8)                    if nbunch is None:
346: (12)                       if weight == self._weight:
347: (16)                           return self
348: (12)                       return self.__class__(self._graph, None, weight)
349: (8)                    try:
350: (12)                       if nbunch in self._nodes:
351: (16)                           if weight == self._weight:
352: (20)                               return self[nbunch]
353: (16)                           return self.__class__(self._graph, None, weight)[nbunch]
354: (8)                    except TypeError:
355: (12)                       pass
356: (8)                    return self.__class__(self._graph, nbunch, weight)
357: (4)                def __getitem__(self, n):
358: (8)                    weight = self._weight
359: (8)                    succs = self._succ[n]
360: (8)                    preds = self._pred[n]
361: (8)                    if weight is None:
362: (12)                       return len(succs) + len(preds)
363: (8)                    return sum(dd.get(weight, 1) for dd in succs.values()) + sum(
364: (12)                       dd.get(weight, 1) for dd in preds.values()
365: (8)                    )
366: (4)                def __iter__(self):
367: (8)                    weight = self._weight
368: (8)                    if weight is None:
369: (12)                       for n in self._nodes:
370: (16)                           succs = self._succ[n]
371: (16)                           preds = self._pred[n]
372: (16)                           yield (n, len(succs) + len(preds))
373: (8)                    else:
374: (12)                       for n in self._nodes:
375: (16)                           succs = self._succ[n]
376: (16)                           preds = self._pred[n]
377: (16)                           deg = sum(dd.get(weight, 1) for dd in succs.values()) + sum(
378: (20)                               dd.get(weight, 1) for dd in preds.values()
379: (16)                           )
380: (16)                           yield (n, deg)
381: (4)                def __len__(self):
382: (8)                    return len(self._nodes)
383: (4)                def __str__(self):
384: (8)                    return str(list(self))
385: (4)                def __repr__(self):
386: (8)                    return f"{self.__class__.__name__}({dict(self)})"
387: (0)            class DegreeView(DiDegreeView):
388: (4)                """A DegreeView class to act as G.degree for a NetworkX Graph
389: (4)                Typical usage focuses on iteration over `(node, degree)` pairs.
390: (4)                The degree is by default the number of edges incident to the node.
391: (4)                Optional argument `weight` enables weighted degree using the edge
392: (4)                attribute named in the `weight` argument.  Reporting and iteration
393: (4)                can also be restricted to a subset of nodes using `nbunch`.
394: (4)                Additional functionality include node lookup so that `G.degree[n]`
395: (4)                reported the (possibly weighted) degree of node `n`. Calling the
396: (4)                view creates a view with different arguments `nbunch` or `weight`.
```

```
(4)                 Parameters
(4)                 ==========
(4)                 graph : NetworkX graph-like class
(4)                 nbunch : node, container of nodes, or None meaning all nodes (default=None)
(4)                 weight : string or None (default=None)
(4)                 Notes
(4)                 -----
(4)                 DegreeView can still lookup any node even if nbunch is specified.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(3)
(4)                 >>> DV = G.degree()
(4)                 >>> assert DV[2] == 1
(4)                 >>> assert G.degree[2] == 1
(4)                 >>> assert sum(deg for n, deg in DV) == 4
(4)                 >>> DVweight = G.degree(weight="span")
(4)                 >>> G.add_edge(1, 2, span=34)
(4)                 >>> DVweight[2]
(4)                 34
(4)                 >>> DVweight[0]  #  default edge weight is 1
(4)                 1
(4)                 >>> sum(span for n, span in DVweight)  # sum weighted degrees
(4)                 70
(4)                 >>> DVnbunch = G.degree(nbunch=(1, 2))
(4)                 >>> assert len(list(DVnbunch)) == 2  # iteration over nbunch only
(4)                 """
(4)                 def __getitem__(self, n):
(8)                     weight = self._weight
(8)                     nbrs = self._succ[n]
(8)                     if weight is None:
(12)                        return len(nbrs) + (n in nbrs)
(8)                     return sum(dd.get(weight, 1) for dd in nbrs.values()) + (
(12)                        n in nbrs and nbrs[n].get(weight, 1)
(8)                     )
(4)                 def __iter__(self):
(8)                     weight = self._weight
(8)                     if weight is None:
(12)                        for n in self._nodes:
(16)                            nbrs = self._succ[n]
(16)                            yield (n, len(nbrs) + (n in nbrs))
(8)                     else:
(12)                        for n in self._nodes:
(16)                            nbrs = self._succ[n]
(16)                            deg = sum(dd.get(weight, 1) for dd in nbrs.values()) + (
(20)                                n in nbrs and nbrs[n].get(weight, 1)
(16)                            )
(16)                            yield (n, deg)
(0)             class OutDegreeView(DiDegreeView):
(4)                 """A DegreeView class to report out_degree for a DiGraph; See DegreeView"""
(4)                 def __getitem__(self, n):
(8)                     weight = self._weight
(8)                     nbrs = self._succ[n]
(8)                     if self._weight is None:
(12)                        return len(nbrs)
(8)                     return sum(dd.get(self._weight, 1) for dd in nbrs.values())
(4)                 def __iter__(self):
(8)                     weight = self._weight
(8)                     if weight is None:
(12)                        for n in self._nodes:
(16)                            succs = self._succ[n]
(16)                            yield (n, len(succs))
(8)                     else:
(12)                        for n in self._nodes:
(16)                            succs = self._succ[n]
(16)                            deg = sum(dd.get(weight, 1) for dd in succs.values())
(16)                            yield (n, deg)
(0)             class InDegreeView(DiDegreeView):
```

```
(4)                 """A DegreeView class to report in_degree for a DiGraph; See DegreeView"""
(4)                 def __getitem__(self, n):
(8)                     weight = self._weight
(8)                     nbrs = self._pred[n]
(8)                     if weight is None:
(12)                        return len(nbrs)
(8)                     return sum(dd.get(weight, 1) for dd in nbrs.values())
(4)                 def __iter__(self):
(8)                     weight = self._weight
(8)                     if weight is None:
(12)                        for n in self._nodes:
(16)                            preds = self._pred[n]
(16)                            yield (n, len(preds))
(8)                     else:
(12)                        for n in self._nodes:
(16)                            preds = self._pred[n]
(16)                            deg = sum(dd.get(weight, 1) for dd in preds.values())
(16)                            yield (n, deg)
(0)             class MultiDegreeView(DiDegreeView):
(4)                 """A DegreeView class for undirected multigraphs; See DegreeView"""
(4)                 def __getitem__(self, n):
(8)                     weight = self._weight
(8)                     nbrs = self._succ[n]
(8)                     if weight is None:
(12)                        return sum(len(keys) for keys in nbrs.values()) + (
(16)                            n in nbrs and len(nbrs[n])
(12)                        )
(8)                     deg = sum(
(12)                        d.get(weight, 1) for key_dict in nbrs.values() for d in
key_dict.values()
(8)                     )
(8)                     if n in nbrs:
(12)                        deg += sum(d.get(weight, 1) for d in nbrs[n].values())
(8)                     return deg
(4)                 def __iter__(self):
(8)                     weight = self._weight
(8)                     if weight is None:
(12)                        for n in self._nodes:
(16)                            nbrs = self._succ[n]
(16)                            deg = sum(len(keys) for keys in nbrs.values()) + (
(20)                                n in nbrs and len(nbrs[n])
(16)                            )
(16)                            yield (n, deg)
(8)                     else:
(12)                        for n in self._nodes:
(16)                            nbrs = self._succ[n]
(16)                            deg = sum(
(20)                                d.get(weight, 1)
(20)                                for key_dict in nbrs.values()
(20)                                for d in key_dict.values()
(16)                            )
(16)                            if n in nbrs:
(20)                                deg += sum(d.get(weight, 1) for d in nbrs[n].values())
(16)                            yield (n, deg)
(0)             class DiMultiDegreeView(DiDegreeView):
(4)                 """A DegreeView class for MultiDiGraph; See DegreeView"""
(4)                 def __getitem__(self, n):
(8)                     weight = self._weight
(8)                     succs = self._succ[n]
(8)                     preds = self._pred[n]
(8)                     if weight is None:
(12)                        return sum(len(keys) for keys in succs.values()) + sum(
(16)                            len(keys) for keys in preds.values()
(12)                        )
(8)                     deg = sum(
(12)                        d.get(weight, 1) for key_dict in succs.values() for d in
key_dict.values()
(8)                     ) + sum(
(12)                        d.get(weight, 1) for key_dict in preds.values() for d in
```

```
key_dict.values()
531: (8)                    )
532: (8)                    return deg
533: (4)                def __iter__(self):
534: (8)                    weight = self._weight
535: (8)                    if weight is None:
536: (12)                       for n in self._nodes:
537: (16)                           succs = self._succ[n]
538: (16)                           preds = self._pred[n]
539: (16)                           deg = sum(len(keys) for keys in succs.values()) + sum(
540: (20)                               len(keys) for keys in preds.values()
541: (16)                           )
542: (16)                           yield (n, deg)
543: (8)                    else:
544: (12)                       for n in self._nodes:
545: (16)                           succs = self._succ[n]
546: (16)                           preds = self._pred[n]
547: (16)                           deg = sum(
548: (20)                               d.get(weight, 1)
549: (20)                               for key_dict in succs.values()
550: (20)                               for d in key_dict.values()
551: (16)                           ) + sum(
552: (20)                               d.get(weight, 1)
553: (20)                               for key_dict in preds.values()
554: (20)                               for d in key_dict.values()
555: (16)                           )
556: (16)                           yield (n, deg)
557: (0)            class InMultiDegreeView(DiDegreeView):
558: (4)                """A DegreeView class for inward degree of MultiDiGraph; See DegreeView"""
559: (4)                def __getitem__(self, n):
560: (8)                    weight = self._weight
561: (8)                    nbrs = self._pred[n]
562: (8)                    if weight is None:
563: (12)                       return sum(len(data) for data in nbrs.values())
564: (8)                    return sum(
565: (12)                       d.get(weight, 1) for key_dict in nbrs.values() for d in
key_dict.values()
566: (8)                    )
567: (4)                def __iter__(self):
568: (8)                    weight = self._weight
569: (8)                    if weight is None:
570: (12)                       for n in self._nodes:
571: (16)                           nbrs = self._pred[n]
572: (16)                           deg = sum(len(data) for data in nbrs.values())
573: (16)                           yield (n, deg)
574: (8)                    else:
575: (12)                       for n in self._nodes:
576: (16)                           nbrs = self._pred[n]
577: (16)                           deg = sum(
578: (20)                               d.get(weight, 1)
579: (20)                               for key_dict in nbrs.values()
580: (20)                               for d in key_dict.values()
581: (16)                           )
582: (16)                           yield (n, deg)
583: (0)            class OutMultiDegreeView(DiDegreeView):
584: (4)                """A DegreeView class for outward degree of MultiDiGraph; See
DegreeView"""
585: (4)                def __getitem__(self, n):
586: (8)                    weight = self._weight
587: (8)                    nbrs = self._succ[n]
588: (8)                    if weight is None:
589: (12)                       return sum(len(data) for data in nbrs.values())
590: (8)                    return sum(
591: (12)                       d.get(weight, 1) for key_dict in nbrs.values() for d in
key_dict.values()
592: (8)                    )
593: (4)                def __iter__(self):
594: (8)                    weight = self._weight
595: (8)                    if weight is None:
```

```
596: (12)                        for n in self._nodes:
597: (16)                            nbrs = self._succ[n]
598: (16)                            deg = sum(len(data) for data in nbrs.values())
599: (16)                            yield (n, deg)
600: (8)                    else:
601: (12)                        for n in self._nodes:
602: (16)                            nbrs = self._succ[n]
603: (16)                            deg = sum(
604: (20)                                d.get(weight, 1)
605: (20)                                for key_dict in nbrs.values()
606: (20)                                for d in key_dict.values()
607: (16)                            )
608: (16)                            yield (n, deg)
609: (0)            class OutEdgeDataView:
610: (4)                """EdgeDataView for outward edges of DiGraph; See EdgeDataView"""
611: (4)                __slots__ = (
612: (8)                    "_viewer",
613: (8)                    "_nbunch",
614: (8)                    "_data",
615: (8)                    "_default",
616: (8)                    "_adjdict",
617: (8)                    "_nodes_nbrs",
618: (8)                    "_report",
619: (4)                )
620: (4)                def __getstate__(self):
621: (8)                    return {
622: (12)                        "viewer": self._viewer,
623: (12)                        "nbunch": self._nbunch,
624: (12)                        "data": self._data,
625: (12)                        "default": self._default,
626: (8)                    }
627: (4)                def __setstate__(self, state):
628: (8)                    self.__init__(**state)
629: (4)                def __init__(self, viewer, nbunch=None, data=False, *, default=None):
630: (8)                    self._viewer = viewer
631: (8)                    adjdict = self._adjdict = viewer._adjdict
632: (8)                    if nbunch is None:
633: (12)                       self._nodes_nbrs = adjdict.items
634: (8)                    else:
635: (12)                       nbunch = dict.fromkeys(viewer._graph.nbunch_iter(nbunch))
636: (12)                       self._nodes_nbrs = lambda: [(n, adjdict[n]) for n in nbunch]
637: (8)                    self._nbunch = nbunch
638: (8)                    self._data = data
639: (8)                    self._default = default
640: (8)                    if data is True:
641: (12)                       self._report = lambda n, nbr, dd: (n, nbr, dd)
642: (8)                    elif data is False:
643: (12)                       self._report = lambda n, nbr, dd: (n, nbr)
644: (8)                    else:  # data is attribute name
645: (12)                       self._report = (
646: (16)                           lambda n, nbr, dd: (n, nbr, dd[data])
647: (16)                           if data in dd
648: (16)                           else (n, nbr, default)
649: (12)                       )
650: (4)                def __len__(self):
651: (8)                    return sum(len(nbrs) for n, nbrs in self._nodes_nbrs())
652: (4)                def __iter__(self):
653: (8)                    return (
654: (12)                       self._report(n, nbr, dd)
655: (12)                       for n, nbrs in self._nodes_nbrs()
656: (12)                       for nbr, dd in nbrs.items()
657: (8)                    )
658: (4)                def __contains__(self, e):
659: (8)                    u, v = e[:2]
660: (8)                    if self._nbunch is not None and u not in self._nbunch:
661: (12)                       return False  # this edge doesn't start in nbunch
662: (8)                    try:
663: (12)                       ddict = self._adjdict[u][v]
664: (8)                    except KeyError:
```

```
(12)                        return False
(8)                     return e == self._report(u, v, ddict)
(4)                 def __str__(self):
(8)                     return str(list(self))
(4)                 def __repr__(self):
(8)                     return f"{self.__class__.__name__}({list(self)})"
(0)             class EdgeDataView(OutEdgeDataView):
(4)                 """A EdgeDataView class for edges of Graph
(4)                 This view is primarily used to iterate over the edges reporting
(4)                 edges as node-tuples with edge data optionally reported. The
(4)                 argument `nbunch` allows restriction to edges incident to nodes
(4)                 in that container/singleton. The default (nbunch=None)
(4)                 reports all edges. The arguments `data` and `default` control
(4)                 what edge data is reported. The default `data is False` reports
(4)                 only node-tuples for each edge. If `data is True` the entire edge
(4)                 data dict is returned. Otherwise `data` is assumed to hold the name
(4)                 of the edge attribute to report with default `default` if  that
(4)                 edge attribute is not present.
(4)                 Parameters
(4)                 ----------
(4)                 nbunch : container of nodes, node or None (default None)
(4)                 data : False, True or string (default False)
(4)                 default : default value (default None)
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(3)
(4)                 >>> G.add_edge(1, 2, foo="bar")
(4)                 >>> list(G.edges(data="foo", default="biz"))
(4)                 [(0, 1, 'biz'), (1, 2, 'bar')]
(4)                 >>> assert (0, 1, "biz") in G.edges(data="foo", default="biz")
(4)                 """
(4)                 __slots__ = ()
(4)                 def __len__(self):
(8)                     return sum(1 for e in self)
(4)                 def __iter__(self):
(8)                     seen = {}
(8)                     for n, nbrs in self._nodes_nbrs():
(12)                        for nbr, dd in nbrs.items():
(16)                            if nbr not in seen:
(20)                                yield self._report(n, nbr, dd)
(12)                        seen[n] = 1
(8)                     del seen
(4)                 def __contains__(self, e):
(8)                     u, v = e[:2]
(8)                     if self._nbunch is not None and u not in self._nbunch and v not in
self._nbunch:
(12)                        return False  # this edge doesn't start and it doesn't end in
nbunch
(8)                     try:
(12)                        ddict = self._adjdict[u][v]
(8)                     except KeyError:
(12)                        return False
(8)                     return e == self._report(u, v, ddict)
(0)             class InEdgeDataView(OutEdgeDataView):
(4)                 """An EdgeDataView class for outward edges of DiGraph; See EdgeDataView"""
(4)                 __slots__ = ()
(4)                 def __iter__(self):
(8)                     return (
(12)                        self._report(nbr, n, dd)
(12)                        for n, nbrs in self._nodes_nbrs()
(12)                        for nbr, dd in nbrs.items()
(8)                     )
(4)                 def __contains__(self, e):
(8)                     u, v = e[:2]
(8)                     if self._nbunch is not None and v not in self._nbunch:
(12)                        return False  # this edge doesn't end in nbunch
(8)                     try:
(12)                        ddict = self._adjdict[v][u]
(8)                     except KeyError:
```

```
732: (12)                        return False
733: (8)                    return e == self._report(u, v, ddict)
734: (0)            class OutMultiEdgeDataView(OutEdgeDataView):
735: (4)                """An EdgeDataView for outward edges of MultiDiGraph; See EdgeDataView"""
736: (4)                __slots__ = ("keys",)
737: (4)                def __getstate__(self):
738: (8)                    return {
739: (12)                        "viewer": self._viewer,
740: (12)                        "nbunch": self._nbunch,
741: (12)                        "keys": self.keys,
742: (12)                        "data": self._data,
743: (12)                        "default": self._default,
744: (8)                    }
745: (4)                def __setstate__(self, state):
746: (8)                    self.__init__(**state)
747: (4)                def __init__(self, viewer, nbunch=None, data=False, *, default=None,
keys=False):
748: (8)                    self._viewer = viewer
749: (8)                    adjdict = self._adjdict = viewer._adjdict
750: (8)                    self.keys = keys
751: (8)                    if nbunch is None:
752: (12)                       self._nodes_nbrs = adjdict.items
753: (8)                    else:
754: (12)                       nbunch = dict.fromkeys(viewer._graph.nbunch_iter(nbunch))
755: (12)                       self._nodes_nbrs = lambda: [(n, adjdict[n]) for n in nbunch]
756: (8)                    self._nbunch = nbunch
757: (8)                    self._data = data
758: (8)                    self._default = default
759: (8)                    if data is True:
760: (12)                       if keys is True:
761: (16)                           self._report = lambda n, nbr, k, dd: (n, nbr, k, dd)
762: (12)                       else:
763: (16)                           self._report = lambda n, nbr, k, dd: (n, nbr, dd)
764: (8)                    elif data is False:
765: (12)                       if keys is True:
766: (16)                           self._report = lambda n, nbr, k, dd: (n, nbr, k)
767: (12)                       else:
768: (16)                           self._report = lambda n, nbr, k, dd: (n, nbr)
769: (8)                    else:  # data is attribute name
770: (12)                       if keys is True:
771: (16)                           self._report = (
772: (20)                               lambda n, nbr, k, dd: (n, nbr, k, dd[data])
773: (20)                               if data in dd
774: (20)                               else (n, nbr, k, default)
775: (16)                           )
776: (12)                       else:
777: (16)                           self._report = (
778: (20)                               lambda n, nbr, k, dd: (n, nbr, dd[data])
779: (20)                               if data in dd
780: (20)                               else (n, nbr, default)
781: (16)                           )
782: (4)                def __len__(self):
783: (8)                    return sum(1 for e in self)
784: (4)                def __iter__(self):
785: (8)                    return (
786: (12)                       self._report(n, nbr, k, dd)
787: (12)                       for n, nbrs in self._nodes_nbrs()
788: (12)                       for nbr, kd in nbrs.items()
789: (12)                       for k, dd in kd.items()
790: (8)                    )
791: (4)                def __contains__(self, e):
792: (8)                    u, v = e[:2]
793: (8)                    if self._nbunch is not None and u not in self._nbunch:
794: (12)                       return False  # this edge doesn't start in nbunch
795: (8)                    try:
796: (12)                       kdict = self._adjdict[u][v]
797: (8)                    except KeyError:
798: (12)                       return False
799: (8)                    if self.keys is True:
```

```
800: (12)                        k = e[2]
801: (12)                        try:
802: (16)                            dd = kdict[k]
803: (12)                        except KeyError:
804: (16)                            return False
805: (12)                        return e == self._report(u, v, k, dd)
806: (8)                    return any(e == self._report(u, v, k, dd) for k, dd in kdict.items())
807: (0)            class MultiEdgeDataView(OutMultiEdgeDataView):
808: (4)                """An EdgeDataView class for edges of MultiGraph; See EdgeDataView"""
809: (4)                __slots__ = ()
810: (4)                def __iter__(self):
811: (8)                    seen = {}
812: (8)                    for n, nbrs in self._nodes_nbrs():
813: (12)                        for nbr, kd in nbrs.items():
814: (16)                            if nbr not in seen:
815: (20)                                for k, dd in kd.items():
816: (24)                                    yield self._report(n, nbr, k, dd)
817: (12)                        seen[n] = 1
818: (8)                    del seen
819: (4)                def __contains__(self, e):
820: (8)                    u, v = e[:2]
821: (8)                    if self._nbunch is not None and u not in self._nbunch and v not in
self._nbunch:
822: (12)                        return False  # this edge doesn't start and doesn't end in nbunch
823: (8)                    try:
824: (12)                        kdict = self._adjdict[u][v]
825: (8)                    except KeyError:
826: (12)                        try:
827: (16)                            kdict = self._adjdict[v][u]
828: (12)                        except KeyError:
829: (16)                            return False
830: (8)                    if self.keys is True:
831: (12)                        k = e[2]
832: (12)                        try:
833: (16)                            dd = kdict[k]
834: (12)                        except KeyError:
835: (16)                            return False
836: (12)                        return e == self._report(u, v, k, dd)
837: (8)                    return any(e == self._report(u, v, k, dd) for k, dd in kdict.items())
838: (0)            class InMultiEdgeDataView(OutMultiEdgeDataView):
839: (4)                """An EdgeDataView for inward edges of MultiDiGraph; See EdgeDataView"""
840: (4)                __slots__ = ()
841: (4)                def __iter__(self):
842: (8)                    return (
843: (12)                        self._report(nbr, n, k, dd)
844: (12)                        for n, nbrs in self._nodes_nbrs()
845: (12)                        for nbr, kd in nbrs.items()
846: (12)                        for k, dd in kd.items()
847: (8)                    )
848: (4)                def __contains__(self, e):
849: (8)                    u, v = e[:2]
850: (8)                    if self._nbunch is not None and v not in self._nbunch:
851: (12)                        return False  # this edge doesn't end in nbunch
852: (8)                    try:
853: (12)                        kdict = self._adjdict[v][u]
854: (8)                    except KeyError:
855: (12)                        return False
856: (8)                    if self.keys is True:
857: (12)                        k = e[2]
858: (12)                        dd = kdict[k]
859: (12)                        return e == self._report(u, v, k, dd)
860: (8)                    return any(e == self._report(u, v, k, dd) for k, dd in kdict.items())
861: (0)            class OutEdgeView(Set, Mapping):
862: (4)                """A EdgeView class for outward edges of a DiGraph"""
863: (4)                __slots__ = ("_adjdict", "_graph", "_nodes_nbrs")
864: (4)                def __getstate__(self):
865: (8)                    return {"_graph": self._graph, "_adjdict": self._adjdict}
866: (4)                def __setstate__(self, state):
867: (8)                    self._graph = state["_graph"]
```

```
868: (8)                    self._adjdict = state["_adjdict"]
869: (8)                    self._nodes_nbrs = self._adjdict.items
870: (4)                @classmethod
871: (4)                def _from_iterable(cls, it):
872: (8)                    return set(it)
873: (4)                dataview = OutEdgeDataView
874: (4)                def __init__(self, G):
875: (8)                    self._graph = G
876: (8)                    self._adjdict = G._succ if hasattr(G, "succ") else G._adj
877: (8)                    self._nodes_nbrs = self._adjdict.items
878: (4)                def __len__(self):
879: (8)                    return sum(len(nbrs) for n, nbrs in self._nodes_nbrs())
880: (4)                def __iter__(self):
881: (8)                    for n, nbrs in self._nodes_nbrs():
882: (12)                       for nbr in nbrs:
883: (16)                           yield (n, nbr)
884: (4)                def __contains__(self, e):
885: (8)                    try:
886: (12)                       u, v = e
887: (12)                       return v in self._adjdict[u]
888: (8)                    except KeyError:
889: (12)                       return False
890: (4)                def __getitem__(self, e):
891: (8)                    if isinstance(e, slice):
892: (12)                       raise nx.NetworkXError(
893: (16)                           f"{type(self).__name__} does not support slicing, "
894: (16)                           f"try list(G.edges)[{e.start}:{e.stop}:{e.step}]"
895: (12)                       )
896: (8)                    u, v = e
897: (8)                    try:
898: (12)                       return self._adjdict[u][v]
899: (8)                    except KeyError as ex:  # Customize msg to indicate exception origin
900: (12)                       raise KeyError(f"The edge {e} is not in the graph.")
901: (4)                def __call__(self, nbunch=None, data=False, *, default=None):
902: (8)                    if nbunch is None and data is False:
903: (12)                       return self
904: (8)                    return self.dataview(self, nbunch, data, default=default)
905: (4)                def data(self, data=True, default=None, nbunch=None):
906: (8)                    """
907: (8)                    Return a read-only view of edge data.
908: (8)                    Parameters
909: (8)                    ----------
910: (8)                    data : bool or edge attribute key
911: (12)                       If ``data=True``, then the data view maps each edge to a
dictionary
912: (12)                       containing all of its attributes. If `data` is a key in the edge
913: (12)                       dictionary, then the data view maps each edge to its value for
914: (12)                       the keyed attribute. In this case, if the edge doesn't have the
915: (12)                       attribute, the `default` value is returned.
916: (8)                    default : object, default=None
917: (12)                       The value used when an edge does not have a specific attribute
918: (8)                    nbunch : container of nodes, optional (default=None)
919: (12)                       Allows restriction to edges only involving certain nodes. All
edges
920: (12)                       are considered by default.
921: (8)                    Returns
922: (8)                    -------
923: (8)                    dataview
924: (12)                       Returns an `EdgeDataView` for undirected Graphs, `OutEdgeDataView`
925: (12)                       for DiGraphs, `MultiEdgeDataView` for MultiGraphs and
926: (12)                       `OutMultiEdgeDataView` for MultiDiGraphs.
927: (8)                    Notes
928: (8)                    -----
929: (8)                    If ``data=False``, returns an `EdgeView` without any edge data.
930: (8)                    See Also
931: (8)                    --------
932: (8)                    EdgeDataView
933: (8)                    OutEdgeDataView
934: (8)                    MultiEdgeDataView
```

```
935: (8)                    OutMultiEdgeDataView
936: (8)                    Examples
937: (8)                    --------
938: (8)                    >>> G = nx.Graph()
939: (8)                    >>> G.add_edges_from(
940: (8)                    ...     [
941: (8)                    ...         (0, 1, {"dist": 3, "capacity": 20}),
942: (8)                    ...         (1, 2, {"dist": 4}),
943: (8)                    ...         (2, 0, {"dist": 5}),
944: (8)                    ...     ]
945: (8)                    ... )
946: (8)                    Accessing edge data with ``data=True`` (the default) returns an
947: (8)                    edge data view object listing each edge with all of its attributes:
948: (8)                    >>> G.edges.data()
949: (8)                    EdgeDataView([(0, 1, {'dist': 3, 'capacity': 20}), (0, 2, {'dist':
5}), (1, 2, {'dist': 4})])
950: (8)                    If `data` represents a key in the edge attribute dict, a dataview
listing
951: (8)                    each edge with its value for that specific key is returned:
952: (8)                    >>> G.edges.data("dist")
953: (8)                    EdgeDataView([(0, 1, 3), (0, 2, 5), (1, 2, 4)])
954: (8)                    `nbunch` can be used to limit the edges:
955: (8)                    >>> G.edges.data("dist", nbunch=[0])
956: (8)                    EdgeDataView([(0, 1, 3), (0, 2, 5)])
957: (8)                    If a specific key is not found in an edge attribute dict, the value
958: (8)                    specified by `default` is used:
959: (8)                    >>> G.edges.data("capacity")
960: (8)                    EdgeDataView([(0, 1, 20), (0, 2, None), (1, 2, None)])
961: (8)                    Note that there is no check that the `data` key is present in any of
962: (8)                    the edge attribute dictionaries:
963: (8)                    >>> G.edges.data("speed")
964: (8)                    EdgeDataView([(0, 1, None), (0, 2, None), (1, 2, None)])
965: (8)                    """
966: (8)                    if nbunch is None and data is False:
967: (12)                       return self
968: (8)                    return self.dataview(self, nbunch, data, default=default)
969: (4)                def __str__(self):
970: (8)                    return str(list(self))
971: (4)                def __repr__(self):
972: (8)                    return f"{self.__class__.__name__}({list(self)})"
973: (0)            class EdgeView(OutEdgeView):
974: (4)                """A EdgeView class for edges of a Graph
975: (4)                This densely packed View allows iteration over edges, data lookup
976: (4)                like a dict and set operations on edges represented by node-tuples.
977: (4)                In addition, edge data can be controlled by calling this object
978: (4)                possibly creating an EdgeDataView. Typically edges are iterated over
979: (4)                and reported as `(u, v)` node tuples or `(u, v, key)` node/key tuples
980: (4)                for multigraphs. Those edge representations can also be using to
981: (4)                lookup the data dict for any edge. Set operations also are available
982: (4)                where those tuples are the elements of the set.
983: (4)                Calling this object with optional arguments `data`, `default` and `keys`
984: (4)                controls the form of the tuple (see EdgeDataView). Optional argument
985: (4)                `nbunch` allows restriction to edges only involving certain nodes.
986: (4)                If `data is False` (the default) then iterate over 2-tuples `(u, v)`.
987: (4)                If `data is True` iterate over 3-tuples `(u, v, datadict)`.
988: (4)                Otherwise iterate over `(u, v, datadict.get(data, default))`.
989: (4)                For Multigraphs, if `keys is True`, replace `u, v` with `u, v, key` above.
990: (4)                Parameters
991: (4)                ==========
992: (4)                graph : NetworkX graph-like class
993: (4)                nbunch : (default= all nodes in graph) only report edges with these nodes
994: (4)                keys : (only for MultiGraph. default=False) report edge key in tuple
995: (4)                data : bool or string (default=False) see above
996: (4)                default : object (default=None)
997: (4)                Examples
998: (4)                ========
999: (4)                >>> G = nx.path_graph(4)
1000: (4)               >>> EV = G.edges()
1001: (4)               >>> (2, 3) in EV
```

```
(4)                 True
(4)                 >>> for u, v in EV:
(4)                 ...     print((u, v))
(4)                 (0, 1)
(4)                 (1, 2)
(4)                 (2, 3)
(4)                 >>> assert EV & {(1, 2), (3, 4)} == {(1, 2)}
(4)                 >>> EVdata = G.edges(data="color", default="aqua")
(4)                 >>> G.add_edge(2, 3, color="blue")
(4)                 >>> assert (2, 3, "blue") in EVdata
(4)                 >>> for u, v, c in EVdata:
(4)                 ...     print(f"({u}, {v}) has color: {c}")
(4)                 (0, 1) has color: aqua
(4)                 (1, 2) has color: aqua
(4)                 (2, 3) has color: blue
(4)                 >>> EVnbunch = G.edges(nbunch=2)
(4)                 >>> assert (2, 3) in EVnbunch
(4)                 >>> assert (0, 1) not in EVnbunch
(4)                 >>> for u, v in EVnbunch:
(4)                 ...     assert u == 2 or v == 2
(4)                 >>> MG = nx.path_graph(4, create_using=nx.MultiGraph)
(4)                 >>> EVmulti = MG.edges(keys=True)
(4)                 >>> (2, 3, 0) in EVmulti
(4)                 True
(4)                 >>> (2, 3) in EVmulti  # 2-tuples work even when keys is True
(4)                 True
(4)                 >>> key = MG.add_edge(2, 3)
(4)                 >>> for u, v, k in EVmulti:
(4)                 ...     print((u, v, k))
(4)                 (0, 1, 0)
(4)                 (1, 2, 0)
(4)                 (2, 3, 0)
(4)                 (2, 3, 1)
(4)                 """
(4)                 __slots__ = ()
(4)                 dataview = EdgeDataView
(4)                 def __len__(self):
(8)                     num_nbrs = (len(nbrs) + (n in nbrs) for n, nbrs in self._nodes_nbrs())
(8)                     return sum(num_nbrs) // 2
(4)                 def __iter__(self):
(8)                     seen = {}
(8)                     for n, nbrs in self._nodes_nbrs():
(12)                        for nbr in list(nbrs):
(16)                            if nbr not in seen:
(20)                                yield (n, nbr)
(12)                        seen[n] = 1
(8)                     del seen
(4)                 def __contains__(self, e):
(8)                     try:
(12)                        u, v = e[:2]
(12)                        return v in self._adjdict[u] or u in self._adjdict[v]
(8)                     except (KeyError, ValueError):
(12)                        return False
(0)             class InEdgeView(OutEdgeView):
(4)                 """A EdgeView class for inward edges of a DiGraph"""
(4)                 __slots__ = ()
(4)                 def __setstate__(self, state):
(8)                     self._graph = state["_graph"]
(8)                     self._adjdict = state["_adjdict"]
(8)                     self._nodes_nbrs = self._adjdict.items
(4)                 dataview = InEdgeDataView
(4)                 def __init__(self, G):
(8)                     self._graph = G
(8)                     self._adjdict = G._pred if hasattr(G, "pred") else G._adj
(8)                     self._nodes_nbrs = self._adjdict.items
(4)                 def __iter__(self):
(8)                     for n, nbrs in self._nodes_nbrs():
(12)                        for nbr in nbrs:
(16)                            yield (nbr, n)
```

```
(4)     def __contains__(self, e):
(8)         try:
(12)            u, v = e
(12)            return u in self._adjdict[v]
(8)         except KeyError:
(12)            return False
(4)     def __getitem__(self, e):
(8)         if isinstance(e, slice):
(12)            raise nx.NetworkXError(
(16)                f"{type(self).__name__} does not support slicing, "
(16)                f"try list(G.in_edges)[{e.start}:{e.stop}:{e.step}]"
(12)            )
(8)         u, v = e
(8)         return self._adjdict[v][u]
(0)     class OutMultiEdgeView(OutEdgeView):
(4)         """A EdgeView class for outward edges of a MultiDiGraph"""
(4)         __slots__ = ()
(4)         dataview = OutMultiEdgeDataView
(4)         def __len__(self):
(8)         return sum(
(12)            len(kdict) for n, nbrs in self._nodes_nbrs() for nbr, kdict in nbrs.items()
(8)         )
(4)         def __iter__(self):
(8)         for n, nbrs in self._nodes_nbrs():
(12)            for nbr, kdict in nbrs.items():
(16)                for key in kdict:
(20)                    yield (n, nbr, key)
(4)         def __contains__(self, e):
(8)         N = len(e)
(8)         if N == 3:
(12)            u, v, k = e
(8)         elif N == 2:
(12)            u, v = e
(12)            k = 0
(8)         else:
(12)            raise ValueError("MultiEdge must have length 2 or 3")
(8)         try:
(12)            return k in self._adjdict[u][v]
(8)         except KeyError:
(12)            return False
(4)         def __getitem__(self, e):
(8)         if isinstance(e, slice):
(12)            raise nx.NetworkXError(
(16)                f"{type(self).__name__} does not support slicing, "
(16)                f"try list(G.edges)[{e.start}:{e.stop}:{e.step}]"
(12)            )
(8)         u, v, k = e
(8)         return self._adjdict[u][v][k]
(4)         def __call__(self, nbunch=None, data=False, *, default=None, keys=False):
(8)         if nbunch is None and data is False and keys is True:
(12)            return self
(8)         return self.dataview(self, nbunch, data, default=default, keys=keys)
(4)         def data(self, data=True, default=None, nbunch=None, keys=False):
(8)         if nbunch is None and data is False and keys is True:
(12)            return self
(8)         return self.dataview(self, nbunch, data, default=default, keys=keys)
(0)     class MultiEdgeView(OutMultiEdgeView):
(4)         """A EdgeView class for edges of a MultiGraph"""
(4)         __slots__ = ()
(4)         dataview = MultiEdgeDataView
(4)         def __len__(self):
(8)         return sum(1 for e in self)
(4)         def __iter__(self):
(8)         seen = {}
(8)         for n, nbrs in self._nodes_nbrs():
(12)            for nbr, kd in nbrs.items():
(16)                if nbr not in seen:
(20)                    for k, dd in kd.items():
```

```
1139: (24)                                yield (n, nbr, k)
1140: (12)                    seen[n] = 1
1141: (8)                 del seen
1142: (0)         class InMultiEdgeView(OutMultiEdgeView):
1143: (4)             """A EdgeView class for inward edges of a MultiDiGraph"""
1144: (4)             __slots__ = ()
1145: (4)             def __setstate__(self, state):
1146: (8)                 self._graph = state["_graph"]
1147: (8)                 self._adjdict = state["_adjdict"]
1148: (8)                 self._nodes_nbrs = self._adjdict.items
1149: (4)             dataview = InMultiEdgeDataView
1150: (4)             def __init__(self, G):
1151: (8)                 self._graph = G
1152: (8)                 self._adjdict = G._pred if hasattr(G, "pred") else G._adj
1153: (8)                 self._nodes_nbrs = self._adjdict.items
1154: (4)             def __iter__(self):
1155: (8)                 for n, nbrs in self._nodes_nbrs():
1156: (12)                    for nbr, kdict in nbrs.items():
1157: (16)                        for key in kdict:
1158: (20)                            yield (nbr, n, key)
1159: (4)             def __contains__(self, e):
1160: (8)                 N = len(e)
1161: (8)                 if N == 3:
1162: (12)                    u, v, k = e
1163: (8)                 elif N == 2:
1164: (12)                    u, v = e
1165: (12)                    k = 0
1166: (8)                 else:
1167: (12)                    raise ValueError("MultiEdge must have length 2 or 3")
1168: (8)                 try:
1169: (12)                    return k in self._adjdict[v][u]
1170: (8)                 except KeyError:
1171: (12)                    return False
1172: (4)             def __getitem__(self, e):
1173: (8)                 if isinstance(e, slice):
1174: (12)                    raise nx.NetworkXError(
1175: (16)                        f"{type(self).__name__} does not support slicing, "
1176: (16)                        f"try list(G.in_edges)[{e.start}:{e.stop}:{e.step}]"
1177: (12)                    )
1178: (8)                 u, v, k = e
1179: (8)                 return self._adjdict[v][u][k]
```

File 398 - __init__.py:

```
1: (0)
```

File 399 - dispatch_interface.py:

```
1: (0)            import networkx as nx
2: (0)            from networkx import DiGraph, Graph, MultiDiGraph, MultiGraph, PlanarEmbedding
3: (0)            from networkx.classes.reportviews import NodeView
4: (0)            class LoopbackGraph(Graph):
5: (4)                __networkx_backend__ = "nx-loopback"
6: (0)            class LoopbackDiGraph(DiGraph):
7: (4)                __networkx_backend__ = "nx-loopback"
8: (0)            class LoopbackMultiGraph(MultiGraph):
9: (4)                __networkx_backend__ = "nx-loopback"
10: (0)           class LoopbackMultiDiGraph(MultiDiGraph):
11: (4)               __networkx_backend__ = "nx-loopback"
12: (0)           class LoopbackPlanarEmbedding(PlanarEmbedding):
13: (4)               __networkx_backend__ = "nx-loopback"
14: (0)           def convert(graph):
15: (4)               if isinstance(graph, PlanarEmbedding):
16: (8)                   return LoopbackPlanarEmbedding(graph)
17: (4)               if isinstance(graph, MultiDiGraph):
```

```
 18: (8)                     return LoopbackMultiDiGraph(graph)
 19: (4)                 if isinstance(graph, MultiGraph):
 20: (8)                     return LoopbackMultiGraph(graph)
 21: (4)                 if isinstance(graph, DiGraph):
 22: (8)                     return LoopbackDiGraph(graph)
 23: (4)                 if isinstance(graph, Graph):
 24: (8)                     return LoopbackGraph(graph)
 25: (4)                 raise TypeError(f"Unsupported type of graph: {type(graph)}")
 26: (0)             class LoopbackDispatcher:
 27: (4)                 def __getattr__(self, item):
 28: (8)                     try:
 29: (12)                        return nx.utils.backends._registered_algorithms[item].orig_func
 30: (8)                     except KeyError:
 31: (12)                        raise AttributeError(item) from None
 32: (4)                 @staticmethod
 33: (4)                 def convert_from_nx(
 34: (8)                     graph,
 35: (8)                     *,
 36: (8)                     edge_attrs=None,
 37: (8)                     node_attrs=None,
 38: (8)                     preserve_edge_attrs=None,
 39: (8)                     preserve_node_attrs=None,
 40: (8)                     preserve_graph_attrs=None,
 41: (8)                     name=None,
 42: (8)                     graph_name=None,
 43: (4)                 ):
 44: (8)                     if name in {
 45: (12)                        "lexicographical_topological_sort",
 46: (12)                        "topological_generations",
 47: (12)                        "topological_sort",
 48: (12)                        "dfs_labeled_edges",
 49: (8)                     }:
 50: (12)                        return graph
 51: (8)                     if isinstance(graph, NodeView):
 52: (12)                        new_graph = Graph()
 53: (12)                        new_graph.add_nodes_from(graph.items())
 54: (12)                        graph = new_graph
 55: (12)                        G = LoopbackGraph()
 56: (8)                     elif not isinstance(graph, Graph):
 57: (12)                        raise TypeError(
 58: (16)                            f"Bad type for graph argument {graph_name} in {name}:
{type(graph)}"
 59: (12)                        )
 60: (8)                     elif graph.__class__ in {Graph, LoopbackGraph}:
 61: (12)                        G = LoopbackGraph()
 62: (8)                     elif graph.__class__ in {DiGraph, LoopbackDiGraph}:
 63: (12)                        G = LoopbackDiGraph()
 64: (8)                     elif graph.__class__ in {MultiGraph, LoopbackMultiGraph}:
 65: (12)                        G = LoopbackMultiGraph()
 66: (8)                     elif graph.__class__ in {MultiDiGraph, LoopbackMultiDiGraph}:
 67: (12)                        G = LoopbackMultiDiGraph()
 68: (8)                     elif graph.__class__ in {PlanarEmbedding, LoopbackPlanarEmbedding}:
 69: (12)                        G = LoopbackDiGraph()  # or LoopbackPlanarEmbedding
 70: (8)                     else:
 71: (12)                        G = graph.__class__()
 72: (8)                     if preserve_graph_attrs:
 73: (12)                        G.graph.update(graph.graph)
 74: (8)                     if preserve_node_attrs:
 75: (12)                        G.add_nodes_from(graph.nodes(data=True))
 76: (8)                     elif node_attrs:
 77: (12)                        G.add_nodes_from(
 78: (16)                            (
 79: (20)                                node,
 80: (20)                                {
 81: (24)                                    k: datadict.get(k, default)
 82: (24)                                    for k, default in node_attrs.items()
 83: (24)                                    if default is not None or k in datadict
 84: (20)                                },
 85: (16)                            )
```

```
86: (16)                            for node, datadict in graph.nodes(data=True)
87: (12)                        )
88: (8)                     else:
89: (12)                        G.add_nodes_from(graph)
90: (8)                     if graph.is_multigraph():
91: (12)                        if preserve_edge_attrs:
92: (16)                            G.add_edges_from(
93: (20)                                (u, v, key, datadict)
94: (20)                                for u, nbrs in graph._adj.items()
95: (20)                                for v, keydict in nbrs.items()
96: (20)                                for key, datadict in keydict.items()
97: (16)                            )
98: (12)                        elif edge_attrs:
99: (16)                            G.add_edges_from(
100: (20)                               (
101: (24)                                   u,
102: (24)                                   v,
103: (24)                                   key,
104: (24)                                   {
105: (28)                                       k: datadict.get(k, default)
106: (28)                                       for k, default in edge_attrs.items()
107: (28)                                       if default is not None or k in datadict
108: (24)                                   },
109: (20)                               )
110: (20)                               for u, nbrs in graph._adj.items()
111: (20)                               for v, keydict in nbrs.items()
112: (20)                               for key, datadict in keydict.items()
113: (16)                           )
114: (12)                       else:
115: (16)                           G.add_edges_from(
116: (20)                               (u, v, key, {})
117: (20)                               for u, nbrs in graph._adj.items()
118: (20)                               for v, keydict in nbrs.items()
119: (20)                               for key, datadict in keydict.items()
120: (16)                           )
121: (8)                    elif preserve_edge_attrs:
122: (12)                       G.add_edges_from(graph.edges(data=True))
123: (8)                    elif edge_attrs:
124: (12)                       G.add_edges_from(
125: (16)                           (
126: (20)                               u,
127: (20)                               v,
128: (20)                               {
129: (24)                                   k: datadict.get(k, default)
130: (24)                                   for k, default in edge_attrs.items()
131: (24)                                   if default is not None or k in datadict
132: (20)                               },
133: (16)                           )
134: (16)                           for u, v, datadict in graph.edges(data=True)
135: (12)                       )
136: (8)                    else:
137: (12)                       G.add_edges_from(graph.edges)
138: (8)                    return G
139: (4)                @staticmethod
140: (4)                def convert_to_nx(obj, *, name=None):
141: (8)                    return obj
142: (4)                @staticmethod
143: (4)                def on_start_tests(items):
144: (8)                    for item in items:
145: (12)                       assert hasattr(item, "add_marker")
146: (4)                def can_run(self, name, args, kwargs):
147: (8)                    return hasattr(self, name)
148: (0)            dispatcher = LoopbackDispatcher()
```

File 400 - historical_tests.py:

```
1: (0)              """Original NetworkX graph tests"""
```

```
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx import convert_node_labels_to_integers as cnlti
5: (0)              from networkx.utils import edges_equal, nodes_equal
6: (0)              class HistoricalTests:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      cls.null = nx.null_graph()
10: (8)                     cls.P1 = cnlti(nx.path_graph(1), first_label=1)
11: (8)                     cls.P3 = cnlti(nx.path_graph(3), first_label=1)
12: (8)                     cls.P10 = cnlti(nx.path_graph(10), first_label=1)
13: (8)                     cls.K1 = cnlti(nx.complete_graph(1), first_label=1)
14: (8)                     cls.K3 = cnlti(nx.complete_graph(3), first_label=1)
15: (8)                     cls.K4 = cnlti(nx.complete_graph(4), first_label=1)
16: (8)                     cls.K5 = cnlti(nx.complete_graph(5), first_label=1)
17: (8)                     cls.K10 = cnlti(nx.complete_graph(10), first_label=1)
18: (8)                     cls.G = nx.Graph
19: (4)                 def test_name(self):
20: (8)                     G = self.G(name="test")
21: (8)                     assert G.name == "test"
22: (8)                     H = self.G()
23: (8)                     assert H.name == ""
24: (4)                 def test_add_remove_node(self):
25: (8)                     G = self.G()
26: (8)                     G.add_node("A")
27: (8)                     assert G.has_node("A")
28: (8)                     G.remove_node("A")
29: (8)                     assert not G.has_node("A")
30: (4)                 def test_nonhashable_node(self):
31: (8)                     G = self.G()
32: (8)                     assert not G.has_node(["A"])
33: (8)                     assert not G.has_node({"A": 1})
34: (4)                 def test_add_nodes_from(self):
35: (8)                     G = self.G()
36: (8)                     G.add_nodes_from(list("ABCDEFGHIJKL"))
37: (8)                     assert G.has_node("L")
38: (8)                     G.remove_nodes_from(["H", "I", "J", "K", "L"])
39: (8)                     G.add_nodes_from([1, 2, 3, 4])
40: (8)                     assert sorted(G.nodes(), key=str) == [
41: (12)                        1,
42: (12)                        2,
43: (12)                        3,
44: (12)                        4,
45: (12)                        "A",
46: (12)                        "B",
47: (12)                        "C",
48: (12)                        "D",
49: (12)                        "E",
50: (12)                        "F",
51: (12)                        "G",
52: (8)                     ]
53: (8)                     assert sorted(G, key=str) == [1, 2, 3, 4, "A", "B", "C", "D", "E",
"F", "G"]
54: (4)                 def test_contains(self):
55: (8)                     G = self.G()
56: (8)                     G.add_node("A")
57: (8)                     assert "A" in G
58: (8)                     assert [] not in G  # never raise a Key or TypeError in this test
59: (8)                     assert {1: 1} not in G
60: (4)                 def test_add_remove(self):
61: (8)                     G = self.G()
62: (8)                     G.add_node("m")
63: (8)                     assert G.has_node("m")
64: (8)                     G.add_node("m")  # no complaints
65: (8)                     pytest.raises(nx.NetworkXError, G.remove_node, "j")
66: (8)                     G.remove_node("m")
67: (8)                     assert list(G) == []
68: (4)                 def test_nbunch_is_list(self):
69: (8)                     G = self.G()
```

```
70: (8)                     G.add_nodes_from(list("ABCD"))
71: (8)                     G.add_nodes_from(self.P3)  # add nbunch of nodes (nbunch=Graph)
72: (8)                     assert sorted(G.nodes(), key=str) == [1, 2, 3, "A", "B", "C", "D"]
73: (8)                     G.remove_nodes_from(self.P3)  # remove nbunch of nodes (nbunch=Graph)
74: (8)                     assert sorted(G.nodes(), key=str) == ["A", "B", "C", "D"]
75: (4)                 def test_nbunch_is_set(self):
76: (8)                     G = self.G()
77: (8)                     nbunch = set("ABCDEFGHIJKL")
78: (8)                     G.add_nodes_from(nbunch)
79: (8)                     assert G.has_node("L")
80: (4)                 def test_nbunch_dict(self):
81: (8)                     G = self.G()
82: (8)                     nbunch = set("ABCDEFGHIJKL")
83: (8)                     G.add_nodes_from(nbunch)
84: (8)                     nbunch = {"I": "foo", "J": 2, "K": True, "L": "spam"}
85: (8)                     G.remove_nodes_from(nbunch)
86: (8)                     assert sorted(G.nodes(), key=str), ["A", "B", "C", "D", "E", "F", "G",
"H"]
87: (4)                 def test_nbunch_iterator(self):
88: (8)                     G = self.G()
89: (8)                     G.add_nodes_from(["A", "B", "C", "D", "E", "F", "G", "H"])
90: (8)                     n_iter = self.P3.nodes()
91: (8)                     G.add_nodes_from(n_iter)
92: (8)                     assert sorted(G.nodes(), key=str) == [
93: (12)                        1,
94: (12)                        2,
95: (12)                        3,
96: (12)                        "A",
97: (12)                        "B",
98: (12)                        "C",
99: (12)                        "D",
100: (12)                       "E",
101: (12)                       "F",
102: (12)                       "G",
103: (12)                       "H",
104: (8)                    ]
105: (8)                    n_iter = self.P3.nodes()  # rebuild same iterator
106: (8)                    G.remove_nodes_from(n_iter)  # remove nbunch of nodes
(nbunch=iterator)
107: (8)                    assert sorted(G.nodes(), key=str) == ["A", "B", "C", "D", "E", "F",
"G", "H"]
108: (4)                def test_nbunch_graph(self):
109: (8)                    G = self.G()
110: (8)                    G.add_nodes_from(["A", "B", "C", "D", "E", "F", "G", "H"])
111: (8)                    nbunch = self.K3
112: (8)                    G.add_nodes_from(nbunch)
113: (8)                    assert sorted(G.nodes(), key=str), [
114: (12)                       1,
115: (12)                       2,
116: (12)                       3,
117: (12)                       "A",
118: (12)                       "B",
119: (12)                       "C",
120: (12)                       "D",
121: (12)                       "E",
122: (12)                       "F",
123: (12)                       "G",
124: (12)                       "H",
125: (8)                    ]
126: (4)                def test_add_edge(self):
127: (8)                    G = self.G()
128: (8)                    pytest.raises(TypeError, G.add_edge, "A")
129: (8)                    G.add_edge("A", "B")  # testing add_edge()
130: (8)                    G.add_edge("A", "B")  # should fail silently
131: (8)                    assert G.has_edge("A", "B")
132: (8)                    assert not G.has_edge("A", "C")
133: (8)                    assert G.has_edge(*("A", "B"))
134: (8)                    if G.is_directed():
135: (12)                       assert not G.has_edge("B", "A")
```

```
(8)                     else:
(12)                        assert G.has_edge("B", "A")
(8)                     G.add_edge("A", "C")  # test directedness
(8)                     G.add_edge("C", "A")
(8)                     G.remove_edge("C", "A")
(8)                     if G.is_directed():
(12)                        assert G.has_edge("A", "C")
(8)                     else:
(12)                        assert not G.has_edge("A", "C")
(8)                     assert not G.has_edge("C", "A")
(4)                 def test_self_loop(self):
(8)                     G = self.G()
(8)                     G.add_edge("A", "A")  # test self loops
(8)                     assert G.has_edge("A", "A")
(8)                     G.remove_edge("A", "A")
(8)                     G.add_edge("X", "X")
(8)                     assert G.has_node("X")
(8)                     G.remove_node("X")
(8)                     G.add_edge("A", "Z")  # should add the node silently
(8)                     assert G.has_node("Z")
(4)                 def test_add_edges_from(self):
(8)                     G = self.G()
(8)                     G.add_edges_from([("B", "C")])  # test add_edges_from()
(8)                     assert G.has_edge("B", "C")
(8)                     if G.is_directed():
(12)                        assert not G.has_edge("C", "B")
(8)                     else:
(12)                        assert G.has_edge("C", "B")  # undirected
(8)                     G.add_edges_from([("D", "F"), ("B", "D")])
(8)                     assert G.has_edge("D", "F")
(8)                     assert G.has_edge("B", "D")
(8)                     if G.is_directed():
(12)                        assert not G.has_edge("D", "B")
(8)                     else:
(12)                        assert G.has_edge("D", "B")  # undirected
(4)                 def test_add_edges_from2(self):
(8)                     G = self.G()
(8)                     G.add_edges_from([tuple("IJ"), list("KK"), tuple("JK")])
(8)                     assert G.has_edge(*("I", "J"))
(8)                     assert G.has_edge(*("K", "K"))
(8)                     assert G.has_edge(*("J", "K"))
(8)                     if G.is_directed():
(12)                        assert not G.has_edge(*("K", "J"))
(8)                     else:
(12)                        assert G.has_edge(*("K", "J"))
(4)                 def test_add_edges_from3(self):
(8)                     G = self.G()
(8)                     G.add_edges_from(zip(list("ACD"), list("CDE")))
(8)                     assert G.has_edge("D", "E")
(8)                     assert not G.has_edge("E", "C")
(4)                 def test_remove_edge(self):
(8)                     G = self.G()
(8)                     G.add_nodes_from([1, 2, 3, "A", "B", "C", "D", "E", "F", "G", "H"])
(8)                     G.add_edges_from(zip(list("MNOP"), list("NOPM")))
(8)                     assert G.has_edge("O", "P")
(8)                     assert G.has_edge("P", "M")
(8)                     G.remove_node("P")  # tests remove_node()'s handling of edges.
(8)                     assert not G.has_edge("P", "M")
(8)                     pytest.raises(TypeError, G.remove_edge, "M")
(8)                     G.add_edge("N", "M")
(8)                     assert G.has_edge("M", "N")
(8)                     G.remove_edge("M", "N")
(8)                     assert not G.has_edge("M", "N")
(8)                     G.remove_edges_from([list("HI"), list("DF"), tuple("KK"),
tuple("JK")])
(8)                     assert not G.has_edge("H", "I")
(8)                     assert not G.has_edge("J", "K")
(8)                     G.remove_edges_from([list("IJ"), list("KK"), list("JK")])
(8)                     assert not G.has_edge("I", "J")
```

```
204: (8)                    G.remove_nodes_from(set("ZEFHIMNO"))
205: (8)                    G.add_edge("J", "K")
206: (4)                def test_edges_nbunch(self):
207: (8)                    G = self.G()
208: (8)                    G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"),
("C", "D")])
209: (8)                    pytest.raises(nx.NetworkXError, G.edges, 6)
210: (8)                    assert list(G.edges("Z")) == []  # iterable non-node
211: (8)                    assert list(G.edges([])) == []
212: (8)                    if G.is_directed():
213: (12)                       elist = [("A", "B"), ("A", "C"), ("B", "D")]
214: (8)                    else:
215: (12)                       elist = [("A", "B"), ("A", "C"), ("B", "C"), ("B", "D")]
216: (8)                    assert edges_equal(list(G.edges(["A", "B"])), elist)
217: (8)                    assert edges_equal(G.edges({"A", "B"}), elist)
218: (8)                    G1 = self.G()
219: (8)                    G1.add_nodes_from("AB")
220: (8)                    assert edges_equal(G.edges(G1), elist)
221: (8)                    ndict = {"A": "thing1", "B": "thing2"}
222: (8)                    assert edges_equal(G.edges(ndict), elist)
223: (8)                    assert edges_equal(list(G.edges("A")), [("A", "B"), ("A", "C")])
224: (8)                    assert nodes_equal(sorted(G), ["A", "B", "C", "D"])
225: (8)                    assert edges_equal(
226: (12)                       list(G.edges()),
227: (12)                       [("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"), ("C", "D")],
228: (8)                    )
229: (4)                def test_degree(self):
230: (8)                    G = self.G()
231: (8)                    G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"),
("C", "D")])
232: (8)                    assert G.degree("A") == 2
233: (8)                    assert list(G.degree(["A"])) == [("A", 2)]
234: (8)                    assert sorted(d for n, d in G.degree(["A", "B"])) == [2, 3]
235: (8)                    assert sorted(d for n, d in G.degree()) == [2, 2, 3, 3]
236: (4)                def test_degree2(self):
237: (8)                    H = self.G()
238: (8)                    H.add_edges_from([(1, 24), (1, 2)])
239: (8)                    assert sorted(d for n, d in H.degree([1, 24])) == [1, 2]
240: (4)                def test_degree_graph(self):
241: (8)                    P3 = nx.path_graph(3)
242: (8)                    P5 = nx.path_graph(5)
243: (8)                    assert dict(d for n, d in P3.degree(["A", "B"])) == {}
244: (8)                    assert sorted(d for n, d in P5.degree(P3)) == [1, 2, 2]
245: (8)                    assert sorted(d for n, d in P3.degree(P5)) == [1, 1, 2]
246: (8)                    assert list(P5.degree([])) == []
247: (8)                    assert dict(P5.degree([])) == {}
248: (4)                def test_null(self):
249: (8)                    null = nx.null_graph()
250: (8)                    assert list(null.degree()) == []
251: (8)                    assert dict(null.degree()) == {}
252: (4)                def test_order_size(self):
253: (8)                    G = self.G()
254: (8)                    G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"),
("C", "D")])
255: (8)                    assert G.order() == 4
256: (8)                    assert G.size() == 5
257: (8)                    assert G.number_of_edges() == 5
258: (8)                    assert G.number_of_edges("A", "B") == 1
259: (8)                    assert G.number_of_edges("A", "D") == 0
260: (4)                def test_copy(self):
261: (8)                    G = self.G()
262: (8)                    H = G.copy()  # copy
263: (8)                    assert H.adj == G.adj
264: (8)                    assert H.name == G.name
265: (8)                    assert H is not G
266: (4)                def test_subgraph(self):
267: (8)                    G = self.G()
268: (8)                    G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"),
("C", "D")])
```

```
(8)                 SG = G.subgraph(["A", "B", "D"])
(8)                 assert nodes_equal(list(SG), ["A", "B", "D"])
(8)                 assert edges_equal(list(SG.edges()), [("A", "B"), ("B", "D")])
(4)             def test_to_directed(self):
(8)                 G = self.G()
(8)                 if not G.is_directed():
(12)                    G.add_edges_from(
(16)                        [("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"), ("C", "D")]
(12)                    )
(12)                    DG = G.to_directed()
(12)                    assert DG is not G  # directed copy or copy
(12)                    assert DG.is_directed()
(12)                    assert DG.name == G.name
(12)                    assert DG.adj == G.adj
(12)                    assert sorted(DG.out_edges(list("AB"))) == [
(16)                        ("A", "B"),
(16)                        ("A", "C"),
(16)                        ("B", "A"),
(16)                        ("B", "C"),
(16)                        ("B", "D"),
(12)                    ]
(12)                    DG.remove_edge("A", "B")
(12)                    assert DG.has_edge("B", "A")  # this removes B-A but not  A-B
(12)                    assert not DG.has_edge("A", "B")
(4)             def test_to_undirected(self):
(8)                 G = self.G()
(8)                 if G.is_directed():
(12)                    G.add_edges_from(
(16)                        [("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"), ("C", "D")]
(12)                    )
(12)                    UG = G.to_undirected()  # to_undirected
(12)                    assert UG is not G
(12)                    assert not UG.is_directed()
(12)                    assert G.is_directed()
(12)                    assert UG.name == G.name
(12)                    assert UG.adj != G.adj
(12)                    assert sorted(UG.edges(list("AB"))) == [
(16)                        ("A", "B"),
(16)                        ("A", "C"),
(16)                        ("B", "C"),
(16)                        ("B", "D"),
(12)                    ]
(12)                    assert sorted(UG.edges(["A", "B"])) == [
(16)                        ("A", "B"),
(16)                        ("A", "C"),
(16)                        ("B", "C"),
(16)                        ("B", "D"),
(12)                    ]
(12)                    UG.remove_edge("A", "B")
(12)                    assert not UG.has_edge("B", "A")
(12)                    assert not UG.has_edge("A", "B")
(4)             def test_neighbors(self):
(8)                 G = self.G()
(8)                 G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"),
("C", "D")])
(8)                 G.add_nodes_from("GJK")
(8)                 assert sorted(G["A"]) == ["B", "C"]
(8)                 assert sorted(G.neighbors("A")) == ["B", "C"]
(8)                 assert sorted(G.neighbors("A")) == ["B", "C"]
(8)                 assert sorted(G.neighbors("G")) == []
(8)                 pytest.raises(nx.NetworkXError, G.neighbors, "j")
(4)             def test_iterators(self):
(8)                 G = self.G()
(8)                 G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"),
("C", "D")])
(8)                 G.add_nodes_from("GJK")
(8)                 assert sorted(G.nodes()) == ["A", "B", "C", "D", "G", "J", "K"]
(8)                 assert edges_equal(
(12)                    G.edges(), [("A", "B"), ("A", "C"), ("B", "D"), ("C", "B"), ("C",
```

```
"D")]
336: (8)                    )
337: (8)                    assert sorted(v for k, v in G.degree()) == [0, 0, 0, 2, 2, 3, 3]
338: (8)                    assert sorted(G.degree(), key=str) == [
339: (12)                       ("A", 2),
340: (12)                       ("B", 3),
341: (12)                       ("C", 3),
342: (12)                       ("D", 2),
343: (12)                       ("G", 0),
344: (12)                       ("J", 0),
345: (12)                       ("K", 0),
346: (8)                    ]
347: (8)                    assert sorted(G.neighbors("A")) == ["B", "C"]
348: (8)                    pytest.raises(nx.NetworkXError, G.neighbors, "X")
349: (8)                    G.clear()
350: (8)                    assert nx.number_of_nodes(G) == 0
351: (8)                    assert nx.number_of_edges(G) == 0
352: (4)                def test_null_subgraph(self):
353: (8)                    nullgraph = nx.null_graph()
354: (8)                    G = nx.null_graph()
355: (8)                    H = G.subgraph([])
356: (8)                    assert nx.is_isomorphic(H, nullgraph)
357: (4)                def test_empty_subgraph(self):
358: (8)                    nullgraph = nx.null_graph()
359: (8)                    E5 = nx.empty_graph(5)
360: (8)                    E10 = nx.empty_graph(10)
361: (8)                    H = E10.subgraph([])
362: (8)                    assert nx.is_isomorphic(H, nullgraph)
363: (8)                    H = E10.subgraph([1, 2, 3, 4, 5])
364: (8)                    assert nx.is_isomorphic(H, E5)
365: (4)                def test_complete_subgraph(self):
366: (8)                    K1 = nx.complete_graph(1)
367: (8)                    K3 = nx.complete_graph(3)
368: (8)                    K5 = nx.complete_graph(5)
369: (8)                    H = K5.subgraph([1, 2, 3])
370: (8)                    assert nx.is_isomorphic(H, K3)
371: (4)                def test_subgraph_nbunch(self):
372: (8)                    nullgraph = nx.null_graph()
373: (8)                    K1 = nx.complete_graph(1)
374: (8)                    K3 = nx.complete_graph(3)
375: (8)                    K5 = nx.complete_graph(5)
376: (8)                    H = K5.subgraph(1)
377: (8)                    assert nx.is_isomorphic(H, K1)
378: (8)                    H = K5.subgraph({1})
379: (8)                    assert nx.is_isomorphic(H, K1)
380: (8)                    H = K5.subgraph(iter(K3))
381: (8)                    assert nx.is_isomorphic(H, K3)
382: (8)                    H = K5.subgraph(K3)
383: (8)                    assert nx.is_isomorphic(H, K3)
384: (8)                    H = K5.subgraph([9])
385: (8)                    assert nx.is_isomorphic(H, nullgraph)
386: (4)                def test_node_tuple_issue(self):
387: (8)                    H = self.G()
388: (8)                    pytest.raises(nx.NetworkXError, H.remove_node, (1, 2))
389: (8)                    H.remove_nodes_from([(1, 2)])  # no error
390: (8)                    pytest.raises(nx.NetworkXError, H.neighbors, (1, 2))

----------------------------------------

File 401 - test_coreviews.py:

1: (0)              import pickle
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestAtlasView:
5: (4)                  def setup_method(self):
6: (8)                      self.d = {0: {"color": "blue", "weight": 1.2}, 1: {}, 2: {"color": 1}}
7: (8)                      self.av = nx.classes.coreviews.AtlasView(self.d)
8: (4)                  def test_pickle(self):
```

```
 9: (8)                     view = self.av
10: (8)                     pview = pickle.loads(pickle.dumps(view, -1))
11: (8)                     assert view == pview
12: (8)                     assert view.__slots__ == pview.__slots__
13: (8)                     pview = pickle.loads(pickle.dumps(view))
14: (8)                     assert view == pview
15: (8)                     assert view.__slots__ == pview.__slots__
16: (4)                 def test_len(self):
17: (8)                     assert len(self.av) == len(self.d)
18: (4)                 def test_iter(self):
19: (8)                     assert list(self.av) == list(self.d)
20: (4)                 def test_getitem(self):
21: (8)                     assert self.av[1] is self.d[1]
22: (8)                     assert self.av[2]["color"] == 1
23: (8)                     pytest.raises(KeyError, self.av.__getitem__, 3)
24: (4)                 def test_copy(self):
25: (8)                     avcopy = self.av.copy()
26: (8)                     assert avcopy[0] == self.av[0]
27: (8)                     assert avcopy == self.av
28: (8)                     assert avcopy[0] is not self.av[0]
29: (8)                     assert avcopy is not self.av
30: (8)                     avcopy[5] = {}
31: (8)                     assert avcopy != self.av
32: (8)                     avcopy[0]["ht"] = 4
33: (8)                     assert avcopy[0] != self.av[0]
34: (8)                     self.av[0]["ht"] = 4
35: (8)                     assert avcopy[0] == self.av[0]
36: (8)                     del self.av[0]["ht"]
37: (8)                     assert not hasattr(self.av, "__setitem__")
38: (4)                 def test_items(self):
39: (8)                     assert sorted(self.av.items()) == sorted(self.d.items())
40: (4)                 def test_str(self):
41: (8)                     out = str(self.d)
42: (8)                     assert str(self.av) == out
43: (4)                 def test_repr(self):
44: (8)                     out = "AtlasView(" + str(self.d) + ")"
45: (8)                     assert repr(self.av) == out
46: (0)             class TestAdjacencyView:
47: (4)                 def setup_method(self):
48: (8)                     dd = {"color": "blue", "weight": 1.2}
49: (8)                     self.nd = {0: dd, 1: {}, 2: {"color": 1}}
50: (8)                     self.adj = {3: self.nd, 0: {3: dd}, 1: {}, 2: {3: {"color": 1}}}
51: (8)                     self.adjview = nx.classes.coreviews.AdjacencyView(self.adj)
52: (4)                 def test_pickle(self):
53: (8)                     view = self.adjview
54: (8)                     pview = pickle.loads(pickle.dumps(view, -1))
55: (8)                     assert view == pview
56: (8)                     assert view.__slots__ == pview.__slots__
57: (4)                 def test_len(self):
58: (8)                     assert len(self.adjview) == len(self.adj)
59: (4)                 def test_iter(self):
60: (8)                     assert list(self.adjview) == list(self.adj)
61: (4)                 def test_getitem(self):
62: (8)                     assert self.adjview[1] is not self.adj[1]
63: (8)                     assert self.adjview[3][0] is self.adjview[0][3]
64: (8)                     assert self.adjview[2][3]["color"] == 1
65: (8)                     pytest.raises(KeyError, self.adjview.__getitem__, 4)
66: (4)                 def test_copy(self):
67: (8)                     avcopy = self.adjview.copy()
68: (8)                     assert avcopy[0] == self.adjview[0]
69: (8)                     assert avcopy[0] is not self.adjview[0]
70: (8)                     avcopy[2][3]["ht"] = 4
71: (8)                     assert avcopy[2] != self.adjview[2]
72: (8)                     self.adjview[2][3]["ht"] = 4
73: (8)                     assert avcopy[2] == self.adjview[2]
74: (8)                     del self.adjview[2][3]["ht"]
75: (8)                     assert not hasattr(self.adjview, "__setitem__")
76: (4)                 def test_items(self):
77: (8)                     view_items = sorted((n, dict(d)) for n, d in self.adjview.items())
```

```
 78: (8)                    assert view_items == sorted(self.adj.items())
 79: (4)                def test_str(self):
 80: (8)                    out = str(dict(self.adj))
 81: (8)                    assert str(self.adjview) == out
 82: (4)                def test_repr(self):
 83: (8)                    out = self.adjview.__class__.__name__ + "(" + str(self.adj) + ")"
 84: (8)                    assert repr(self.adjview) == out
 85: (0)            class TestMultiAdjacencyView(TestAdjacencyView):
 86: (4)                def setup_method(self):
 87: (8)                    dd = {"color": "blue", "weight": 1.2}
 88: (8)                    self.kd = {0: dd, 1: {}, 2: {"color": 1}}
 89: (8)                    self.nd = {3: self.kd, 0: {3: dd}, 1: {0: {}}, 2: {3: {"color": 1}}}
 90: (8)                    self.adj = {3: self.nd, 0: {3: {3: dd}}, 1: {}, 2: {3: {8: {}}}}
 91: (8)                    self.adjview = nx.classes.coreviews.MultiAdjacencyView(self.adj)
 92: (4)                def test_getitem(self):
 93: (8)                    assert self.adjview[1] is not self.adj[1]
 94: (8)                    assert self.adjview[3][0][3] is self.adjview[0][3][3]
 95: (8)                    assert self.adjview[3][2][3]["color"] == 1
 96: (8)                    pytest.raises(KeyError, self.adjview.__getitem__, 4)
 97: (4)                def test_copy(self):
 98: (8)                    avcopy = self.adjview.copy()
 99: (8)                    assert avcopy[0] == self.adjview[0]
100: (8)                    assert avcopy[0] is not self.adjview[0]
101: (8)                    avcopy[2][3][8]["ht"] = 4
102: (8)                    assert avcopy[2] != self.adjview[2]
103: (8)                    self.adjview[2][3][8]["ht"] = 4
104: (8)                    assert avcopy[2] == self.adjview[2]
105: (8)                    del self.adjview[2][3][8]["ht"]
106: (8)                    assert not hasattr(self.adjview, "__setitem__")
107: (0)            class TestUnionAtlas:
108: (4)                def setup_method(self):
109: (8)                    self.s = {0: {"color": "blue", "weight": 1.2}, 1: {}, 2: {"color": 1}}
110: (8)                    self.p = {3: {"color": "blue", "weight": 1.2}, 4: {}, 2: {"watch": 2}}
111: (8)                    self.av = nx.classes.coreviews.UnionAtlas(self.s, self.p)
112: (4)                def test_pickle(self):
113: (8)                    view = self.av
114: (8)                    pview = pickle.loads(pickle.dumps(view, -1))
115: (8)                    assert view == pview
116: (8)                    assert view.__slots__ == pview.__slots__
117: (4)                def test_len(self):
118: (8)                    assert len(self.av) == len(self.s.keys() | self.p.keys()) == 5
119: (4)                def test_iter(self):
120: (8)                    assert set(self.av) == set(self.s) | set(self.p)
121: (4)                def test_getitem(self):
122: (8)                    assert self.av[0] is self.s[0]
123: (8)                    assert self.av[4] is self.p[4]
124: (8)                    assert self.av[2]["color"] == 1
125: (8)                    pytest.raises(KeyError, self.av[2].__getitem__, "watch")
126: (8)                    pytest.raises(KeyError, self.av.__getitem__, 8)
127: (4)                def test_copy(self):
128: (8)                    avcopy = self.av.copy()
129: (8)                    assert avcopy[0] == self.av[0]
130: (8)                    assert avcopy[0] is not self.av[0]
131: (8)                    assert avcopy is not self.av
132: (8)                    avcopy[5] = {}
133: (8)                    assert avcopy != self.av
134: (8)                    avcopy[0]["ht"] = 4
135: (8)                    assert avcopy[0] != self.av[0]
136: (8)                    self.av[0]["ht"] = 4
137: (8)                    assert avcopy[0] == self.av[0]
138: (8)                    del self.av[0]["ht"]
139: (8)                    assert not hasattr(self.av, "__setitem__")
140: (4)                def test_items(self):
141: (8)                    expected = dict(self.p.items())
142: (8)                    expected.update(self.s)
143: (8)                    assert sorted(self.av.items()) == sorted(expected.items())
144: (4)                def test_str(self):
145: (8)                    out = str(dict(self.av))
146: (8)                    assert str(self.av) == out
```

```
(4)     def test_repr(self):
(8)         out = f"{self.av.__class__.__name__}({self.s}, {self.p})"
(8)         assert repr(self.av) == out
(0) class TestUnionAdjacency:
(4)     def setup_method(self):
(8)         dd = {"color": "blue", "weight": 1.2}
(8)         self.nd = {0: dd, 1: {}, 2: {"color": 1}}
(8)         self.s = {3: self.nd, 0: {}, 1: {}, 2: {3: {"color": 1}}}
(8)         self.p = {3: {}, 0: {3: dd}, 1: {0: {}}, 2: {1: {"color": 1}}}
(8)         self.adjview = nx.classes.coreviews.UnionAdjacency(self.s, self.p)
(4)     def test_pickle(self):
(8)         view = self.adjview
(8)         pview = pickle.loads(pickle.dumps(view, -1))
(8)         assert view == pview
(8)         assert view.__slots__ == pview.__slots__
(4)     def test_len(self):
(8)         assert len(self.adjview) == len(self.s)
(4)     def test_iter(self):
(8)         assert sorted(self.adjview) == sorted(self.s)
(4)     def test_getitem(self):
(8)         assert self.adjview[1] is not self.s[1]
(8)         assert self.adjview[3][0] is self.adjview[0][3]
(8)         assert self.adjview[2][3]["color"] == 1
(8)         pytest.raises(KeyError, self.adjview.__getitem__, 4)
(4)     def test_copy(self):
(8)         avcopy = self.adjview.copy()
(8)         assert avcopy[0] == self.adjview[0]
(8)         assert avcopy[0] is not self.adjview[0]
(8)         avcopy[2][3]["ht"] = 4
(8)         assert avcopy[2] != self.adjview[2]
(8)         self.adjview[2][3]["ht"] = 4
(8)         assert avcopy[2] == self.adjview[2]
(8)         del self.adjview[2][3]["ht"]
(8)         assert not hasattr(self.adjview, "__setitem__")
(4)     def test_str(self):
(8)         out = str(dict(self.adjview))
(8)         assert str(self.adjview) == out
(4)     def test_repr(self):
(8)         clsname = self.adjview.__class__.__name__
(8)         out = f"{clsname}({self.s}, {self.p})"
(8)         assert repr(self.adjview) == out
(0) class TestUnionMultiInner(TestUnionAdjacency):
(4)     def setup_method(self):
(8)         dd = {"color": "blue", "weight": 1.2}
(8)         self.kd = {7: {}, "ekey": {}, 9: {"color": 1}}
(8)         self.s = {3: self.kd, 0: {7: dd}, 1: {}, 2: {"key": {"color": 1}}}
(8)         self.p = {3: {}, 0: {3: dd}, 1: {}, 2: {1: {"span": 2}}}
(8)         self.adjview = nx.classes.coreviews.UnionMultiInner(self.s, self.p)
(4)     def test_len(self):
(8)         assert len(self.adjview) == len(self.s.keys() | self.p.keys()) == 4
(4)     def test_getitem(self):
(8)         assert self.adjview[1] is not self.s[1]
(8)         assert self.adjview[0][7] is self.adjview[0][3]
(8)         assert self.adjview[2]["key"]["color"] == 1
(8)         assert self.adjview[2][1]["span"] == 2
(8)         pytest.raises(KeyError, self.adjview.__getitem__, 4)
(8)         pytest.raises(KeyError, self.adjview[1].__getitem__, "key")
(4)     def test_copy(self):
(8)         avcopy = self.adjview.copy()
(8)         assert avcopy[0] == self.adjview[0]
(8)         assert avcopy[0] is not self.adjview[0]
(8)         avcopy[2][1]["width"] = 8
(8)         assert avcopy[2] != self.adjview[2]
(8)         self.adjview[2][1]["width"] = 8
(8)         assert avcopy[2] == self.adjview[2]
(8)         del self.adjview[2][1]["width"]
(8)         assert not hasattr(self.adjview, "__setitem__")
(8)         assert hasattr(avcopy, "__setitem__")
(0) class TestUnionMultiAdjacency(TestUnionAdjacency):
```

```
(4)     def setup_method(self):
(8)         dd = {"color": "blue", "weight": 1.2}
(8)         self.kd = {7: {}, 8: {}, 9: {"color": 1}}
(8)         self.nd = {3: self.kd, 0: {9: dd}, 1: {8: {}}, 2: {9: {"color": 1}}}
(8)         self.s = {3: self.nd, 0: {3: {7: dd}}, 1: {}, 2: {3: {8: {}}}}
(8)         self.p = {3: {}, 0: {3: {9: dd}}, 1: {}, 2: {1: {8: {}}}}
(8)         self.adjview = nx.classes.coreviews.UnionMultiAdjacency(self.s,
self.p)
(4)     def test_getitem(self):
(8)         assert self.adjview[1] is not self.s[1]
(8)         assert self.adjview[3][0][9] is self.adjview[0][3][9]
(8)         assert self.adjview[3][2][9]["color"] == 1
(8)         pytest.raises(KeyError, self.adjview.__getitem__, 4)
(4)     def test_copy(self):
(8)         avcopy = self.adjview.copy()
(8)         assert avcopy[0] == self.adjview[0]
(8)         assert avcopy[0] is not self.adjview[0]
(8)         avcopy[2][3][8]["ht"] = 4
(8)         assert avcopy[2] != self.adjview[2]
(8)         self.adjview[2][3][8]["ht"] = 4
(8)         assert avcopy[2] == self.adjview[2]
(8)         del self.adjview[2][3][8]["ht"]
(8)         assert not hasattr(self.adjview, "__setitem__")
(8)         assert hasattr(avcopy, "__setitem__")
(0)     class TestFilteredGraphs:
(4)     def setup_method(self):
(8)         self.Graphs = [nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph]
(4)     def test_hide_show_nodes(self):
(8)         SubGraph = nx.subgraph_view
(8)         for Graph in self.Graphs:
(12)            G = nx.path_graph(4, Graph)
(12)            SG = G.subgraph([2, 3])
(12)            RG = SubGraph(G, filter_node=nx.filters.hide_nodes([0, 1]))
(12)            assert SG.nodes == RG.nodes
(12)            assert SG.edges == RG.edges
(12)            SGC = SG.copy()
(12)            RGC = RG.copy()
(12)            assert SGC.nodes == RGC.nodes
(12)            assert SGC.edges == RGC.edges
(4)     def test_str_repr(self):
(8)         SubGraph = nx.subgraph_view
(8)         for Graph in self.Graphs:
(12)            G = nx.path_graph(4, Graph)
(12)            SG = G.subgraph([2, 3])
(12)            RG = SubGraph(G, filter_node=nx.filters.hide_nodes([0, 1]))
(12)            str(SG.adj)
(12)            str(RG.adj)
(12)            repr(SG.adj)
(12)            repr(RG.adj)
(12)            str(SG.adj[2])
(12)            str(RG.adj[2])
(12)            repr(SG.adj[2])
(12)            repr(RG.adj[2])
(4)     def test_copy(self):
(8)         SubGraph = nx.subgraph_view
(8)         for Graph in self.Graphs:
(12)            G = nx.path_graph(4, Graph)
(12)            SG = G.subgraph([2, 3])
(12)            RG = SubGraph(G, filter_node=nx.filters.hide_nodes([0, 1]))
(12)            RsG = SubGraph(G, filter_node=nx.filters.show_nodes([2, 3]))
(12)            assert G.adj.copy() == G.adj
(12)            assert G.adj[2].copy() == G.adj[2]
(12)            assert SG.adj.copy() == SG.adj
(12)            assert SG.adj[2].copy() == SG.adj[2]
(12)            assert RG.adj.copy() == RG.adj
(12)            assert RG.adj[2].copy() == RG.adj[2]
(12)            assert RsG.adj.copy() == RsG.adj
(12)            assert RsG.adj[2].copy() == RsG.adj[2]
```

```
----------------------------------------

File 402 - test_digraph.py:

import pytest
import networkx as nx
from networkx.utils import nodes_equal
from .test_graph import BaseAttrGraphTester, BaseGraphTester
from .test_graph import TestEdgeSubgraph as _TestGraphEdgeSubgraph
from .test_graph import TestGraph as _TestGraph
class BaseDiGraphTester(BaseGraphTester):
    def test_has_successor(self):
        G = self.K3
        assert G.has_successor(0, 1)
        assert not G.has_successor(0, -1)
    def test_successors(self):
        G = self.K3
        assert sorted(G.successors(0)) == [1, 2]
        with pytest.raises(nx.NetworkXError):
            G.successors(-1)
    def test_has_predecessor(self):
        G = self.K3
        assert G.has_predecessor(0, 1)
        assert not G.has_predecessor(0, -1)
    def test_predecessors(self):
        G = self.K3
        assert sorted(G.predecessors(0)) == [1, 2]
        with pytest.raises(nx.NetworkXError):
            G.predecessors(-1)
    def test_edges(self):
        G = self.K3
        assert sorted(G.edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
        assert sorted(G.edges(0)) == [(0, 1), (0, 2)]
        assert sorted(G.edges([0, 1])) == [(0, 1), (0, 2), (1, 0), (1, 2)]
        with pytest.raises(nx.NetworkXError):
            G.edges(-1)
    def test_out_edges(self):
        G = self.K3
        assert sorted(G.out_edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
        assert sorted(G.out_edges(0)) == [(0, 1), (0, 2)]
        with pytest.raises(nx.NetworkXError):
            G.out_edges(-1)
    def test_out_edges_dir(self):
        G = self.P3
        assert sorted(G.out_edges()) == [(0, 1), (1, 2)]
        assert sorted(G.out_edges(0)) == [(0, 1)]
        assert sorted(G.out_edges(2)) == []
    def test_out_edges_data(self):
        G = nx.DiGraph([(0, 1, {"data": 0}), (1, 0, {})])
        assert sorted(G.out_edges(data=True)) == [(0, 1, {"data": 0}), (1, 0, {})]
        assert sorted(G.out_edges(0, data=True)) == [(0, 1, {"data": 0})]
        assert sorted(G.out_edges(data="data")) == [(0, 1, 0), (1, 0, None)]
        assert sorted(G.out_edges(0, data="data")) == [(0, 1, 0)]
    def test_in_edges_dir(self):
        G = self.P3
        assert sorted(G.in_edges()) == [(0, 1), (1, 2)]
        assert sorted(G.in_edges(0)) == []
        assert sorted(G.in_edges(2)) == [(1, 2)]
    def test_in_edges_data(self):
        G = nx.DiGraph([(0, 1, {"data": 0}), (1, 0, {})])
        assert sorted(G.in_edges(data=True)) == [(0, 1, {"data": 0}), (1, 0, {})]
        assert sorted(G.in_edges(1, data=True)) == [(0, 1, {"data": 0})]
        assert sorted(G.in_edges(data="data")) == [(0, 1, 0), (1, 0, None)]
        assert sorted(G.in_edges(1, data="data")) == [(0, 1, 0)]
    def test_degree(self):
```

```
 62: (8)                     G = self.K3
 63: (8)                     assert sorted(G.degree()) == [(0, 4), (1, 4), (2, 4)]
 64: (8)                     assert dict(G.degree()) == {0: 4, 1: 4, 2: 4}
 65: (8)                     assert G.degree(0) == 4
 66: (8)                     assert list(G.degree(iter([0]))) == [(0, 4)]  # run through iterator
 67: (4)                 def test_in_degree(self):
 68: (8)                     G = self.K3
 69: (8)                     assert sorted(G.in_degree()) == [(0, 2), (1, 2), (2, 2)]
 70: (8)                     assert dict(G.in_degree()) == {0: 2, 1: 2, 2: 2}
 71: (8)                     assert G.in_degree(0) == 2
 72: (8)                     assert list(G.in_degree(iter([0]))) == [(0, 2)]  # run through
iterator
 73: (4)                 def test_out_degree(self):
 74: (8)                     G = self.K3
 75: (8)                     assert sorted(G.out_degree()) == [(0, 2), (1, 2), (2, 2)]
 76: (8)                     assert dict(G.out_degree()) == {0: 2, 1: 2, 2: 2}
 77: (8)                     assert G.out_degree(0) == 2
 78: (8)                     assert list(G.out_degree(iter([0]))) == [(0, 2)]
 79: (4)                 def test_size(self):
 80: (8)                     G = self.K3
 81: (8)                     assert G.size() == 6
 82: (8)                     assert G.number_of_edges() == 6
 83: (4)                 def test_to_undirected_reciprocal(self):
 84: (8)                     G = self.Graph()
 85: (8)                     G.add_edge(1, 2)
 86: (8)                     assert G.to_undirected().has_edge(1, 2)
 87: (8)                     assert not G.to_undirected(reciprocal=True).has_edge(1, 2)
 88: (8)                     G.add_edge(2, 1)
 89: (8)                     assert G.to_undirected(reciprocal=True).has_edge(1, 2)
 90: (4)                 def test_reverse_copy(self):
 91: (8)                     G = nx.DiGraph([(0, 1), (1, 2)])
 92: (8)                     R = G.reverse()
 93: (8)                     assert sorted(R.edges()) == [(1, 0), (2, 1)]
 94: (8)                     R.remove_edge(1, 0)
 95: (8)                     assert sorted(R.edges()) == [(2, 1)]
 96: (8)                     assert sorted(G.edges()) == [(0, 1), (1, 2)]
 97: (4)                 def test_reverse_nocopy(self):
 98: (8)                     G = nx.DiGraph([(0, 1), (1, 2)])
 99: (8)                     R = G.reverse(copy=False)
 100: (8)                    assert sorted(R.edges()) == [(1, 0), (2, 1)]
 101: (8)                    with pytest.raises(nx.NetworkXError):
 102: (12)                       R.remove_edge(1, 0)
 103: (4)                dd def test_reverse_hashable(self):
 104: (8)                    class Foo:
 105: (12)                       pass
 106: (8)                    x = Foo()
 107: (8)                    y = Foo()
 108: (8)                    G = nx.DiGraph()
 109: (8)                    G.add_edge(x, y)
 110: (8)                    assert nodes_equal(G.nodes(), G.reverse().nodes())
 111: (8)                    assert [(y, x)] == list(G.reverse().edges())
 112: (4)                def test_di_cache_reset(self):
 113: (8)                    G = self.K3.copy()
 114: (8)                    old_succ = G.succ
 115: (8)                    assert id(G.succ) == id(old_succ)
 116: (8)                    old_adj = G.adj
 117: (8)                    assert id(G.adj) == id(old_adj)
 118: (8)                    G._succ = {}
 119: (8)                    assert id(G.succ) != id(old_succ)
 120: (8)                    assert id(G.adj) != id(old_adj)
 121: (8)                    old_pred = G.pred
 122: (8)                    assert id(G.pred) == id(old_pred)
 123: (8)                    G._pred = {}
 124: (8)                    assert id(G.pred) != id(old_pred)
 125: (4)                def test_di_attributes_cached(self):
 126: (8)                    G = self.K3.copy()
 127: (8)                    assert id(G.in_edges) == id(G.in_edges)
 128: (8)                    assert id(G.out_edges) == id(G.out_edges)
 129: (8)                    assert id(G.in_degree) == id(G.in_degree)
```

```
130: (8)                    assert id(G.out_degree) == id(G.out_degree)
131: (8)                    assert id(G.succ) == id(G.succ)
132: (8)                    assert id(G.pred) == id(G.pred)
133: (0)            class BaseAttrDiGraphTester(BaseDiGraphTester, BaseAttrGraphTester):
134: (4)                def test_edges_data(self):
135: (8)                    G = self.K3
136: (8)                    all_edges = [
137: (12)                       (0, 1, {}),
138: (12)                       (0, 2, {}),
139: (12)                       (1, 0, {}),
140: (12)                       (1, 2, {}),
141: (12)                       (2, 0, {}),
142: (12)                       (2, 1, {}),
143: (8)                    ]
144: (8)                    assert sorted(G.edges(data=True)) == all_edges
145: (8)                    assert sorted(G.edges(0, data=True)) == all_edges[:2]
146: (8)                    assert sorted(G.edges([0, 1], data=True)) == all_edges[:4]
147: (8)                    with pytest.raises(nx.NetworkXError):
148: (12)                       G.edges(-1, True)
149: (4)                def test_in_degree_weighted(self):
150: (8)                    G = self.K3.copy()
151: (8)                    G.add_edge(0, 1, weight=0.3, other=1.2)
152: (8)                    assert sorted(G.in_degree(weight="weight")) == [(0, 2), (1, 1.3), (2,
2)]
153: (8)                    assert dict(G.in_degree(weight="weight")) == {0: 2, 1: 1.3, 2: 2}
154: (8)                    assert G.in_degree(1, weight="weight") == 1.3
155: (8)                    assert sorted(G.in_degree(weight="other")) == [(0, 2), (1, 2.2), (2,
2)]
156: (8)                    assert dict(G.in_degree(weight="other")) == {0: 2, 1: 2.2, 2: 2}
157: (8)                    assert G.in_degree(1, weight="other") == 2.2
158: (8)                    assert list(G.in_degree(iter([1]), weight="other")) == [(1, 2.2)]
159: (4)                def test_out_degree_weighted(self):
160: (8)                    G = self.K3.copy()
161: (8)                    G.add_edge(0, 1, weight=0.3, other=1.2)
162: (8)                    assert sorted(G.out_degree(weight="weight")) == [(0, 1.3), (1, 2), (2,
2)]
163: (8)                    assert dict(G.out_degree(weight="weight")) == {0: 1.3, 1: 2, 2: 2}
164: (8)                    assert G.out_degree(0, weight="weight") == 1.3
165: (8)                    assert sorted(G.out_degree(weight="other")) == [(0, 2.2), (1, 2), (2,
2)]
166: (8)                    assert dict(G.out_degree(weight="other")) == {0: 2.2, 1: 2, 2: 2}
167: (8)                    assert G.out_degree(0, weight="other") == 2.2
168: (8)                    assert list(G.out_degree(iter([0]), weight="other")) == [(0, 2.2)]
169: (0)            class TestDiGraph(BaseAttrDiGraphTester, _TestGraph):
170: (4)                """Tests specific to dict-of-dict-of-dict digraph data structure"""
171: (4)                def setup_method(self):
172: (8)                    self.Graph = nx.DiGraph
173: (8)                    ed1, ed2, ed3, ed4, ed5, ed6 = ({}, {}, {}, {}, {}, {})
174: (8)                    self.k3adj = {0: {1: ed1, 2: ed2}, 1: {0: ed3, 2: ed4}, 2: {0: ed5, 1:
ed6}}
175: (8)                    self.k3edges = [(0, 1), (0, 2), (1, 2)]
176: (8)                    self.k3nodes = [0, 1, 2]
177: (8)                    self.K3 = self.Graph()
178: (8)                    self.K3._succ = self.k3adj  # K3._adj is synced with K3._succ
179: (8)                    self.K3._pred = {0: {1: ed3, 2: ed5}, 1: {0: ed1, 2: ed6}, 2: {0: ed2,
1: ed4}}
180: (8)                    self.K3._node = {}
181: (8)                    self.K3._node[0] = {}
182: (8)                    self.K3._node[1] = {}
183: (8)                    self.K3._node[2] = {}
184: (8)                    ed1, ed2 = ({}, {})
185: (8)                    self.P3 = self.Graph()
186: (8)                    self.P3._succ = {0: {1: ed1}, 1: {2: ed2}, 2: {}}
187: (8)                    self.P3._pred = {0: {}, 1: {0: ed1}, 2: {1: ed2}}
188: (8)                    self.P3._node = {}
189: (8)                    self.P3._node[0] = {}
190: (8)                    self.P3._node[1] = {}
191: (8)                    self.P3._node[2] = {}
192: (4)                def test_data_input(self):
```

```
(8)                 G = self.Graph({1: [2], 2: [1]}, name="test")
(8)                 assert G.name == "test"
(8)                 assert sorted(G.adj.items()) == [(1, {2: {}}), (2, {1: {}})]
(8)                 assert sorted(G.succ.items()) == [(1, {2: {}}), (2, {1: {}})]
(8)                 assert sorted(G.pred.items()) == [(1, {2: {}}), (2, {1: {}})]
(4)             def test_add_edge(self):
(8)                 G = self.Graph()
(8)                 G.add_edge(0, 1)
(8)                 assert G.adj == {0: {1: {}}, 1: {}}
(8)                 assert G.succ == {0: {1: {}}, 1: {}}
(8)                 assert G.pred == {0: {}, 1: {0: {}}}
(8)                 G = self.Graph()
(8)                 G.add_edge(*(0, 1))
(8)                 assert G.adj == {0: {1: {}}, 1: {}}
(8)                 assert G.succ == {0: {1: {}}, 1: {}}
(8)                 assert G.pred == {0: {}, 1: {0: {}}}
(8)                 with pytest.raises(ValueError, match="None cannot be a node"):
(12)                    G.add_edge(None, 3)
(4)             def test_add_edges_from(self):
(8)                 G = self.Graph()
(8)                 G.add_edges_from([(0, 1), (0, 2, {"data": 3})], data=2)
(8)                 assert G.adj == {0: {1: {"data": 2}, 2: {"data": 3}}, 1: {}, 2: {}}
(8)                 assert G.succ == {0: {1: {"data": 2}, 2: {"data": 3}}, 1: {}, 2: {}}
(8)                 assert G.pred == {0: {}, 1: {0: {"data": 2}}, 2: {0: {"data": 3}}}
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.add_edges_from([(0,)])  # too few in tuple
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.add_edges_from([(0, 1, 2, 3)])  # too many in tuple
(8)                 with pytest.raises(TypeError):
(12)                    G.add_edges_from([0])  # not a tuple
(8)                 with pytest.raises(ValueError, match="None cannot be a node"):
(12)                    G.add_edges_from([(None, 3), (3, 2)])
(4)             def test_remove_edge(self):
(8)                 G = self.K3.copy()
(8)                 G.remove_edge(0, 1)
(8)                 assert G.succ == {0: {2: {}}, 1: {0: {}, 2: {}}, 2: {0: {}, 1: {}}}
(8)                 assert G.pred == {0: {1: {}, 2: {}}, 1: {2: {}}, 2: {0: {}, 1: {}}}
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.remove_edge(-1, 0)
(4)             def test_remove_edges_from(self):
(8)                 G = self.K3.copy()
(8)                 G.remove_edges_from([(0, 1)])
(8)                 assert G.succ == {0: {2: {}}, 1: {0: {}, 2: {}}, 2: {0: {}, 1: {}}}
(8)                 assert G.pred == {0: {1: {}, 2: {}}, 1: {2: {}}, 2: {0: {}, 1: {}}}
(8)                 G.remove_edges_from([(0, 0)])  # silent fail
(4)             def test_clear(self):
(8)                 G = self.K3
(8)                 G.graph["name"] = "K3"
(8)                 G.clear()
(8)                 assert list(G.nodes) == []
(8)                 assert G.succ == {}
(8)                 assert G.pred == {}
(8)                 assert G.graph == {}
(4)             def test_clear_edges(self):
(8)                 G = self.K3
(8)                 G.graph["name"] = "K3"
(8)                 nodes = list(G.nodes)
(8)                 G.clear_edges()
(8)                 assert list(G.nodes) == nodes
(8)                 expected = {0: {}, 1: {}, 2: {}}
(8)                 assert G.succ == expected
(8)                 assert G.pred == expected
(8)                 assert list(G.edges) == []
(8)                 assert G.graph["name"] == "K3"
(0)         class TestEdgeSubgraph(_TestGraphEdgeSubgraph):
(4)             """Unit tests for the :meth:`DiGraph.edge_subgraph` method."""
(4)             def setup_method(self):
(8)                 G = nx.DiGraph(nx.path_graph(5))
(8)                 for i in range(5):
```

```
262: (12)                        G.nodes[i]["name"] = f"node{i}"
263: (8)                     G.edges[0, 1]["name"] = "edge01"
264: (8)                     G.edges[3, 4]["name"] = "edge34"
265: (8)                     G.graph["name"] = "graph"
266: (8)                     self.G = G
267: (8)                     self.H = G.edge_subgraph([(0, 1), (3, 4)])
268: (4)                 def test_pred_succ(self):
269: (8)                     """Test that nodes are added to predecessors and successors.
270: (8)                     For more information, see GitHub issue #2370.
271: (8)                     """
272: (8)                     G = nx.DiGraph()
273: (8)                     G.add_edge(0, 1)
274: (8)                     H = G.edge_subgraph([(0, 1)])
275: (8)                     assert list(H.predecessors(0)) == []
276: (8)                     assert list(H.successors(0)) == [1]
277: (8)                     assert list(H.predecessors(1)) == [0]
278: (8)                     assert list(H.successors(1)) == []
```

File 403 - test_filters.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestFilterFactory:
4: (4)                  def test_no_filter(self):
5: (8)                      nf = nx.filters.no_filter
6: (8)                      assert nf()
7: (8)                      assert nf(1)
8: (8)                      assert nf(2, 1)
9: (4)                  def test_hide_nodes(self):
10: (8)                     f = nx.classes.filters.hide_nodes([1, 2, 3])
11: (8)                     assert not f(1)
12: (8)                     assert not f(2)
13: (8)                     assert not f(3)
14: (8)                     assert f(4)
15: (8)                     assert f(0)
16: (8)                     assert f("a")
17: (8)                     pytest.raises(TypeError, f, 1, 2)
18: (8)                     pytest.raises(TypeError, f)
19: (4)                 def test_show_nodes(self):
20: (8)                     f = nx.classes.filters.show_nodes([1, 2, 3])
21: (8)                     assert f(1)
22: (8)                     assert f(2)
23: (8)                     assert f(3)
24: (8)                     assert not f(4)
25: (8)                     assert not f(0)
26: (8)                     assert not f("a")
27: (8)                     pytest.raises(TypeError, f, 1, 2)
28: (8)                     pytest.raises(TypeError, f)
29: (4)                 def test_hide_edges(self):
30: (8)                     factory = nx.classes.filters.hide_edges
31: (8)                     f = factory([(1, 2), (3, 4)])
32: (8)                     assert not f(1, 2)
33: (8)                     assert not f(3, 4)
34: (8)                     assert not f(4, 3)
35: (8)                     assert f(2, 3)
36: (8)                     assert f(0, -1)
37: (8)                     assert f("a", "b")
38: (8)                     pytest.raises(TypeError, f, 1, 2, 3)
39: (8)                     pytest.raises(TypeError, f, 1)
40: (8)                     pytest.raises(TypeError, f)
41: (8)                     pytest.raises(TypeError, factory, [1, 2, 3])
42: (8)                     pytest.raises(ValueError, factory, [(1, 2, 3)])
43: (4)                 def test_show_edges(self):
44: (8)                     factory = nx.classes.filters.show_edges
45: (8)                     f = factory([(1, 2), (3, 4)])
46: (8)                     assert f(1, 2)
47: (8)                     assert f(3, 4)
```

```
(8)                     assert f(4, 3)
(8)                     assert not f(2, 3)
(8)                     assert not f(0, -1)
(8)                     assert not f("a", "b")
(8)                     pytest.raises(TypeError, f, 1, 2, 3)
(8)                     pytest.raises(TypeError, f, 1)
(8)                     pytest.raises(TypeError, f)
(8)                     pytest.raises(TypeError, factory, [1, 2, 3])
(8)                     pytest.raises(ValueError, factory, [(1, 2, 3)])
(4)                 def test_hide_diedges(self):
(8)                     factory = nx.classes.filters.hide_diedges
(8)                     f = factory([(1, 2), (3, 4)])
(8)                     assert not f(1, 2)
(8)                     assert not f(3, 4)
(8)                     assert f(4, 3)
(8)                     assert f(2, 3)
(8)                     assert f(0, -1)
(8)                     assert f("a", "b")
(8)                     pytest.raises(TypeError, f, 1, 2, 3)
(8)                     pytest.raises(TypeError, f, 1)
(8)                     pytest.raises(TypeError, f)
(8)                     pytest.raises(TypeError, factory, [1, 2, 3])
(8)                     pytest.raises(ValueError, factory, [(1, 2, 3)])
(4)                 def test_show_diedges(self):
(8)                     factory = nx.classes.filters.show_diedges
(8)                     f = factory([(1, 2), (3, 4)])
(8)                     assert f(1, 2)
(8)                     assert f(3, 4)
(8)                     assert not f(4, 3)
(8)                     assert not f(2, 3)
(8)                     assert not f(0, -1)
(8)                     assert not f("a", "b")
(8)                     pytest.raises(TypeError, f, 1, 2, 3)
(8)                     pytest.raises(TypeError, f, 1)
(8)                     pytest.raises(TypeError, f)
(8)                     pytest.raises(TypeError, factory, [1, 2, 3])
(8)                     pytest.raises(ValueError, factory, [(1, 2, 3)])
(4)                 def test_hide_multiedges(self):
(8)                     factory = nx.classes.filters.hide_multiedges
(8)                     f = factory([(1, 2, 0), (3, 4, 1), (1, 2, 1)])
(8)                     assert not f(1, 2, 0)
(8)                     assert not f(1, 2, 1)
(8)                     assert f(1, 2, 2)
(8)                     assert f(3, 4, 0)
(8)                     assert not f(3, 4, 1)
(8)                     assert not f(4, 3, 1)
(8)                     assert f(4, 3, 0)
(8)                     assert f(2, 3, 0)
(8)                     assert f(0, -1, 0)
(8)                     assert f("a", "b", 0)
(8)                     pytest.raises(TypeError, f, 1, 2, 3, 4)
(8)                     pytest.raises(TypeError, f, 1, 2)
(8)                     pytest.raises(TypeError, f, 1)
(8)                     pytest.raises(TypeError, f)
(8)                     pytest.raises(TypeError, factory, [1, 2, 3])
(8)                     pytest.raises(ValueError, factory, [(1, 2)])
(8)                     pytest.raises(ValueError, factory, [(1, 2, 3, 4)])
(4)                 def test_show_multiedges(self):
(8)                     factory = nx.classes.filters.show_multiedges
(8)                     f = factory([(1, 2, 0), (3, 4, 1), (1, 2, 1)])
(8)                     assert f(1, 2, 0)
(8)                     assert f(1, 2, 1)
(8)                     assert not f(1, 2, 2)
(8)                     assert not f(3, 4, 0)
(8)                     assert f(3, 4, 1)
(8)                     assert f(4, 3, 1)
(8)                     assert not f(4, 3, 0)
(8)                     assert not f(2, 3, 0)
(8)                     assert not f(0, -1, 0)
```

```
117: (8)                    assert not f("a", "b", 0)
118: (8)                    pytest.raises(TypeError, f, 1, 2, 3, 4)
119: (8)                    pytest.raises(TypeError, f, 1, 2)
120: (8)                    pytest.raises(TypeError, f, 1)
121: (8)                    pytest.raises(TypeError, f)
122: (8)                    pytest.raises(TypeError, factory, [1, 2, 3])
123: (8)                    pytest.raises(ValueError, factory, [(1, 2)])
124: (8)                    pytest.raises(ValueError, factory, [(1, 2, 3, 4)])
125: (4)                def test_hide_multidiedges(self):
126: (8)                    factory = nx.classes.filters.hide_multidiedges
127: (8)                    f = factory([(1, 2, 0), (3, 4, 1), (1, 2, 1)])
128: (8)                    assert not f(1, 2, 0)
129: (8)                    assert not f(1, 2, 1)
130: (8)                    assert f(1, 2, 2)
131: (8)                    assert f(3, 4, 0)
132: (8)                    assert not f(3, 4, 1)
133: (8)                    assert f(4, 3, 1)
134: (8)                    assert f(4, 3, 0)
135: (8)                    assert f(2, 3, 0)
136: (8)                    assert f(0, -1, 0)
137: (8)                    assert f("a", "b", 0)
138: (8)                    pytest.raises(TypeError, f, 1, 2, 3, 4)
139: (8)                    pytest.raises(TypeError, f, 1, 2)
140: (8)                    pytest.raises(TypeError, f, 1)
141: (8)                    pytest.raises(TypeError, f)
142: (8)                    pytest.raises(TypeError, factory, [1, 2, 3])
143: (8)                    pytest.raises(ValueError, factory, [(1, 2)])
144: (8)                    pytest.raises(ValueError, factory, [(1, 2, 3, 4)])
145: (4)                def test_show_multidiedges(self):
146: (8)                    factory = nx.classes.filters.show_multidiedges
147: (8)                    f = factory([(1, 2, 0), (3, 4, 1), (1, 2, 1)])
148: (8)                    assert f(1, 2, 0)
149: (8)                    assert f(1, 2, 1)
150: (8)                    assert not f(1, 2, 2)
151: (8)                    assert not f(3, 4, 0)
152: (8)                    assert f(3, 4, 1)
153: (8)                    assert not f(4, 3, 1)
154: (8)                    assert not f(4, 3, 0)
155: (8)                    assert not f(2, 3, 0)
156: (8)                    assert not f(0, -1, 0)
157: (8)                    assert not f("a", "b", 0)
158: (8)                    pytest.raises(TypeError, f, 1, 2, 3, 4)
159: (8)                    pytest.raises(TypeError, f, 1, 2)
160: (8)                    pytest.raises(TypeError, f, 1)
161: (8)                    pytest.raises(TypeError, f)
162: (8)                    pytest.raises(TypeError, factory, [1, 2, 3])
163: (8)                    pytest.raises(ValueError, factory, [(1, 2)])
164: (8)                    pytest.raises(ValueError, factory, [(1, 2, 3, 4)])

----------------------------------------

File 404 - test_digraph_historical.py:

1: (0)              """Original NetworkX graph tests"""
2: (0)              import pytest
3: (0)              import networkx
4: (0)              import networkx as nx
5: (0)              from .historical_tests import HistoricalTests
6: (0)              class TestDiGraphHistorical(HistoricalTests):
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      HistoricalTests.setup_class()
10: (8)                     cls.G = nx.DiGraph
11: (4)                 def test_in_degree(self):
12: (8)                     G = self.G()
13: (8)                     G.add_nodes_from("GJK")
14: (8)                     G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("B", "C"),
("C", "D")])
15: (8)                     assert sorted(d for n, d in G.in_degree()) == [0, 0, 0, 0, 1, 2, 2]
```

```
        assert dict(G.in_degree()) == {
            "A": 0,
            "C": 2,
            "B": 1,
            "D": 2,
            "G": 0,
            "K": 0,
            "J": 0,
        }
    def test_out_degree(self):
        G = self.G()
        G.add_nodes_from("GJK")
        G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("B", "C"), ("C", "D")])
        assert sorted(v for k, v in G.in_degree()) == [0, 0, 0, 0, 1, 2, 2]
        assert dict(G.out_degree()) == {
            "A": 2,
            "C": 1,
            "B": 2,
            "D": 0,
            "G": 0,
            "K": 0,
            "J": 0,
        }
    def test_degree_digraph(self):
        H = nx.DiGraph()
        H.add_edges_from([(1, 24), (1, 2)])
        assert sorted(d for n, d in H.in_degree([1, 24])) == [0, 1]
        assert sorted(d for n, d in H.out_degree([1, 24])) == [0, 2]
        assert sorted(d for n, d in H.degree([1, 24])) == [1, 2]
    def test_neighbors(self):
        G = self.G()
        G.add_nodes_from("GJK")
        G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("B", "C"), ("C", "D")])
        assert sorted(G.neighbors("C")) == ["D"]
        assert sorted(G["C"]) == ["D"]
        assert sorted(G.neighbors("A")) == ["B", "C"]
        pytest.raises(nx.NetworkXError, G.neighbors, "j")
        pytest.raises(nx.NetworkXError, G.neighbors, "j")
    def test_successors(self):
        G = self.G()
        G.add_nodes_from("GJK")
        G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("B", "C"), ("C", "D")])
        assert sorted(G.successors("A")) == ["B", "C"]
        assert sorted(G.successors("A")) == ["B", "C"]
        assert sorted(G.successors("G")) == []
        assert sorted(G.successors("D")) == []
        assert sorted(G.successors("G")) == []
        pytest.raises(nx.NetworkXError, G.successors, "j")
        pytest.raises(nx.NetworkXError, G.successors, "j")
    def test_predecessors(self):
        G = self.G()
        G.add_nodes_from("GJK")
        G.add_edges_from([("A", "B"), ("A", "C"), ("B", "D"), ("B", "C"), ("C", "D")])
        assert sorted(G.predecessors("C")) == ["A", "B"]
        assert sorted(G.predecessors("C")) == ["A", "B"]
        assert sorted(G.predecessors("G")) == []
        assert sorted(G.predecessors("A")) == []
        assert sorted(G.predecessors("G")) == []
        assert sorted(G.predecessors("A")) == []
        assert sorted(G.successors("D")) == []
        pytest.raises(nx.NetworkXError, G.predecessors, "j")
        pytest.raises(nx.NetworkXError, G.predecessors, "j")
    def test_reverse(self):
        G = nx.complete_graph(10)
        H = G.to_directed()
```

```
81: (8)                     HR = H.reverse()
82: (8)                     assert nx.is_isomorphic(H, HR)
83: (8)                     assert sorted(H.edges()) == sorted(HR.edges())
84: (4)                 def test_reverse2(self):
85: (8)                     H = nx.DiGraph()
86: (8)                     foo = [H.add_edge(u, u + 1) for u in range(5)]
87: (8)                     HR = H.reverse()
88: (8)                     for u in range(5):
89: (12)                        assert HR.has_edge(u + 1, u)
90: (4)                 def test_reverse3(self):
91: (8)                     H = nx.DiGraph()
92: (8)                     H.add_nodes_from([1, 2, 3, 4])
93: (8)                     HR = H.reverse()
94: (8)                     assert sorted(HR.nodes()) == [1, 2, 3, 4]
```

File 405 - test_graph.py:

```
1: (0)              import gc
2: (0)              import pickle
3: (0)              import platform
4: (0)              import weakref
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              from networkx.utils import edges_equal, graphs_equal, nodes_equal
8: (0)              class BaseGraphTester:
9: (4)                  """Tests for data-structure independent graph class features."""
10: (4)                 def test_contains(self):
11: (8)                     G = self.K3
12: (8)                     assert 1 in G
13: (8)                     assert 4 not in G
14: (8)                     assert "b" not in G
15: (8)                     assert [] not in G  # no exception for nonhashable
16: (8)                     assert {1: 1} not in G  # no exception for nonhashable
17: (4)                 def test_order(self):
18: (8)                     G = self.K3
19: (8)                     assert len(G) == 3
20: (8)                     assert G.order() == 3
21: (8)                     assert G.number_of_nodes() == 3
22: (4)                 def test_nodes(self):
23: (8)                     G = self.K3
24: (8)                     assert isinstance(G._node, G.node_dict_factory)
25: (8)                     assert isinstance(G._adj, G.adjlist_outer_dict_factory)
26: (8)                     assert all(
27: (12)                        isinstance(adj, G.adjlist_inner_dict_factory) for adj in
G._adj.values()
28: (8)                     )
29: (8)                     assert sorted(G.nodes()) == self.k3nodes
30: (8)                     assert sorted(G.nodes(data=True)) == [(0, {}), (1, {}), (2, {})]
31: (4)                 def test_none_node(self):
32: (8)                     G = self.Graph()
33: (8)                     with pytest.raises(ValueError):
34: (12)                        G.add_node(None)
35: (8)                     with pytest.raises(ValueError):
36: (12)                        G.add_nodes_from([None])
37: (8)                     with pytest.raises(ValueError):
38: (12)                        G.add_edge(0, None)
39: (8)                     with pytest.raises(ValueError):
40: (12)                        G.add_edges_from([(0, None)])
41: (4)                 def test_has_node(self):
42: (8)                     G = self.K3
43: (8)                     assert G.has_node(1)
44: (8)                     assert not G.has_node(4)
45: (8)                     assert not G.has_node([])  # no exception for nonhashable
46: (8)                     assert not G.has_node({1: 1})  # no exception for nonhashable
47: (4)                 def test_has_edge(self):
48: (8)                     G = self.K3
49: (8)                     assert G.has_edge(0, 1)
```

```
50: (8)                     assert not G.has_edge(0, -1)
51: (4)                 def test_neighbors(self):
52: (8)                     G = self.K3
53: (8)                     assert sorted(G.neighbors(0)) == [1, 2]
54: (8)                     with pytest.raises(nx.NetworkXError):
55: (12)                        G.neighbors(-1)
56: (4)                 @pytest.mark.skipif(
57: (8)                     platform.python_implementation() == "PyPy", reason="PyPy gc is
different"
58: (4)                 )
59: (4)                 def test_memory_leak(self):
60: (8)                     G = self.Graph()
61: (8)                     def count_objects_of_type(_type):
62: (12)                        return sum(
63: (16)                            1
64: (16)                            for obj in gc.get_objects()
65: (16)                            if not isinstance(obj, weakref.ProxyTypes) and isinstance(obj,
_type)
66: (12)                        )
67: (8)                     gc.collect()
68: (8)                     before = count_objects_of_type(self.Graph)
69: (8)                     G.copy()
70: (8)                     gc.collect()
71: (8)                     after = count_objects_of_type(self.Graph)
72: (8)                     assert before == after
73: (8)                     class MyGraph(self.Graph):
74: (12)                        pass
75: (8)                     gc.collect()
76: (8)                     G = MyGraph()
77: (8)                     before = count_objects_of_type(MyGraph)
78: (8)                     G.copy()
79: (8)                     gc.collect()
80: (8)                     after = count_objects_of_type(MyGraph)
81: (8)                     assert before == after
82: (4)                 def test_edges(self):
83: (8)                     G = self.K3
84: (8)                     assert isinstance(G._adj, G.adjlist_outer_dict_factory)
85: (8)                     assert edges_equal(G.edges(), [(0, 1), (0, 2), (1, 2)])
86: (8)                     assert edges_equal(G.edges(0), [(0, 1), (0, 2)])
87: (8)                     assert edges_equal(G.edges([0, 1]), [(0, 1), (0, 2), (1, 2)])
88: (8)                     with pytest.raises(nx.NetworkXError):
89: (12)                        G.edges(-1)
90: (4)                 def test_degree(self):
91: (8)                     G = self.K3
92: (8)                     assert sorted(G.degree()) == [(0, 2), (1, 2), (2, 2)]
93: (8)                     assert dict(G.degree()) == {0: 2, 1: 2, 2: 2}
94: (8)                     assert G.degree(0) == 2
95: (8)                     with pytest.raises(nx.NetworkXError):
96: (12)                        G.degree(-1)  # node not in graph
97: (4)                 def test_size(self):
98: (8)                     G = self.K3
99: (8)                     assert G.size() == 3
100: (8)                    assert G.number_of_edges() == 3
101: (4)                def test_nbunch_iter(self):
102: (8)                    G = self.K3
103: (8)                    assert nodes_equal(G.nbunch_iter(), self.k3nodes)  # all nodes
104: (8)                    assert nodes_equal(G.nbunch_iter(0), [0])  # single node
105: (8)                    assert nodes_equal(G.nbunch_iter([0, 1]), [0, 1])  # sequence
106: (8)                    assert nodes_equal(G.nbunch_iter([-1]), [])
107: (8)                    assert nodes_equal(G.nbunch_iter("foo"), [])
108: (8)                    bunch = G.nbunch_iter(-1)
109: (8)                    with pytest.raises(nx.NetworkXError, match="is not a node or a
sequence"):
110: (12)                       list(bunch)
111: (8)                    bunch = G.nbunch_iter([0, 1, 2, {}])
112: (8)                    with pytest.raises(
113: (12)                       nx.NetworkXError, match="in sequence nbunch is not a valid node"
114: (8)                    ):
115: (12)                       list(bunch)
```

```
(4)     def test_nbunch_iter_node_format_raise(self):
(8)         G = self.Graph()
(8)         nbunch = [("x", set())]
(8)         with pytest.raises(nx.NetworkXError):
(12)            list(G.nbunch_iter(nbunch))
(4)     def test_selfloop_degree(self):
(8)         G = self.Graph()
(8)         G.add_edge(1, 1)
(8)         assert sorted(G.degree()) == [(1, 2)]
(8)         assert dict(G.degree()) == {1: 2}
(8)         assert G.degree(1) == 2
(8)         assert sorted(G.degree([1])) == [(1, 2)]
(8)         assert G.degree(1, weight="weight") == 2
(4)     def test_selfloops(self):
(8)         G = self.K3.copy()
(8)         G.add_edge(0, 0)
(8)         assert nodes_equal(nx.nodes_with_selfloops(G), [0])
(8)         assert edges_equal(nx.selfloop_edges(G), [(0, 0)])
(8)         assert nx.number_of_selfloops(G) == 1
(8)         G.remove_edge(0, 0)
(8)         G.add_edge(0, 0)
(8)         G.remove_edges_from([(0, 0)])
(8)         G.add_edge(1, 1)
(8)         G.remove_node(1)
(8)         G.add_edge(0, 0)
(8)         G.add_edge(1, 1)
(8)         G.remove_nodes_from([0, 1])
(4)     def test_cache_reset(self):
(8)         G = self.K3.copy()
(8)         old_adj = G.adj
(8)         assert id(G.adj) == id(old_adj)
(8)         G._adj = {}
(8)         assert id(G.adj) != id(old_adj)
(8)         old_nodes = G.nodes
(8)         assert id(G.nodes) == id(old_nodes)
(8)         G._node = {}
(8)         assert id(G.nodes) != id(old_nodes)
(4)     def test_attributes_cached(self):
(8)         G = self.K3.copy()
(8)         assert id(G.nodes) == id(G.nodes)
(8)         assert id(G.edges) == id(G.edges)
(8)         assert id(G.degree) == id(G.degree)
(8)         assert id(G.adj) == id(G.adj)
(0) class BaseAttrGraphTester(BaseGraphTester):
(4)     """Tests of graph class attribute features."""
(4)     def test_weighted_degree(self):
(8)         G = self.Graph()
(8)         G.add_edge(1, 2, weight=2, other=3)
(8)         G.add_edge(2, 3, weight=3, other=4)
(8)         assert sorted(d for n, d in G.degree(weight="weight")) == [2, 3, 5]
(8)         assert dict(G.degree(weight="weight")) == {1: 2, 2: 5, 3: 3}
(8)         assert G.degree(1, weight="weight") == 2
(8)         assert nodes_equal((G.degree([1], weight="weight")), [(1, 2)])
(8)         assert nodes_equal((d for n, d in G.degree(weight="other")), [3, 7,
4])
(8)         assert dict(G.degree(weight="other")) == {1: 3, 2: 7, 3: 4}
(8)         assert G.degree(1, weight="other") == 3
(8)         assert edges_equal((G.degree([1], weight="other")), [(1, 3)])
(4)     def add_attributes(self, G):
(8)         G.graph["foo"] = []
(8)         G.nodes[0]["foo"] = []
(8)         G.remove_edge(1, 2)
(8)         ll = []
(8)         G.add_edge(1, 2, foo=ll)
(8)         G.add_edge(2, 1, foo=ll)
(4)     def test_name(self):
(8)         G = self.Graph(name="")
(8)         assert G.name == ""
(8)         G = self.Graph(name="test")
```

```
184: (8)                    assert G.name == "test"
185: (4)                def test_str_unnamed(self):
186: (8)                    G = self.Graph()
187: (8)                    G.add_edges_from([(1, 2), (2, 3)])
188: (8)                    assert str(G) == f"{type(G).__name__} with 3 nodes and 2 edges"
189: (4)                def test_str_named(self):
190: (8)                    G = self.Graph(name="foo")
191: (8)                    G.add_edges_from([(1, 2), (2, 3)])
192: (8)                    assert str(G) == f"{type(G).__name__} named 'foo' with 3 nodes and 2
edges"
193: (4)                def test_graph_chain(self):
194: (8)                    G = self.Graph([(0, 1), (1, 2)])
195: (8)                    DG = G.to_directed(as_view=True)
196: (8)                    SDG = DG.subgraph([0, 1])
197: (8)                    RSDG = SDG.reverse(copy=False)
198: (8)                    assert G is DG._graph
199: (8)                    assert DG is SDG._graph
200: (8)                    assert SDG is RSDG._graph
201: (4)                def test_copy(self):
202: (8)                    G = self.Graph()
203: (8)                    G.add_node(0)
204: (8)                    G.add_edge(1, 2)
205: (8)                    self.add_attributes(G)
206: (8)                    H = G.copy()
207: (8)                    self.graphs_equal(H, G)
208: (8)                    self.different_attrdict(H, G)
209: (8)                    self.shallow_copy_attrdict(H, G)
210: (4)                def test_class_copy(self):
211: (8)                    G = self.Graph()
212: (8)                    G.add_node(0)
213: (8)                    G.add_edge(1, 2)
214: (8)                    self.add_attributes(G)
215: (8)                    H = G.__class__(G)
216: (8)                    self.graphs_equal(H, G)
217: (8)                    self.different_attrdict(H, G)
218: (8)                    self.shallow_copy_attrdict(H, G)
219: (4)                def test_fresh_copy(self):
220: (8)                    G = self.Graph()
221: (8)                    G.add_node(0)
222: (8)                    G.add_edge(1, 2)
223: (8)                    self.add_attributes(G)
224: (8)                    H = G.__class__()
225: (8)                    H.add_nodes_from(G)
226: (8)                    H.add_edges_from(G.edges())
227: (8)                    assert len(G.nodes[0]) == 1
228: (8)                    ddict = G.adj[1][2][0] if G.is_multigraph() else G.adj[1][2]
229: (8)                    assert len(ddict) == 1
230: (8)                    assert len(H.nodes[0]) == 0
231: (8)                    ddict = H.adj[1][2][0] if H.is_multigraph() else H.adj[1][2]
232: (8)                    assert len(ddict) == 0
233: (4)                def is_deepcopy(self, H, G):
234: (8)                    self.graphs_equal(H, G)
235: (8)                    self.different_attrdict(H, G)
236: (8)                    self.deep_copy_attrdict(H, G)
237: (4)                def deep_copy_attrdict(self, H, G):
238: (8)                    self.deepcopy_graph_attr(H, G)
239: (8)                    self.deepcopy_node_attr(H, G)
240: (8)                    self.deepcopy_edge_attr(H, G)
241: (4)                def deepcopy_graph_attr(self, H, G):
242: (8)                    assert G.graph["foo"] == H.graph["foo"]
243: (8)                    G.graph["foo"].append(1)
244: (8)                    assert G.graph["foo"] != H.graph["foo"]
245: (4)                def deepcopy_node_attr(self, H, G):
246: (8)                    assert G.nodes[0]["foo"] == H.nodes[0]["foo"]
247: (8)                    G.nodes[0]["foo"].append(1)
248: (8)                    assert G.nodes[0]["foo"] != H.nodes[0]["foo"]
249: (4)                def deepcopy_edge_attr(self, H, G):
250: (8)                    assert G[1][2]["foo"] == H[1][2]["foo"]
251: (8)                    G[1][2]["foo"].append(1)
```

```
(8)                 assert G[1][2]["foo"] != H[1][2]["foo"]
(4)             def is_shallow_copy(self, H, G):
(8)                 self.graphs_equal(H, G)
(8)                 self.shallow_copy_attrdict(H, G)
(4)             def shallow_copy_attrdict(self, H, G):
(8)                 self.shallow_copy_graph_attr(H, G)
(8)                 self.shallow_copy_node_attr(H, G)
(8)                 self.shallow_copy_edge_attr(H, G)
(4)             def shallow_copy_graph_attr(self, H, G):
(8)                 assert G.graph["foo"] == H.graph["foo"]
(8)                 G.graph["foo"].append(1)
(8)                 assert G.graph["foo"] == H.graph["foo"]
(4)             def shallow_copy_node_attr(self, H, G):
(8)                 assert G.nodes[0]["foo"] == H.nodes[0]["foo"]
(8)                 G.nodes[0]["foo"].append(1)
(8)                 assert G.nodes[0]["foo"] == H.nodes[0]["foo"]
(4)             def shallow_copy_edge_attr(self, H, G):
(8)                 assert G[1][2]["foo"] == H[1][2]["foo"]
(8)                 G[1][2]["foo"].append(1)
(8)                 assert G[1][2]["foo"] == H[1][2]["foo"]
(4)             def same_attrdict(self, H, G):
(8)                 old_foo = H[1][2]["foo"]
(8)                 H.adj[1][2]["foo"] = "baz"
(8)                 assert G.edges == H.edges
(8)                 H.adj[1][2]["foo"] = old_foo
(8)                 assert G.edges == H.edges
(8)                 old_foo = H.nodes[0]["foo"]
(8)                 H.nodes[0]["foo"] = "baz"
(8)                 assert G.nodes == H.nodes
(8)                 H.nodes[0]["foo"] = old_foo
(8)                 assert G.nodes == H.nodes
(4)             def different_attrdict(self, H, G):
(8)                 old_foo = H[1][2]["foo"]
(8)                 H.adj[1][2]["foo"] = "baz"
(8)                 assert G._adj != H._adj
(8)                 H.adj[1][2]["foo"] = old_foo
(8)                 assert G._adj == H._adj
(8)                 old_foo = H.nodes[0]["foo"]
(8)                 H.nodes[0]["foo"] = "baz"
(8)                 assert G._node != H._node
(8)                 H.nodes[0]["foo"] = old_foo
(8)                 assert G._node == H._node
(4)             def graphs_equal(self, H, G):
(8)                 assert G._adj == H._adj
(8)                 assert G._node == H._node
(8)                 assert G.graph == H.graph
(8)                 assert G.name == H.name
(8)                 if not G.is_directed() and not H.is_directed():
(12)                    assert H._adj[1][2] is H._adj[2][1]
(12)                    assert G._adj[1][2] is G._adj[2][1]
(8)                 else:  # at least one is directed
(12)                    if not G.is_directed():
(16)                        G._pred = G._adj
(16)                        G._succ = G._adj
(12)                    if not H.is_directed():
(16)                        H._pred = H._adj
(16)                        H._succ = H._adj
(12)                    assert G._pred == H._pred
(12)                    assert G._succ == H._succ
(12)                    assert H._succ[1][2] is H._pred[2][1]
(12)                    assert G._succ[1][2] is G._pred[2][1]
(4)             def test_graph_attr(self):
(8)                 G = self.K3.copy()
(8)                 G.graph["foo"] = "bar"
(8)                 assert isinstance(G.graph, G.graph_attr_dict_factory)
(8)                 assert G.graph["foo"] == "bar"
(8)                 del G.graph["foo"]
(8)                 assert G.graph == {}
(8)                 H = self.Graph(foo="bar")
```

```
        assert H.graph["foo"] == "bar"
    def test_node_attr(self):
        G = self.K3.copy()
        G.add_node(1, foo="bar")
        assert all(
            isinstance(d, G.node_attr_dict_factory) for u, d in G.nodes(data=True)
        )
        assert nodes_equal(G.nodes(), [0, 1, 2])
        assert nodes_equal(G.nodes(data=True), [(0, {}), (1, {"foo": "bar"}), (2, {})])
        G.nodes[1]["foo"] = "baz"
        assert nodes_equal(G.nodes(data=True), [(0, {}), (1, {"foo": "baz"}), (2, {})])
        assert nodes_equal(G.nodes(data="foo"), [(0, None), (1, "baz"), (2, None)])
        assert nodes_equal(
            G.nodes(data="foo", default="bar"), [(0, "bar"), (1, "baz"), (2, "bar")]
        )
    def test_node_attr2(self):
        G = self.K3.copy()
        a = {"foo": "bar"}
        G.add_node(3, **a)
        assert nodes_equal(G.nodes(), [0, 1, 2, 3])
        assert nodes_equal(
            G.nodes(data=True), [(0, {}), (1, {}), (2, {}), (3, {"foo": "bar"})]
        )
    def test_edge_lookup(self):
        G = self.Graph()
        G.add_edge(1, 2, foo="bar")
        assert edges_equal(G.edges[1, 2], {"foo": "bar"})
    def test_edge_attr(self):
        G = self.Graph()
        G.add_edge(1, 2, foo="bar")
        assert all(
            isinstance(d, G.edge_attr_dict_factory) for u, v, d in G.edges(data=True)
        )
        assert edges_equal(G.edges(data=True), [(1, 2, {"foo": "bar"})])
        assert edges_equal(G.edges(data="foo"), [(1, 2, "bar")])
    def test_edge_attr2(self):
        G = self.Graph()
        G.add_edges_from([(1, 2), (3, 4)], foo="foo")
        assert edges_equal(
            G.edges(data=True), [(1, 2, {"foo": "foo"}), (3, 4, {"foo": "foo"})]
        )
        assert edges_equal(G.edges(data="foo"), [(1, 2, "foo"), (3, 4, "foo")])
    def test_edge_attr3(self):
        G = self.Graph()
        G.add_edges_from([(1, 2, {"weight": 32}), (3, 4, {"weight": 64})], foo="foo")
        assert edges_equal(
            G.edges(data=True),
            [
                (1, 2, {"foo": "foo", "weight": 32}),
                (3, 4, {"foo": "foo", "weight": 64}),
            ],
        )
        G.remove_edges_from([(1, 2), (3, 4)])
        G.add_edge(1, 2, data=7, spam="bar", bar="foo")
        assert edges_equal(
            G.edges(data=True), [(1, 2, {"data": 7, "spam": "bar", "bar": "foo"})]
        )
    def test_edge_attr4(self):
```

```
379: (8)                    G = self.Graph()
380: (8)                    G.add_edge(1, 2, data=7, spam="bar", bar="foo")
381: (8)                    assert edges_equal(
382: (12)                       G.edges(data=True), [(1, 2, {"data": 7, "spam": "bar", "bar":
"foo"})]
383: (8)                    )
384: (8)                    G[1][2]["data"] = 10  # OK to set data like this
385: (8)                    assert edges_equal(
386: (12)                       G.edges(data=True), [(1, 2, {"data": 10, "spam": "bar", "bar":
"foo"})]
387: (8)                    )
388: (8)                    G.adj[1][2]["data"] = 20
389: (8)                    assert edges_equal(
390: (12)                       G.edges(data=True), [(1, 2, {"data": 20, "spam": "bar", "bar":
"foo"})]
391: (8)                    )
392: (8)                    G.edges[1, 2]["data"] = 21  # another spelling, "edge"
393: (8)                    assert edges_equal(
394: (12)                       G.edges(data=True), [(1, 2, {"data": 21, "spam": "bar", "bar":
"foo"})]
395: (8)                    )
396: (8)                    G.adj[1][2]["listdata"] = [20, 200]
397: (8)                    G.adj[1][2]["weight"] = 20
398: (8)                    dd = {
399: (12)                       "data": 21,
400: (12)                       "spam": "bar",
401: (12)                       "bar": "foo",
402: (12)                       "listdata": [20, 200],
403: (12)                       "weight": 20,
404: (8)                    }
405: (8)                    assert edges_equal(G.edges(data=True), [(1, 2, dd)])
406: (4)                def test_to_undirected(self):
407: (8)                    G = self.K3
408: (8)                    self.add_attributes(G)
409: (8)                    H = nx.Graph(G)
410: (8)                    self.is_shallow_copy(H, G)
411: (8)                    self.different_attrdict(H, G)
412: (8)                    H = G.to_undirected()
413: (8)                    self.is_deepcopy(H, G)
414: (4)                def test_to_directed_as_view(self):
415: (8)                    H = nx.path_graph(2, create_using=self.Graph)
416: (8)                    H2 = H.to_directed(as_view=True)
417: (8)                    assert H is H2._graph
418: (8)                    assert H2.has_edge(0, 1)
419: (8)                    assert H2.has_edge(1, 0) or H.is_directed()
420: (8)                    pytest.raises(nx.NetworkXError, H2.add_node, -1)
421: (8)                    pytest.raises(nx.NetworkXError, H2.add_edge, 1, 2)
422: (8)                    H.add_edge(1, 2)
423: (8)                    assert H2.has_edge(1, 2)
424: (8)                    assert H2.has_edge(2, 1) or H.is_directed()
425: (4)                def test_to_undirected_as_view(self):
426: (8)                    H = nx.path_graph(2, create_using=self.Graph)
427: (8)                    H2 = H.to_undirected(as_view=True)
428: (8)                    assert H is H2._graph
429: (8)                    assert H2.has_edge(0, 1)
430: (8)                    assert H2.has_edge(1, 0)
431: (8)                    pytest.raises(nx.NetworkXError, H2.add_node, -1)
432: (8)                    pytest.raises(nx.NetworkXError, H2.add_edge, 1, 2)
433: (8)                    H.add_edge(1, 2)
434: (8)                    assert H2.has_edge(1, 2)
435: (8)                    assert H2.has_edge(2, 1)
436: (4)                def test_directed_class(self):
437: (8)                    G = self.Graph()
438: (8)                    class newGraph(G.to_undirected_class()):
439: (12)                       def to_directed_class(self):
440: (16)                           return newDiGraph
441: (12)                       def to_undirected_class(self):
442: (16)                           return newGraph
443: (8)                    class newDiGraph(G.to_directed_class()):
```

```
(12)                        def to_directed_class(self):
(16)                            return newDiGraph
(12)                        def to_undirected_class(self):
(16)                            return newGraph
(8)                     G = newDiGraph() if G.is_directed() else newGraph()
(8)                     H = G.to_directed()
(8)                     assert isinstance(H, newDiGraph)
(8)                     H = G.to_undirected()
(8)                     assert isinstance(H, newGraph)
(4)                 def test_to_directed(self):
(8)                     G = self.K3
(8)                     self.add_attributes(G)
(8)                     H = nx.DiGraph(G)
(8)                     self.is_shallow_copy(H, G)
(8)                     self.different_attrdict(H, G)
(8)                     H = G.to_directed()
(8)                     self.is_deepcopy(H, G)
(4)                 def test_subgraph(self):
(8)                     G = self.K3
(8)                     self.add_attributes(G)
(8)                     H = G.subgraph([0, 1, 2, 5])
(8)                     self.graphs_equal(H, G)
(8)                     self.same_attrdict(H, G)
(8)                     self.shallow_copy_attrdict(H, G)
(8)                     H = G.subgraph(0)
(8)                     assert H.adj == {0: {}}
(8)                     H = G.subgraph([])
(8)                     assert H.adj == {}
(8)                     assert G.adj != {}
(4)                 def test_selfloops_attr(self):
(8)                     G = self.K3.copy()
(8)                     G.add_edge(0, 0)
(8)                     G.add_edge(1, 1, weight=2)
(8)                     assert edges_equal(
(12)                        nx.selfloop_edges(G, data=True), [(0, 0, {}), (1, 1, {"weight":
2})]
(8)                     )
(8)                     assert edges_equal(
(12)                        nx.selfloop_edges(G, data="weight"), [(0, 0, None), (1, 1, 2)]
(8)                     )
(0)             class TestGraph(BaseAttrGraphTester):
(4)                 """Tests specific to dict-of-dict-of-dict graph data structure"""
(4)                 def setup_method(self):
(8)                     self.Graph = nx.Graph
(8)                     ed1, ed2, ed3 = ({}, {}, {})
(8)                     self.k3adj = {0: {1: ed1, 2: ed2}, 1: {0: ed1, 2: ed3}, 2: {0: ed2, 1:
ed3}}
(8)                     self.k3edges = [(0, 1), (0, 2), (1, 2)]
(8)                     self.k3nodes = [0, 1, 2]
(8)                     self.K3 = self.Graph()
(8)                     self.K3._adj = self.k3adj
(8)                     self.K3._node = {}
(8)                     self.K3._node[0] = {}
(8)                     self.K3._node[1] = {}
(8)                     self.K3._node[2] = {}
(4)                 def test_pickle(self):
(8)                     G = self.K3
(8)                     pg = pickle.loads(pickle.dumps(G, -1))
(8)                     self.graphs_equal(pg, G)
(8)                     pg = pickle.loads(pickle.dumps(G))
(8)                     self.graphs_equal(pg, G)
(4)                 def test_data_input(self):
(8)                     G = self.Graph({1: [2], 2: [1]}, name="test")
(8)                     assert G.name == "test"
(8)                     assert sorted(G.adj.items()) == [(1, {2: {}}), (2, {1: {}})]
(4)                 def test_adjacency(self):
(8)                     G = self.K3
(8)                     assert dict(G.adjacency()) == {
(12)                        0: {1: {}, 2: {}},
```

```
(12)                        1: {0: {}, 2: {}},
(12)                        2: {0: {}, 1: {}},
(8)                     }
(4)                 def test_getitem(self):
(8)                     G = self.K3
(8)                     assert G.adj[0] == {1: {}, 2: {}}
(8)                     assert G[0] == {1: {}, 2: {}}
(8)                     with pytest.raises(KeyError):
(12)                        G.__getitem__("j")
(8)                     with pytest.raises(TypeError):
(12)                        G.__getitem__(["A"])
(4)                 def test_add_node(self):
(8)                     G = self.Graph()
(8)                     G.add_node(0)
(8)                     assert G.adj == {0: {}}
(8)                     G.add_node(1, c="red")
(8)                     G.add_node(2, c="blue")
(8)                     G.add_node(3, c="red")
(8)                     assert G.nodes[1]["c"] == "red"
(8)                     assert G.nodes[2]["c"] == "blue"
(8)                     assert G.nodes[3]["c"] == "red"
(8)                     G.add_node(1, c="blue")
(8)                     G.add_node(2, c="red")
(8)                     G.add_node(3, c="blue")
(8)                     assert G.nodes[1]["c"] == "blue"
(8)                     assert G.nodes[2]["c"] == "red"
(8)                     assert G.nodes[3]["c"] == "blue"
(4)                 def test_add_nodes_from(self):
(8)                     G = self.Graph()
(8)                     G.add_nodes_from([0, 1, 2])
(8)                     assert G.adj == {0: {}, 1: {}, 2: {}}
(8)                     G.add_nodes_from([0, 1, 2], c="red")
(8)                     assert G.nodes[0]["c"] == "red"
(8)                     assert G.nodes[2]["c"] == "red"
(8)                     assert G.nodes[0] is not G.nodes[1]
(8)                     G.add_nodes_from([0, 1, 2], c="blue")
(8)                     assert G.nodes[0]["c"] == "blue"
(8)                     assert G.nodes[2]["c"] == "blue"
(8)                     assert G.nodes[0] is not G.nodes[1]
(8)                     H = self.Graph()
(8)                     H.add_nodes_from(G.nodes(data=True))
(8)                     assert H.nodes[0]["c"] == "blue"
(8)                     assert H.nodes[2]["c"] == "blue"
(8)                     assert H.nodes[0] is not H.nodes[1]
(8)                     H.add_nodes_from([0, (1, {"c": "green"}), (3, {"c": "cyan"})],
c="red")
(8)                     assert H.nodes[0]["c"] == "red"
(8)                     assert H.nodes[1]["c"] == "green"
(8)                     assert H.nodes[2]["c"] == "blue"
(8)                     assert H.nodes[3]["c"] == "cyan"
(4)                 def test_remove_node(self):
(8)                     G = self.K3.copy()
(8)                     G.remove_node(0)
(8)                     assert G.adj == {1: {2: {}}, 2: {1: {}}}
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.remove_node(-1)
(4)                 def test_remove_nodes_from(self):
(8)                     G = self.K3.copy()
(8)                     G.remove_nodes_from([0, 1])
(8)                     assert G.adj == {2: {}}
(8)                     G.remove_nodes_from([-1])  # silent fail
(4)                 def test_add_edge(self):
(8)                     G = self.Graph()
(8)                     G.add_edge(0, 1)
(8)                     assert G.adj == {0: {1: {}}, 1: {0: {}}}
(8)                     G = self.Graph()
(8)                     G.add_edge(*(0, 1))
(8)                     assert G.adj == {0: {1: {}}, 1: {0: {}}}
(8)                     G = self.Graph()
```

```
(8)                 with pytest.raises(ValueError):
(12)                    G.add_edge(None, "anything")
(4)             def test_add_edges_from(self):
(8)                 G = self.Graph()
(8)                 G.add_edges_from([(0, 1), (0, 2, {"weight": 3})])
(8)                 assert G.adj == {
(12)                    0: {1: {}, 2: {"weight": 3}},
(12)                    1: {0: {}},
(12)                    2: {0: {"weight": 3}},
(8)                 }
(8)                 G = self.Graph()
(8)                 G.add_edges_from([(0, 1), (0, 2, {"weight": 3}), (1, 2, {"data": 4})],
data=2)
(8)                 assert G.adj == {
(12)                    0: {1: {"data": 2}, 2: {"weight": 3, "data": 2}},
(12)                    1: {0: {"data": 2}, 2: {"data": 4}},
(12)                    2: {0: {"weight": 3, "data": 2}, 1: {"data": 4}},
(8)                 }
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.add_edges_from([(0,)])  # too few in tuple
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.add_edges_from([(0, 1, 2, 3)])  # too many in tuple
(8)                 with pytest.raises(TypeError):
(12)                    G.add_edges_from([0])  # not a tuple
(8)                 with pytest.raises(ValueError):
(12)                    G.add_edges_from([(None, 3), (3, 2)])  # None cannot be a node
(4)             def test_remove_edge(self):
(8)                 G = self.K3.copy()
(8)                 G.remove_edge(0, 1)
(8)                 assert G.adj == {0: {2: {}}, 1: {2: {}}, 2: {0: {}, 1: {}}}
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.remove_edge(-1, 0)
(4)             def test_remove_edges_from(self):
(8)                 G = self.K3.copy()
(8)                 G.remove_edges_from([(0, 1)])
(8)                 assert G.adj == {0: {2: {}}, 1: {2: {}}, 2: {0: {}, 1: {}}}
(8)                 G.remove_edges_from([(0, 0)])  # silent fail
(4)             def test_clear(self):
(8)                 G = self.K3.copy()
(8)                 G.graph["name"] = "K3"
(8)                 G.clear()
(8)                 assert list(G.nodes) == []
(8)                 assert G.adj == {}
(8)                 assert G.graph == {}
(4)             def test_clear_edges(self):
(8)                 G = self.K3.copy()
(8)                 G.graph["name"] = "K3"
(8)                 nodes = list(G.nodes)
(8)                 G.clear_edges()
(8)                 assert list(G.nodes) == nodes
(8)                 assert G.adj == {0: {}, 1: {}, 2: {}}
(8)                 assert list(G.edges) == []
(8)                 assert G.graph["name"] == "K3"
(4)             def test_edges_data(self):
(8)                 G = self.K3
(8)                 all_edges = [(0, 1, {}), (0, 2, {}), (1, 2, {})]
(8)                 assert edges_equal(G.edges(data=True), all_edges)
(8)                 assert edges_equal(G.edges(0, data=True), [(0, 1, {}), (0, 2, {})])
(8)                 assert edges_equal(G.edges([0, 1], data=True), all_edges)
(8)                 with pytest.raises(nx.NetworkXError):
(12)                    G.edges(-1, True)
(4)             def test_get_edge_data(self):
(8)                 G = self.K3.copy()
(8)                 assert G.get_edge_data(0, 1) == {}
(8)                 assert G[0][1] == {}
(8)                 assert G.get_edge_data(10, 20) is None
(8)                 assert G.get_edge_data(-1, 0) is None
(8)                 assert G.get_edge_data(-1, 0, default=1) == 1
(4)             def test_update(self):
```

```
(8)                     G = self.K3.copy()
(8)                     G.update(nodes=[3, (4, {"size": 2})], edges=[(4, 5), (6, 7, {"weight":
2})])
(8)                     nlist = [
(12)                        (0, {}),
(12)                        (1, {}),
(12)                        (2, {}),
(12)                        (3, {}),
(12)                        (4, {"size": 2}),
(12)                        (5, {}),
(12)                        (6, {}),
(12)                        (7, {}),
(8)                     ]
(8)                     assert sorted(G.nodes.data()) == nlist
(8)                     if G.is_directed():
(12)                        elist = [
(16)                            (0, 1, {}),
(16)                            (0, 2, {}),
(16)                            (1, 0, {}),
(16)                            (1, 2, {}),
(16)                            (2, 0, {}),
(16)                            (2, 1, {}),
(16)                            (4, 5, {}),
(16)                            (6, 7, {"weight": 2}),
(12)                        ]
(8)                     else:
(12)                        elist = [
(16)                            (0, 1, {}),
(16)                            (0, 2, {}),
(16)                            (1, 2, {}),
(16)                            (4, 5, {}),
(16)                            (6, 7, {"weight": 2}),
(12)                        ]
(8)                     assert sorted(G.edges.data()) == elist
(8)                     assert G.graph == {}
(8)                     G = self.K3.copy()
(8)                     G.update([(4, 5), (6, 7, {"weight": 2})], [3, (4, {"size": 2})])
(8)                     assert sorted(G.nodes.data()) == nlist
(8)                     assert sorted(G.edges.data()) == elist
(8)                     assert G.graph == {}
(8)                     G = self.Graph()
(8)                     G.graph["foo"] = "bar"
(8)                     G.add_node(2, data=4)
(8)                     G.add_edge(0, 1, weight=0.5)
(8)                     GG = G.copy()
(8)                     H = self.Graph()
(8)                     GG.update(H)
(8)                     assert graphs_equal(G, GG)
(8)                     H.update(G)
(8)                     assert graphs_equal(H, G)
(8)                     H = self.Graph()
(8)                     H.update(nodes=[3, 4])
(8)                     assert H.nodes ^ {3, 4} == set()
(8)                     assert H.size() == 0
(8)                     H = self.Graph()
(8)                     H.update(edges=[(3, 4)])
(8)                     assert sorted(H.edges.data()) == [(3, 4, {})]
(8)                     assert H.size() == 1
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        nx.Graph().update()
(0)             class TestEdgeSubgraph:
(4)                 """Unit tests for the :meth:`Graph.edge_subgraph` method."""
(4)                 def setup_method(self):
(8)                     G = nx.path_graph(5)
(8)                     for i in range(5):
(12)                        G.nodes[i]["name"] = f"node{i}"
(8)                     G.edges[0, 1]["name"] = "edge01"
(8)                     G.edges[3, 4]["name"] = "edge34"
(8)                     G.graph["name"] = "graph"
```

```
715: (8)                    self.G = G
716: (8)                    self.H = G.edge_subgraph([(0, 1), (3, 4)])
717: (4)                def test_correct_nodes(self):
718: (8)                    """Tests that the subgraph has the correct nodes."""
719: (8)                    assert [0, 1, 3, 4] == sorted(self.H.nodes())
720: (4)                def test_correct_edges(self):
721: (8)                    """Tests that the subgraph has the correct edges."""
722: (8)                    assert [(0, 1, "edge01"), (3, 4, "edge34")] ==
sorted(self.H.edges(data="name"))
723: (4)                def test_add_node(self):
724: (8)                    """Tests that adding a node to the original graph does not
725: (8)                    affect the nodes of the subgraph.
726: (8)                    """
727: (8)                    self.G.add_node(5)
728: (8)                    assert [0, 1, 3, 4] == sorted(self.H.nodes())
729: (4)                def test_remove_node(self):
730: (8)                    """Tests that removing a node in the original graph does
731: (8)                    affect the nodes of the subgraph.
732: (8)                    """
733: (8)                    self.G.remove_node(0)
734: (8)                    assert [1, 3, 4] == sorted(self.H.nodes())
735: (4)                def test_node_attr_dict(self):
736: (8)                    """Tests that the node attribute dictionary of the two graphs is
737: (8)                    the same object.
738: (8)                    """
739: (8)                    for v in self.H:
740: (12)                       assert self.G.nodes[v] == self.H.nodes[v]
741: (8)                    self.G.nodes[0]["name"] = "foo"
742: (8)                    assert self.G.nodes[0] == self.H.nodes[0]
743: (8)                    self.H.nodes[1]["name"] = "bar"
744: (8)                    assert self.G.nodes[1] == self.H.nodes[1]
745: (4)                def test_edge_attr_dict(self):
746: (8)                    """Tests that the edge attribute dictionary of the two graphs is
747: (8)                    the same object.
748: (8)                    """
749: (8)                    for u, v in self.H.edges():
750: (12)                       assert self.G.edges[u, v] == self.H.edges[u, v]
751: (8)                    self.G.edges[0, 1]["name"] = "foo"
752: (8)                    assert self.G.edges[0, 1]["name"] == self.H.edges[0, 1]["name"]
753: (8)                    self.H.edges[3, 4]["name"] = "bar"
754: (8)                    assert self.G.edges[3, 4]["name"] == self.H.edges[3, 4]["name"]
755: (4)                def test_graph_attr_dict(self):
756: (8)                    """Tests that the graph attribute dictionary of the two graphs
757: (8)                    is the same object.
758: (8)                    """
759: (8)                    assert self.G.graph is self.H.graph

----------------------------------------

File 406 - test_function.py:

1: (0)              import random
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import edges_equal, nodes_equal
5: (0)              def test_degree_histogram_empty():
6: (4)                  G = nx.Graph()
7: (4)                  assert nx.degree_histogram(G) == []
8: (0)              class TestFunction:
9: (4)                  def setup_method(self):
10: (8)                     self.G = nx.Graph({0: [1, 2, 3], 1: [1, 2, 0], 4: []}, name="Test")
11: (8)                     self.Gdegree = {0: 3, 1: 2, 2: 2, 3: 1, 4: 0}
12: (8)                     self.Gnodes = list(range(5))
13: (8)                     self.Gedges = [(0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2)]
14: (8)                     self.DG = nx.DiGraph({0: [1, 2, 3], 1: [1, 2, 0], 4: []})
15: (8)                     self.DGin_degree = {0: 1, 1: 2, 2: 2, 3: 1, 4: 0}
16: (8)                     self.DGout_degree = {0: 3, 1: 3, 2: 0, 3: 0, 4: 0}
17: (8)                     self.DGnodes = list(range(5))
18: (8)                     self.DGedges = [(0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2)]
```

```
(4)     def test_nodes(self):
(8)         assert nodes_equal(self.G.nodes(), list(nx.nodes(self.G)))
(8)         assert nodes_equal(self.DG.nodes(), list(nx.nodes(self.DG)))
(4)     def test_edges(self):
(8)         assert edges_equal(self.G.edges(), list(nx.edges(self.G)))
(8)         assert sorted(self.DG.edges()) == sorted(nx.edges(self.DG))
(8)         assert edges_equal(
(12)            self.G.edges(nbunch=[0, 1, 3]), list(nx.edges(self.G, nbunch=[0,
1, 3]))
(8)         )
(8)         assert sorted(self.DG.edges(nbunch=[0, 1, 3])) == sorted(
(12)            nx.edges(self.DG, nbunch=[0, 1, 3])
(8)         )
(4)     def test_degree(self):
(8)         assert edges_equal(self.G.degree(), list(nx.degree(self.G)))
(8)         assert sorted(self.DG.degree()) == sorted(nx.degree(self.DG))
(8)         assert edges_equal(
(12)            self.G.degree(nbunch=[0, 1]), list(nx.degree(self.G, nbunch=[0,
1]))
(8)         )
(8)         assert sorted(self.DG.degree(nbunch=[0, 1])) == sorted(
(12)            nx.degree(self.DG, nbunch=[0, 1])
(8)         )
(8)         assert edges_equal(
(12)            self.G.degree(weight="weight"), list(nx.degree(self.G,
weight="weight"))
(8)         )
(8)         assert sorted(self.DG.degree(weight="weight")) == sorted(
(12)            nx.degree(self.DG, weight="weight")
(8)         )
(4)     def test_neighbors(self):
(8)         assert list(self.G.neighbors(1)) == list(nx.neighbors(self.G, 1))
(8)         assert list(self.DG.neighbors(1)) == list(nx.neighbors(self.DG, 1))
(4)     def test_number_of_nodes(self):
(8)         assert self.G.number_of_nodes() == nx.number_of_nodes(self.G)
(8)         assert self.DG.number_of_nodes() == nx.number_of_nodes(self.DG)
(4)     def test_number_of_edges(self):
(8)         assert self.G.number_of_edges() == nx.number_of_edges(self.G)
(8)         assert self.DG.number_of_edges() == nx.number_of_edges(self.DG)
(4)     def test_is_directed(self):
(8)         assert self.G.is_directed() == nx.is_directed(self.G)
(8)         assert self.DG.is_directed() == nx.is_directed(self.DG)
(4)     def test_add_star(self):
(8)         G = self.G.copy()
(8)         nlist = [12, 13, 14, 15]
(8)         nx.add_star(G, nlist)
(8)         assert edges_equal(G.edges(nlist), [(12, 13), (12, 14), (12, 15)])
(8)         G = self.G.copy()
(8)         nx.add_star(G, nlist, weight=2.0)
(8)         assert edges_equal(
(12)            G.edges(nlist, data=True),
(12)            [
(16)                (12, 13, {"weight": 2.0}),
(16)                (12, 14, {"weight": 2.0}),
(16)                (12, 15, {"weight": 2.0}),
(12)            ],
(8)         )
(8)         G = self.G.copy()
(8)         nlist = [12]
(8)         nx.add_star(G, nlist)
(8)         assert nodes_equal(G, list(self.G) + nlist)
(8)         G = self.G.copy()
(8)         nlist = []
(8)         nx.add_star(G, nlist)
(8)         assert nodes_equal(G.nodes, self.Gnodes)
(8)         assert edges_equal(G.edges, self.G.edges)
(4)     def test_add_path(self):
(8)         G = self.G.copy()
(8)         nlist = [12, 13, 14, 15]
```

```
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges(nlist), [(12, 13), (13, 14), (14, 15)])
(8)                     G = self.G.copy()
(8)                     nx.add_path(G, nlist, weight=2.0)
(8)                     assert edges_equal(
(12)                        G.edges(nlist, data=True),
(12)                        [
(16)                            (12, 13, {"weight": 2.0}),
(16)                            (13, 14, {"weight": 2.0}),
(16)                            (14, 15, {"weight": 2.0}),
(12)                        ],
(8)                     )
(8)                     G = self.G.copy()
(8)                     nlist = ["node"]
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges(nlist), [])
(8)                     assert nodes_equal(G, list(self.G) + ["node"])
(8)                     G = self.G.copy()
(8)                     nlist = iter(["node"])
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges(["node"]), [])
(8)                     assert nodes_equal(G, list(self.G) + ["node"])
(8)                     G = self.G.copy()
(8)                     nlist = [12]
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges(nlist), [])
(8)                     assert nodes_equal(G, list(self.G) + [12])
(8)                     G = self.G.copy()
(8)                     nlist = iter([12])
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges([12]), [])
(8)                     assert nodes_equal(G, list(self.G) + [12])
(8)                     G = self.G.copy()
(8)                     nlist = []
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges, self.G.edges)
(8)                     assert nodes_equal(G, list(self.G))
(8)                     G = self.G.copy()
(8)                     nlist = iter([])
(8)                     nx.add_path(G, nlist)
(8)                     assert edges_equal(G.edges, self.G.edges)
(8)                     assert nodes_equal(G, list(self.G))
(4)                 def test_add_cycle(self):
(8)                     G = self.G.copy()
(8)                     nlist = [12, 13, 14, 15]
(8)                     oklists = [
(12)                        [(12, 13), (12, 15), (13, 14), (14, 15)],
(12)                        [(12, 13), (13, 14), (14, 15), (15, 12)],
(8)                     ]
(8)                     nx.add_cycle(G, nlist)
(8)                     assert sorted(G.edges(nlist)) in oklists
(8)                     G = self.G.copy()
(8)                     oklists = [
(12)                        [
(16)                            (12, 13, {"weight": 1.0}),
(16)                            (12, 15, {"weight": 1.0}),
(16)                            (13, 14, {"weight": 1.0}),
(16)                            (14, 15, {"weight": 1.0}),
(12)                        ],
(12)                        [
(16)                            (12, 13, {"weight": 1.0}),
(16)                            (13, 14, {"weight": 1.0}),
(16)                            (14, 15, {"weight": 1.0}),
(16)                            (15, 12, {"weight": 1.0}),
(12)                        ],
(8)                     ]
(8)                     nx.add_cycle(G, nlist, weight=1.0)
(8)                     assert sorted(G.edges(nlist, data=True)) in oklists
(8)                     G = self.G.copy()
```

```
154: (8)                    nlist = [12]
155: (8)                    nx.add_cycle(G, nlist)
156: (8)                    assert nodes_equal(G, list(self.G) + nlist)
157: (8)                    G = self.G.copy()
158: (8)                    nlist = []
159: (8)                    nx.add_cycle(G, nlist)
160: (8)                    assert nodes_equal(G.nodes, self.Gnodes)
161: (8)                    assert edges_equal(G.edges, self.G.edges)
162: (4)                def test_subgraph(self):
163: (8)                    assert (
164: (12)                       self.G.subgraph([0, 1, 2, 4]).adj == nx.subgraph(self.G, [0, 1, 2,
4]).adj
165: (8)                    )
166: (8)                    assert (
167: (12)                       self.DG.subgraph([0, 1, 2, 4]).adj == nx.subgraph(self.DG, [0, 1,
2, 4]).adj
168: (8)                    )
169: (8)                    assert (
170: (12)                       self.G.subgraph([0, 1, 2, 4]).adj
171: (12)                       == nx.induced_subgraph(self.G, [0, 1, 2, 4]).adj
172: (8)                    )
173: (8)                    assert (
174: (12)                       self.DG.subgraph([0, 1, 2, 4]).adj
175: (12)                       == nx.induced_subgraph(self.DG, [0, 1, 2, 4]).adj
176: (8)                    )
177: (8)                    H = nx.induced_subgraph(self.G.subgraph([0, 1, 2, 4]), [0, 1, 4])
178: (8)                    assert H._graph is not self.G
179: (8)                    assert H.adj == self.G.subgraph([0, 1, 4]).adj
180: (4)                def test_edge_subgraph(self):
181: (8)                    assert (
182: (12)                       self.G.edge_subgraph([(1, 2), (0, 3)]).adj
183: (12)                       == nx.edge_subgraph(self.G, [(1, 2), (0, 3)]).adj
184: (8)                    )
185: (8)                    assert (
186: (12)                       self.DG.edge_subgraph([(1, 2), (0, 3)]).adj
187: (12)                       == nx.edge_subgraph(self.DG, [(1, 2), (0, 3)]).adj
188: (8)                    )
189: (4)                def test_create_empty_copy(self):
190: (8)                    G = nx.create_empty_copy(self.G, with_data=False)
191: (8)                    assert nodes_equal(G, list(self.G))
192: (8)                    assert G.graph == {}
193: (8)                    assert G._node == {}.fromkeys(self.G.nodes(), {})
194: (8)                    assert G._adj == {}.fromkeys(self.G.nodes(), {})
195: (8)                    G = nx.create_empty_copy(self.G)
196: (8)                    assert nodes_equal(G, list(self.G))
197: (8)                    assert G.graph == self.G.graph
198: (8)                    assert G._node == self.G._node
199: (8)                    assert G._adj == {}.fromkeys(self.G.nodes(), {})
200: (4)                def test_degree_histogram(self):
201: (8)                    assert nx.degree_histogram(self.G) == [1, 1, 1, 1, 1]
202: (4)                def test_density(self):
203: (8)                    assert nx.density(self.G) == 0.5
204: (8)                    assert nx.density(self.DG) == 0.3
205: (8)                    G = nx.Graph()
206: (8)                    G.add_node(1)
207: (8)                    assert nx.density(G) == 0.0
208: (4)                def test_density_selfloop(self):
209: (8)                    G = nx.Graph()
210: (8)                    G.add_edge(1, 1)
211: (8)                    assert nx.density(G) == 0.0
212: (8)                    G.add_edge(1, 2)
213: (8)                    assert nx.density(G) == 2.0
214: (4)                def test_freeze(self):
215: (8)                    G = nx.freeze(self.G)
216: (8)                    assert G.frozen
217: (8)                    pytest.raises(nx.NetworkXError, G.add_node, 1)
218: (8)                    pytest.raises(nx.NetworkXError, G.add_nodes_from, [1])
219: (8)                    pytest.raises(nx.NetworkXError, G.remove_node, 1)
220: (8)                    pytest.raises(nx.NetworkXError, G.remove_nodes_from, [1])
```

```
(8)                 pytest.raises(nx.NetworkXError, G.add_edge, 1, 2)
(8)                 pytest.raises(nx.NetworkXError, G.add_edges_from, [(1, 2)])
(8)                 pytest.raises(nx.NetworkXError, G.remove_edge, 1, 2)
(8)                 pytest.raises(nx.NetworkXError, G.remove_edges_from, [(1, 2)])
(8)                 pytest.raises(nx.NetworkXError, G.clear_edges)
(8)                 pytest.raises(nx.NetworkXError, G.clear)
(4)             def test_is_frozen(self):
(8)                 assert not nx.is_frozen(self.G)
(8)                 G = nx.freeze(self.G)
(8)                 assert G.frozen == nx.is_frozen(self.G)
(8)                 assert G.frozen
(4)             def test_node_attributes_are_still_mutable_on_frozen_graph(self):
(8)                 G = nx.freeze(nx.path_graph(3))
(8)                 node = G.nodes[0]
(8)                 node["node_attribute"] = True
(8)                 assert node["node_attribute"] == True
(4)             def test_edge_attributes_are_still_mutable_on_frozen_graph(self):
(8)                 G = nx.freeze(nx.path_graph(3))
(8)                 edge = G.edges[(0, 1)]
(8)                 edge["edge_attribute"] = True
(8)                 assert edge["edge_attribute"] == True
(4)             def test_neighbors_complete_graph(self):
(8)                 graph = nx.complete_graph(100)
(8)                 pop = random.sample(list(graph), 1)
(8)                 nbors = list(nx.neighbors(graph, pop[0]))
(8)                 assert len(nbors) == len(graph) - 1
(8)                 graph = nx.path_graph(100)
(8)                 node = random.sample(list(graph), 1)[0]
(8)                 nbors = list(nx.neighbors(graph, node))
(8)                 if node != 0 and node != 99:
(12)                    assert len(nbors) == 2
(8)                 else:
(12)                    assert len(nbors) == 1
(8)                 graph = nx.star_graph(99)
(8)                 nbors = list(nx.neighbors(graph, 0))
(8)                 assert len(nbors) == 99
(4)             def test_non_neighbors(self):
(8)                 graph = nx.complete_graph(100)
(8)                 pop = random.sample(list(graph), 1)
(8)                 nbors = nx.non_neighbors(graph, pop[0])
(8)                 assert len(nbors) == 0
(8)                 graph = nx.path_graph(100)
(8)                 node = random.sample(list(graph), 1)[0]
(8)                 nbors = nx.non_neighbors(graph, node)
(8)                 if node != 0 and node != 99:
(12)                    assert len(nbors) == 97
(8)                 else:
(12)                    assert len(nbors) == 98
(8)                 graph = nx.star_graph(99)
(8)                 nbors = nx.non_neighbors(graph, 0)
(8)                 assert len(nbors) == 0
(8)                 graph = nx.Graph()
(8)                 graph.add_nodes_from(range(10))
(8)                 nbors = nx.non_neighbors(graph, 0)
(8)                 assert len(nbors) == 9
(4)             def test_non_edges(self):
(8)                 graph = nx.complete_graph(5)
(8)                 nedges = list(nx.non_edges(graph))
(8)                 assert len(nedges) == 0
(8)                 graph = nx.path_graph(4)
(8)                 expected = [(0, 2), (0, 3), (1, 3)]
(8)                 nedges = list(nx.non_edges(graph))
(8)                 for u, v in expected:
(12)                    assert (u, v) in nedges or (v, u) in nedges
(8)                 graph = nx.star_graph(4)
(8)                 expected = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
(8)                 nedges = list(nx.non_edges(graph))
(8)                 for u, v in expected:
(12)                    assert (u, v) in nedges or (v, u) in nedges
```

```
(8)                     graph = nx.DiGraph()
(8)                     graph.add_edges_from([(0, 2), (2, 0), (2, 1)])
(8)                     expected = [(0, 1), (1, 0), (1, 2)]
(8)                     nedges = list(nx.non_edges(graph))
(8)                     for e in expected:
(12)                        assert e in nedges
(4)                 def test_is_weighted(self):
(8)                     G = nx.Graph()
(8)                     assert not nx.is_weighted(G)
(8)                     G = nx.path_graph(4)
(8)                     assert not nx.is_weighted(G)
(8)                     assert not nx.is_weighted(G, (2, 3))
(8)                     G.add_node(4)
(8)                     G.add_edge(3, 4, weight=4)
(8)                     assert not nx.is_weighted(G)
(8)                     assert nx.is_weighted(G, (3, 4))
(8)                     G = nx.DiGraph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            ("0", "3", 3),
(16)                            ("0", "1", -5),
(16)                            ("1", "0", -5),
(16)                            ("0", "2", 2),
(16)                            ("1", "2", 4),
(16)                            ("2", "3", 1),
(12)                        ]
(8)                     )
(8)                     assert nx.is_weighted(G)
(8)                     assert nx.is_weighted(G, ("1", "0"))
(8)                     G = G.to_undirected()
(8)                     assert nx.is_weighted(G)
(8)                     assert nx.is_weighted(G, ("1", "0"))
(8)                     pytest.raises(nx.NetworkXError, nx.is_weighted, G, (1, 2))
(4)                 def test_is_negatively_weighted(self):
(8)                     G = nx.Graph()
(8)                     assert not nx.is_negatively_weighted(G)
(8)                     G.add_node(1)
(8)                     G.add_nodes_from([2, 3, 4, 5])
(8)                     assert not nx.is_negatively_weighted(G)
(8)                     G.add_edge(1, 2, weight=4)
(8)                     assert not nx.is_negatively_weighted(G, (1, 2))
(8)                     G.add_edges_from([(1, 3), (2, 4), (2, 6)])
(8)                     G[1][3]["color"] = "blue"
(8)                     assert not nx.is_negatively_weighted(G)
(8)                     assert not nx.is_negatively_weighted(G, (1, 3))
(8)                     G[2][4]["weight"] = -2
(8)                     assert nx.is_negatively_weighted(G, (2, 4))
(8)                     assert nx.is_negatively_weighted(G)
(8)                     G = nx.DiGraph()
(8)                     G.add_weighted_edges_from(
(12)                        [
(16)                            ("0", "3", 3),
(16)                            ("0", "1", -5),
(16)                            ("1", "0", -2),
(16)                            ("0", "2", 2),
(16)                            ("1", "2", -3),
(16)                            ("2", "3", 1),
(12)                        ]
(8)                     )
(8)                     assert nx.is_negatively_weighted(G)
(8)                     assert not nx.is_negatively_weighted(G, ("0", "3"))
(8)                     assert nx.is_negatively_weighted(G, ("1", "0"))
(8)                     pytest.raises(nx.NetworkXError, nx.is_negatively_weighted, G, (1, 4))
(0)             class TestCommonNeighbors:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.func = staticmethod(nx.common_neighbors)
(8)                     def test_func(G, u, v, expected):
(12)                        result = sorted(cls.func(G, u, v))
```

```
(12)                        assert result == expected
(8)                     cls.test = staticmethod(test_func)
(4)                 def test_K5(self):
(8)                     G = nx.complete_graph(5)
(8)                     self.test(G, 0, 1, [2, 3, 4])
(4)                 def test_P3(self):
(8)                     G = nx.path_graph(3)
(8)                     self.test(G, 0, 2, [1])
(4)                 def test_S4(self):
(8)                     G = nx.star_graph(4)
(8)                     self.test(G, 1, 2, [0])
(4)                 def test_digraph(self):
(8)                     with pytest.raises(nx.NetworkXNotImplemented):
(12)                        G = nx.DiGraph()
(12)                        G.add_edges_from([(0, 1), (1, 2)])
(12)                        self.func(G, 0, 2)
(4)                 def test_nonexistent_nodes(self):
(8)                     G = nx.complete_graph(5)
(8)                     pytest.raises(nx.NetworkXError, nx.common_neighbors, G, 5, 4)
(8)                     pytest.raises(nx.NetworkXError, nx.common_neighbors, G, 4, 5)
(8)                     pytest.raises(nx.NetworkXError, nx.common_neighbors, G, 5, 6)
(4)                 def test_custom1(self):
(8)                     """Case of no common neighbors."""
(8)                     G = nx.Graph()
(8)                     G.add_nodes_from([0, 1])
(8)                     self.test(G, 0, 1, [])
(4)                 def test_custom2(self):
(8)                     """Case of equal nodes."""
(8)                     G = nx.complete_graph(4)
(8)                     self.test(G, 0, 0, [1, 2, 3])
(0)             @pytest.mark.parametrize(
(4)                 "graph_type", (nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph)
(0)             )
(0)             def test_set_node_attributes(graph_type):
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 vals = 100
(4)                 attr = "hello"
(4)                 nx.set_node_attributes(G, vals, attr)
(4)                 assert G.nodes[0][attr] == vals
(4)                 assert G.nodes[1][attr] == vals
(4)                 assert G.nodes[2][attr] == vals
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 vals = dict(zip(sorted(G.nodes()), range(len(G))))
(4)                 attr = "hi"
(4)                 nx.set_node_attributes(G, vals, attr)
(4)                 assert G.nodes[0][attr] == 0
(4)                 assert G.nodes[1][attr] == 1
(4)                 assert G.nodes[2][attr] == 2
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 d = {"hi": 0, "hello": 200}
(4)                 vals = dict.fromkeys(G.nodes(), d)
(4)                 vals.pop(0)
(4)                 nx.set_node_attributes(G, vals)
(4)                 assert G.nodes[0] == {}
(4)                 assert G.nodes[1]["hi"] == 0
(4)                 assert G.nodes[2]["hello"] == 200
(0)             @pytest.mark.parametrize(
(4)                 ("values", "name"),
(4)                 (
(8)                     ({0: "red", 1: "blue"}, "color"),  # values dictionary
(8)                     ({0: {"color": "red"}, 1: {"color": "blue"}}, None),  # dict-of-dict
(4)                 ),
(0)             )
(0)             def test_set_node_attributes_ignores_extra_nodes(values, name):
(4)                 """
(4)                 When `values` is a dict or dict-of-dict keyed by nodes, ensure that keys
(4)                 that correspond to nodes not in G are ignored.
(4)                 """
(4)                 G = nx.Graph()
```

```
(4)                 G.add_node(0)
(4)                 nx.set_node_attributes(G, values, name)
(4)                 assert G.nodes[0]["color"] == "red"
(4)                 assert 1 not in G.nodes
(0)             @pytest.mark.parametrize("graph_type", (nx.Graph, nx.DiGraph))
(0)             def test_set_edge_attributes(graph_type):
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 attr = "hello"
(4)                 vals = 3
(4)                 nx.set_edge_attributes(G, vals, attr)
(4)                 assert G[0][1][attr] == vals
(4)                 assert G[1][2][attr] == vals
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 attr = "hi"
(4)                 edges = [(0, 1), (1, 2)]
(4)                 vals = dict(zip(edges, range(len(edges))))
(4)                 nx.set_edge_attributes(G, vals, attr)
(4)                 assert G[0][1][attr] == 0
(4)                 assert G[1][2][attr] == 1
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 d = {"hi": 0, "hello": 200}
(4)                 edges = [(0, 1)]
(4)                 vals = dict.fromkeys(edges, d)
(4)                 nx.set_edge_attributes(G, vals)
(4)                 assert G[0][1]["hi"] == 0
(4)                 assert G[0][1]["hello"] == 200
(4)                 assert G[1][2] == {}
(0)             @pytest.mark.parametrize(
(4)                 ("values", "name"),
(4)                 (
(8)                     ({(0, 1): 1.0, (0, 2): 2.0}, "weight"),  # values dict
(8)                     ({(0, 1): {"weight": 1.0}, (0, 2): {"weight": 2.0}}, None),  # values
dod
(4)                 ),
(0)             )
(0)             def test_set_edge_attributes_ignores_extra_edges(values, name):
(4)                 """If `values` is a dict or dict-of-dicts containing edges that are not in
(4)                 G, data associate with these edges should be ignored.
(4)                 """
(4)                 G = nx.Graph([(0, 1)])
(4)                 nx.set_edge_attributes(G, values, name)
(4)                 assert G[0][1]["weight"] == 1.0
(4)                 assert (0, 2) not in G.edges
(0)             @pytest.mark.parametrize("graph_type", (nx.MultiGraph, nx.MultiDiGraph))
(0)             def test_set_edge_attributes_multi(graph_type):
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 attr = "hello"
(4)                 vals = 3
(4)                 nx.set_edge_attributes(G, vals, attr)
(4)                 assert G[0][1][0][attr] == vals
(4)                 assert G[1][2][0][attr] == vals
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 attr = "hi"
(4)                 edges = [(0, 1, 0), (1, 2, 0)]
(4)                 vals = dict(zip(edges, range(len(edges))))
(4)                 nx.set_edge_attributes(G, vals, attr)
(4)                 assert G[0][1][0][attr] == 0
(4)                 assert G[1][2][0][attr] == 1
(4)                 G = nx.path_graph(3, create_using=graph_type)
(4)                 d = {"hi": 0, "hello": 200}
(4)                 edges = [(0, 1, 0)]
(4)                 vals = dict.fromkeys(edges, d)
(4)                 nx.set_edge_attributes(G, vals)
(4)                 assert G[0][1][0]["hi"] == 0
(4)                 assert G[0][1][0]["hello"] == 200
(4)                 assert G[1][2][0] == {}
(0)             @pytest.mark.parametrize(
(4)                 ("values", "name"),
(4)                 (
```

```
496: (8)                    ({(0, 1, 0): 1.0, (0, 2, 0): 2.0}, "weight"),  # values dict
497: (8)                    ({(0, 1, 0): {"weight": 1.0}, (0, 2, 0): {"weight": 2.0}}, None),  #
values dod
498: (4)                ),
499: (0)            )
500: (0)            def test_set_edge_attributes_multi_ignores_extra_edges(values, name):
501: (4)                """If `values` is a dict or dict-of-dicts containing edges that are not in
502: (4)                G, data associate with these edges should be ignored.
503: (4)                """
504: (4)                G = nx.MultiGraph([(0, 1, 0), (0, 1, 1)])
505: (4)                nx.set_edge_attributes(G, values, name)
506: (4)                assert G[0][1][0]["weight"] == 1.0
507: (4)                assert G[0][1][1] == {}
508: (4)                assert (0, 2) not in G.edges()
509: (0)            def test_get_node_attributes():
510: (4)                graphs = [nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph()]
511: (4)                for G in graphs:
512: (8)                    G = nx.path_graph(3, create_using=G)
513: (8)                    attr = "hello"
514: (8)                    vals = 100
515: (8)                    nx.set_node_attributes(G, vals, attr)
516: (8)                    attrs = nx.get_node_attributes(G, attr)
517: (8)                    assert attrs[0] == vals
518: (8)                    assert attrs[1] == vals
519: (8)                    assert attrs[2] == vals
520: (8)                    default_val = 1
521: (8)                    G.add_node(4)
522: (8)                    attrs = nx.get_node_attributes(G, attr, default=default_val)
523: (8)                    assert attrs[4] == default_val
524: (0)            def test_get_edge_attributes():
525: (4)                graphs = [nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph()]
526: (4)                for G in graphs:
527: (8)                    G = nx.path_graph(3, create_using=G)
528: (8)                    attr = "hello"
529: (8)                    vals = 100
530: (8)                    nx.set_edge_attributes(G, vals, attr)
531: (8)                    attrs = nx.get_edge_attributes(G, attr)
532: (8)                    assert len(attrs) == 2
533: (8)                    for edge in G.edges:
534: (12)                       assert attrs[edge] == vals
535: (8)                    default_val = vals
536: (8)                    G.add_edge(4, 5)
537: (8)                    deafult_attrs = nx.get_edge_attributes(G, attr, default=default_val)
538: (8)                    assert len(deafult_attrs) == 3
539: (8)                    for edge in G.edges:
540: (12)                       assert deafult_attrs[edge] == vals
541: (0)            def test_is_empty():
542: (4)                graphs = [nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph()]
543: (4)                for G in graphs:
544: (8)                    assert nx.is_empty(G)
545: (8)                    G.add_nodes_from(range(5))
546: (8)                    assert nx.is_empty(G)
547: (8)                    G.add_edges_from([(1, 2), (3, 4)])
548: (8)                    assert not nx.is_empty(G)
549: (0)            @pytest.mark.parametrize(
550: (4)                "graph_type", [nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph]
551: (0)            )
552: (0)            def test_selfloops(graph_type):
553: (4)                G = nx.complete_graph(3, create_using=graph_type)
554: (4)                G.add_edge(0, 0)
555: (4)                assert nodes_equal(nx.nodes_with_selfloops(G), [0])
556: (4)                assert edges_equal(nx.selfloop_edges(G), [(0, 0)])
557: (4)                assert edges_equal(nx.selfloop_edges(G, data=True), [(0, 0, {})])
558: (4)                assert nx.number_of_selfloops(G) == 1
559: (0)            @pytest.mark.parametrize(
560: (4)                "graph_type", [nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph]
561: (0)            )
562: (0)            def test_selfloop_edges_attr(graph_type):
563: (4)                G = nx.complete_graph(3, create_using=graph_type)
```

```
564: (4)                G.add_edge(0, 0)
565: (4)                G.add_edge(1, 1, weight=2)
566: (4)                assert edges_equal(
567: (8)                    nx.selfloop_edges(G, data=True), [(0, 0, {}), (1, 1, {"weight": 2})]
568: (4)                )
569: (4)                assert edges_equal(nx.selfloop_edges(G, data="weight"), [(0, 0, None), (1,
1, 2)])
570: (0)            def test_selfloop_edges_multi_with_data_and_keys():
571: (4)                G = nx.complete_graph(3, create_using=nx.MultiGraph)
572: (4)                G.add_edge(0, 0, weight=10)
573: (4)                G.add_edge(0, 0, weight=100)
574: (4)                assert edges_equal(
575: (8)                    nx.selfloop_edges(G, data="weight", keys=True), [(0, 0, 0, 10), (0, 0,
1, 100)]
576: (4)                )
577: (0)            @pytest.mark.parametrize("graph_type", [nx.Graph, nx.DiGraph])
578: (0)            def test_selfloops_removal(graph_type):
579: (4)                G = nx.complete_graph(3, create_using=graph_type)
580: (4)                G.add_edge(0, 0)
581: (4)                G.remove_edges_from(nx.selfloop_edges(G, keys=True))
582: (4)                G.add_edge(0, 0)
583: (4)                G.remove_edges_from(nx.selfloop_edges(G, data=True))
584: (4)                G.add_edge(0, 0)
585: (4)                G.remove_edges_from(nx.selfloop_edges(G, keys=True, data=True))
586: (0)            @pytest.mark.parametrize("graph_type", [nx.MultiGraph, nx.MultiDiGraph])
587: (0)            def test_selfloops_removal_multi(graph_type):
588: (4)                """test removing selfloops behavior vis-a-vis altering a dict while
iterating.
589: (4)                cf. gh-4068"""
590: (4)                G = nx.complete_graph(3, create_using=graph_type)
591: (4)                G.add_edge(0, 0)
592: (4)                G.add_edge(0, 0)
593: (4)                G.remove_edges_from(nx.selfloop_edges(G))
594: (4)                assert (0, 0) not in G.edges()
595: (4)                G.add_edge(0, 0)
596: (4)                G.add_edge(0, 0)
597: (4)                with pytest.raises(RuntimeError):
598: (8)                    G.remove_edges_from(nx.selfloop_edges(G, keys=True))
599: (4)                G.add_edge(0, 0)
600: (4)                G.add_edge(0, 0)
601: (4)                with pytest.raises(TypeError):
602: (8)                    G.remove_edges_from(nx.selfloop_edges(G, data=True))
603: (4)                G.add_edge(0, 0)
604: (4)                G.add_edge(0, 0)
605: (4)                with pytest.raises(RuntimeError):
606: (8)                    G.remove_edges_from(nx.selfloop_edges(G, data=True, keys=True))
607: (0)            def test_pathweight():
608: (4)                valid_path = [1, 2, 3]
609: (4)                invalid_path = [1, 3, 2]
610: (4)                graphs = [nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph()]
611: (4)                edges = [
612: (8)                    (1, 2, {"cost": 5, "dist": 6}),
613: (8)                    (2, 3, {"cost": 3, "dist": 4}),
614: (8)                    (1, 2, {"cost": 1, "dist": 2}),
615: (4)                ]
616: (4)                for graph in graphs:
617: (8)                    graph.add_edges_from(edges)
618: (8)                    assert nx.path_weight(graph, valid_path, "cost") == 4
619: (8)                    assert nx.path_weight(graph, valid_path, "dist") == 6
620: (8)                    pytest.raises(nx.NetworkXNoPath, nx.path_weight, graph, invalid_path,
"cost")
621: (0)            @pytest.mark.parametrize(
622: (4)                "G", (nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph())
623: (0)            )
624: (0)            def test_ispath(G):
625: (4)                G.add_edges_from([(1, 2), (2, 3), (1, 2), (3, 4)])
626: (4)                valid_path = [1, 2, 3, 4]
627: (4)                invalid_path = [1, 2, 4, 3]  # wrong node order
628: (4)                another_invalid_path = [1, 2, 3, 4, 5]  # contains node not in G
```

```
629: (4)                assert nx.is_path(G, valid_path)
630: (4)                assert not nx.is_path(G, invalid_path)
631: (4)                assert not nx.is_path(G, another_invalid_path)
632: (0)            @pytest.mark.parametrize("G", (nx.Graph(), nx.DiGraph()))
633: (0)            def test_restricted_view(G):
634: (4)                G.add_edges_from([(0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2)])
635: (4)                G.add_node(4)
636: (4)                H = nx.restricted_view(G, [0, 2, 5], [(1, 2), (3, 4)])
637: (4)                assert set(H.nodes()) == {1, 3, 4}
638: (4)                assert set(H.edges()) == {(1, 1)}
639: (0)            @pytest.mark.parametrize("G", (nx.MultiGraph(), nx.MultiDiGraph()))
640: (0)            def test_restricted_view_multi(G):
641: (4)                G.add_edges_from(
642: (8)                    [(0, 1, 0), (0, 2, 0), (0, 3, 0), (0, 1, 1), (1, 0, 0), (1, 1, 0), (1,
2, 0)]
643: (4)                )
644: (4)                G.add_node(4)
645: (4)                H = nx.restricted_view(G, [0, 2, 5], [(1, 2, 0), (3, 4, 0)])
646: (4)                assert set(H.nodes()) == {1, 3, 4}
647: (4)                assert set(H.edges()) == {(1, 1)}
```

File 407 - test_graphviews.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import edges_equal, nodes_equal
4: (0)              class TestReverseView:
5: (4)                  def setup_method(self):
6: (8)                      self.G = nx.path_graph(9, create_using=nx.DiGraph())
7: (8)                      self.rv = nx.reverse_view(self.G)
8: (4)                  def test_pickle(self):
9: (8)                      import pickle
10: (8)                     rv = self.rv
11: (8)                     prv = pickle.loads(pickle.dumps(rv, -1))
12: (8)                     assert rv._node == prv._node
13: (8)                     assert rv._adj == prv._adj
14: (8)                     assert rv.graph == prv.graph
15: (4)                 def test_contains(self):
16: (8)                     assert (2, 3) in self.G.edges
17: (8)                     assert (3, 2) not in self.G.edges
18: (8)                     assert (2, 3) not in self.rv.edges
19: (8)                     assert (3, 2) in self.rv.edges
20: (4)                 def test_iter(self):
21: (8)                     expected = sorted(tuple(reversed(e)) for e in self.G.edges)
22: (8)                     assert sorted(self.rv.edges) == expected
23: (4)                 def test_exceptions(self):
24: (8)                     G = nx.Graph()
25: (8)                     pytest.raises(nx.NetworkXNotImplemented, nx.reverse_view, G)
26: (4)                 def test_subclass(self):
27: (8)                     class MyGraph(nx.DiGraph):
28: (12)                        def my_method(self):
29: (16)                            return "me"
30: (12)                        def to_directed_class(self):
31: (16)                            return MyGraph()
32: (8)                     M = MyGraph()
33: (8)                     M.add_edge(1, 2)
34: (8)                     RM = nx.reverse_view(M)
35: (8)                     print("RM class", RM.__class__)
36: (8)                     RMC = RM.copy()
37: (8)                     print("RMC class", RMC.__class__)
38: (8)                     print(RMC.edges)
39: (8)                     assert RMC.has_edge(2, 1)
40: (8)                     assert RMC.my_method() == "me"
41: (0)             class TestMultiReverseView:
42: (4)                 def setup_method(self):
43: (8)                     self.G = nx.path_graph(9, create_using=nx.MultiDiGraph())
44: (8)                     self.G.add_edge(4, 5)
```

```
45: (8)                     self.rv = nx.reverse_view(self.G)
46: (4)                 def test_pickle(self):
47: (8)                     import pickle
48: (8)                     rv = self.rv
49: (8)                     prv = pickle.loads(pickle.dumps(rv, -1))
50: (8)                     assert rv._node == prv._node
51: (8)                     assert rv._adj == prv._adj
52: (8)                     assert rv.graph == prv.graph
53: (4)                 def test_contains(self):
54: (8)                     assert (2, 3, 0) in self.G.edges
55: (8)                     assert (3, 2, 0) not in self.G.edges
56: (8)                     assert (2, 3, 0) not in self.rv.edges
57: (8)                     assert (3, 2, 0) in self.rv.edges
58: (8)                     assert (5, 4, 1) in self.rv.edges
59: (8)                     assert (4, 5, 1) not in self.rv.edges
60: (4)                 def test_iter(self):
61: (8)                     expected = sorted((v, u, k) for u, v, k in self.G.edges)
62: (8)                     assert sorted(self.rv.edges) == expected
63: (4)                 def test_exceptions(self):
64: (8)                     MG = nx.MultiGraph(self.G)
65: (8)                     pytest.raises(nx.NetworkXNotImplemented, nx.reverse_view, MG)
66: (0)             def test_generic_multitype():
67: (4)                 nxg = nx.graphviews
68: (4)                 G = nx.DiGraph([(1, 2)])
69: (4)                 with pytest.raises(nx.NetworkXError):
70: (8)                     nxg.generic_graph_view(G, create_using=nx.MultiGraph)
71: (4)                 G = nx.MultiDiGraph([(1, 2)])
72: (4)                 with pytest.raises(nx.NetworkXError):
73: (8)                     nxg.generic_graph_view(G, create_using=nx.DiGraph)
74: (0)             class TestToDirected:
75: (4)                 def setup_method(self):
76: (8)                     self.G = nx.path_graph(9)
77: (8)                     self.dv = nx.to_directed(self.G)
78: (8)                     self.MG = nx.path_graph(9, create_using=nx.MultiGraph())
79: (8)                     self.Mdv = nx.to_directed(self.MG)
80: (4)                 def test_directed(self):
81: (8)                     assert not self.G.is_directed()
82: (8)                     assert self.dv.is_directed()
83: (4)                 def test_already_directed(self):
84: (8)                     dd = nx.to_directed(self.dv)
85: (8)                     Mdd = nx.to_directed(self.Mdv)
86: (8)                     assert edges_equal(dd.edges, self.dv.edges)
87: (8)                     assert edges_equal(Mdd.edges, self.Mdv.edges)
88: (4)                 def test_pickle(self):
89: (8)                     import pickle
90: (8)                     dv = self.dv
91: (8)                     pdv = pickle.loads(pickle.dumps(dv, -1))
92: (8)                     assert dv._node == pdv._node
93: (8)                     assert dv._succ == pdv._succ
94: (8)                     assert dv._pred == pdv._pred
95: (8)                     assert dv.graph == pdv.graph
96: (4)                 def test_contains(self):
97: (8)                     assert (2, 3) in self.G.edges
98: (8)                     assert (3, 2) in self.G.edges
99: (8)                     assert (2, 3) in self.dv.edges
100: (8)                    assert (3, 2) in self.dv.edges
101: (4)                def test_iter(self):
102: (8)                    revd = [tuple(reversed(e)) for e in self.G.edges]
103: (8)                    expected = sorted(list(self.G.edges) + revd)
104: (8)                    assert sorted(self.dv.edges) == expected
105: (0)            class TestToUndirected:
106: (4)                def setup_method(self):
107: (8)                    self.DG = nx.path_graph(9, create_using=nx.DiGraph())
108: (8)                    self.uv = nx.to_undirected(self.DG)
109: (8)                    self.MDG = nx.path_graph(9, create_using=nx.MultiDiGraph())
110: (8)                    self.Muv = nx.to_undirected(self.MDG)
111: (4)                def test_directed(self):
112: (8)                    assert self.DG.is_directed()
113: (8)                    assert not self.uv.is_directed()
```

```
    def test_already_directed(self):
        uu = nx.to_undirected(self.uv)
        Muu = nx.to_undirected(self.Muv)
        assert edges_equal(uu.edges, self.uv.edges)
        assert edges_equal(Muu.edges, self.Muv.edges)
    def test_pickle(self):
        import pickle
        uv = self.uv
        puv = pickle.loads(pickle.dumps(uv, -1))
        assert uv._node == puv._node
        assert uv._adj == puv._adj
        assert uv.graph == puv.graph
        assert hasattr(uv, "_graph")
    def test_contains(self):
        assert (2, 3) in self.DG.edges
        assert (3, 2) not in self.DG.edges
        assert (2, 3) in self.uv.edges
        assert (3, 2) in self.uv.edges
    def test_iter(self):
        expected = sorted(self.DG.edges)
        assert sorted(self.uv.edges) == expected
class TestChainsOfViews:
    @classmethod
    def setup_class(cls):
        cls.G = nx.path_graph(9)
        cls.DG = nx.path_graph(9, create_using=nx.DiGraph())
        cls.MG = nx.path_graph(9, create_using=nx.MultiGraph())
        cls.MDG = nx.path_graph(9, create_using=nx.MultiDiGraph())
        cls.Gv = nx.to_undirected(cls.DG)
        cls.DGv = nx.to_directed(cls.G)
        cls.MGv = nx.to_undirected(cls.MDG)
        cls.MDGv = nx.to_directed(cls.MG)
        cls.Rv = cls.DG.reverse()
        cls.MRv = cls.MDG.reverse()
        cls.graphs = [
            cls.G,
            cls.DG,
            cls.MG,
            cls.MDG,
            cls.Gv,
            cls.DGv,
            cls.MGv,
            cls.MDGv,
            cls.Rv,
            cls.MRv,
        ]
        for G in cls.graphs:
            G.edges, G.nodes, G.degree
    def test_pickle(self):
        import pickle
        for G in self.graphs:
            H = pickle.loads(pickle.dumps(G, -1))
            assert edges_equal(H.edges, G.edges)
            assert nodes_equal(H.nodes, G.nodes)
    def test_subgraph_of_subgraph(self):
        SGv = nx.subgraph(self.G, range(3, 7))
        SDGv = nx.subgraph(self.DG, range(3, 7))
        SMGv = nx.subgraph(self.MG, range(3, 7))
        SMDGv = nx.subgraph(self.MDG, range(3, 7))
        for G in self.graphs + [SGv, SDGv, SMGv, SMDGv]:
            SG = nx.induced_subgraph(G, [4, 5, 6])
            assert list(SG) == [4, 5, 6]
            SSG = SG.subgraph([6, 7])
            assert list(SSG) == [6]
            assert SSG._graph is G
    def test_restricted_induced_subgraph_chains(self):
        """Test subgraph chains that both restrict and show nodes/edges.
        A restricted_view subgraph should allow induced subgraphs using
        G.subgraph that automagically without a chain (meaning the result
```

```
183: (8)                    is a subgraph view of the original graph not a subgraph-of-subgraph.
184: (8)                    """
185: (8)                    hide_nodes = [3, 4, 5]
186: (8)                    hide_edges = [(6, 7)]
187: (8)                    RG = nx.restricted_view(self.G, hide_nodes, hide_edges)
188: (8)                    nodes = [4, 5, 6, 7, 8]
189: (8)                    SG = nx.induced_subgraph(RG, nodes)
190: (8)                    SSG = RG.subgraph(nodes)
191: (8)                    assert RG._graph is self.G
192: (8)                    assert SSG._graph is self.G
193: (8)                    assert SG._graph is RG
194: (8)                    assert edges_equal(SG.edges, SSG.edges)
195: (8)                    CG = self.G.copy()
196: (8)                    CG.remove_nodes_from(hide_nodes)
197: (8)                    CG.remove_edges_from(hide_edges)
198: (8)                    assert edges_equal(CG.edges(nodes), SSG.edges)
199: (8)                    CG.remove_nodes_from([0, 1, 2, 3])
200: (8)                    assert edges_equal(CG.edges, SSG.edges)
201: (8)                    SSSG = self.G.subgraph(nodes)
202: (8)                    RSG = nx.restricted_view(SSSG, hide_nodes, hide_edges)
203: (8)                    assert RSG._graph is not self.G
204: (8)                    assert edges_equal(RSG.edges, CG.edges)
205: (4)                def test_subgraph_copy(self):
206: (8)                    for origG in self.graphs:
207: (12)                       G = nx.Graph(origG)
208: (12)                       SG = G.subgraph([4, 5, 6])
209: (12)                       H = SG.copy()
210: (12)                       assert type(G) == type(H)
211: (4)                def test_subgraph_todirected(self):
212: (8)                    SG = nx.induced_subgraph(self.G, [4, 5, 6])
213: (8)                    SSG = SG.to_directed()
214: (8)                    assert sorted(SSG) == [4, 5, 6]
215: (8)                    assert sorted(SSG.edges) == [(4, 5), (5, 4), (5, 6), (6, 5)]
216: (4)                def test_subgraph_toundirected(self):
217: (8)                    SG = nx.induced_subgraph(self.G, [4, 5, 6])
218: (8)                    SSG = SG.to_undirected()
219: (8)                    assert list(SSG) == [4, 5, 6]
220: (8)                    assert sorted(SSG.edges) == [(4, 5), (5, 6)]
221: (4)                def test_reverse_subgraph_toundirected(self):
222: (8)                    G = self.DG.reverse(copy=False)
223: (8)                    SG = G.subgraph([4, 5, 6])
224: (8)                    SSG = SG.to_undirected()
225: (8)                    assert list(SSG) == [4, 5, 6]
226: (8)                    assert sorted(SSG.edges) == [(4, 5), (5, 6)]
227: (4)                def test_reverse_reverse_copy(self):
228: (8)                    G = self.DG.reverse(copy=False)
229: (8)                    H = G.reverse(copy=True)
230: (8)                    assert H.nodes == self.DG.nodes
231: (8)                    assert H.edges == self.DG.edges
232: (8)                    G = self.MDG.reverse(copy=False)
233: (8)                    H = G.reverse(copy=True)
234: (8)                    assert H.nodes == self.MDG.nodes
235: (8)                    assert H.edges == self.MDG.edges
236: (4)                def test_subgraph_edgesubgraph_toundirected(self):
237: (8)                    G = self.G.copy()
238: (8)                    SG = G.subgraph([4, 5, 6])
239: (8)                    SSG = SG.edge_subgraph([(4, 5), (5, 4)])
240: (8)                    USSG = SSG.to_undirected()
241: (8)                    assert list(USSG) == [4, 5]
242: (8)                    assert sorted(USSG.edges) == [(4, 5)]
243: (4)                def test_copy_subgraph(self):
244: (8)                    G = self.G.copy()
245: (8)                    SG = G.subgraph([4, 5, 6])
246: (8)                    CSG = SG.copy(as_view=True)
247: (8)                    DCSG = SG.copy(as_view=False)
248: (8)                    assert hasattr(CSG, "_graph")  # is a view
249: (8)                    assert not hasattr(DCSG, "_graph")  # not a view
250: (4)                def test_copy_disubgraph(self):
251: (8)                    G = self.DG.copy()
```

```
252: (8)                    SG = G.subgraph([4, 5, 6])
253: (8)                    CSG = SG.copy(as_view=True)
254: (8)                    DCSG = SG.copy(as_view=False)
255: (8)                    assert hasattr(CSG, "_graph")  # is a view
256: (8)                    assert not hasattr(DCSG, "_graph")  # not a view
257: (4)                def test_copy_multidisubgraph(self):
258: (8)                    G = self.MDG.copy()
259: (8)                    SG = G.subgraph([4, 5, 6])
260: (8)                    CSG = SG.copy(as_view=True)
261: (8)                    DCSG = SG.copy(as_view=False)
262: (8)                    assert hasattr(CSG, "_graph")  # is a view
263: (8)                    assert not hasattr(DCSG, "_graph")  # not a view
264: (4)                def test_copy_multisubgraph(self):
265: (8)                    G = self.MG.copy()
266: (8)                    SG = G.subgraph([4, 5, 6])
267: (8)                    CSG = SG.copy(as_view=True)
268: (8)                    DCSG = SG.copy(as_view=False)
269: (8)                    assert hasattr(CSG, "_graph")  # is a view
270: (8)                    assert not hasattr(DCSG, "_graph")  # not a view
271: (4)                def test_copy_of_view(self):
272: (8)                    G = nx.MultiGraph(self.MGv)
273: (8)                    assert G.__class__.__name__ == "MultiGraph"
274: (8)                    G = G.copy(as_view=True)
275: (8)                    assert G.__class__.__name__ == "MultiGraph"
276: (4)                def test_subclass(self):
277: (8)                    class MyGraph(nx.DiGraph):
278: (12)                       def my_method(self):
279: (16)                           return "me"
280: (12)                       def to_directed_class(self):
281: (16)                           return MyGraph()
282: (8)                    for origG in self.graphs:
283: (12)                       G = MyGraph(origG)
284: (12)                       SG = G.subgraph([4, 5, 6])
285: (12)                       H = SG.copy()
286: (12)                       assert SG.my_method() == "me"
287: (12)                       assert H.my_method() == "me"
288: (12)                       assert 3 not in H or 3 in SG
```

File 408 - test_graph_historical.py:

```
1: (0)              """Original NetworkX graph tests"""
2: (0)              import networkx
3: (0)              import networkx as nx
4: (0)              from .historical_tests import HistoricalTests
5: (0)              class TestGraphHistorical(HistoricalTests):
6: (4)                  @classmethod
7: (4)                  def setup_class(cls):
8: (8)                      HistoricalTests.setup_class()
9: (8)                      cls.G = nx.Graph
```

File 409 - test_multidigraph.py:

```
1: (0)              from collections import UserDict
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import edges_equal
5: (0)              from .test_multigraph import BaseMultiGraphTester
6: (0)              from .test_multigraph import TestEdgeSubgraph as _TestMultiGraphEdgeSubgraph
7: (0)              from .test_multigraph import TestMultiGraph as _TestMultiGraph
8: (0)              class BaseMultiDiGraphTester(BaseMultiGraphTester):
9: (4)                  def test_edges(self):
10: (8)                     G = self.K3
11: (8)                     edges = [(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
12: (8)                     assert sorted(G.edges()) == edges
13: (8)                     assert sorted(G.edges(0)) == [(0, 1), (0, 2)]
```

```
 14: (8)                     pytest.raises((KeyError, nx.NetworkXError), G.edges, -1)
 15: (4)                 def test_edges_data(self):
 16: (8)                     G = self.K3
 17: (8)                     edges = [(0, 1, {}), (0, 2, {}), (1, 0, {}), (1, 2, {}), (2, 0, {}),
 (2, 1, {})]
 18: (8)                     assert sorted(G.edges(data=True)) == edges
 19: (8)                     assert sorted(G.edges(0, data=True)) == [(0, 1, {}), (0, 2, {})]
 20: (8)                     pytest.raises((KeyError, nx.NetworkXError), G.neighbors, -1)
 21: (4)                 def test_edges_multi(self):
 22: (8)                     G = self.K3
 23: (8)                     assert sorted(G.edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2, 0),
 (2, 1)]
 24: (8)                     assert sorted(G.edges(0)) == [(0, 1), (0, 2)]
 25: (8)                     G.add_edge(0, 1)
 26: (8)                     assert sorted(G.edges()) == [
 27: (12)                        (0, 1),
 28: (12)                        (0, 1),
 29: (12)                        (0, 2),
 30: (12)                        (1, 0),
 31: (12)                        (1, 2),
 32: (12)                        (2, 0),
 33: (12)                        (2, 1),
 34: (8)                     ]
 35: (4)                 def test_out_edges(self):
 36: (8)                     G = self.K3
 37: (8)                     assert sorted(G.out_edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2,
 0), (2, 1)]
 38: (8)                     assert sorted(G.out_edges(0)) == [(0, 1), (0, 2)]
 39: (8)                     pytest.raises((KeyError, nx.NetworkXError), G.out_edges, -1)
 40: (8)                     assert sorted(G.out_edges(0, keys=True)) == [(0, 1, 0), (0, 2, 0)]
 41: (4)                 def test_out_edges_multi(self):
 42: (8)                     G = self.K3
 43: (8)                     assert sorted(G.out_edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2,
 0), (2, 1)]
 44: (8)                     assert sorted(G.out_edges(0)) == [(0, 1), (0, 2)]
 45: (8)                     G.add_edge(0, 1, 2)
 46: (8)                     assert sorted(G.out_edges()) == [
 47: (12)                        (0, 1),
 48: (12)                        (0, 1),
 49: (12)                        (0, 2),
 50: (12)                        (1, 0),
 51: (12)                        (1, 2),
 52: (12)                        (2, 0),
 53: (12)                        (2, 1),
 54: (8)                     ]
 55: (4)                 def test_out_edges_data(self):
 56: (8)                     G = self.K3
 57: (8)                     assert sorted(G.edges(0, data=True)) == [(0, 1, {}), (0, 2, {})]
 58: (8)                     G.remove_edge(0, 1)
 59: (8)                     G.add_edge(0, 1, data=1)
 60: (8)                     assert sorted(G.edges(0, data=True)) == [(0, 1, {"data": 1}), (0, 2,
 {})]
 61: (8)                     assert sorted(G.edges(0, data="data")) == [(0, 1, 1), (0, 2, None)]
 62: (8)                     assert sorted(G.edges(0, data="data", default=-1)) == [(0, 1, 1), (0,
 2, -1)]
 63: (4)                 def test_in_edges(self):
 64: (8)                     G = self.K3
 65: (8)                     assert sorted(G.in_edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2,
 0), (2, 1)]
 66: (8)                     assert sorted(G.in_edges(0)) == [(1, 0), (2, 0)]
 67: (8)                     pytest.raises((KeyError, nx.NetworkXError), G.in_edges, -1)
 68: (8)                     G.add_edge(0, 1, 2)
 69: (8)                     assert sorted(G.in_edges()) == [
 70: (12)                        (0, 1),
 71: (12)                        (0, 1),
 72: (12)                        (0, 2),
 73: (12)                        (1, 0),
 74: (12)                        (1, 2),
 75: (12)                        (2, 0),
```

```
 76: (12)                        (2, 1),
 77: (8)                     ]
 78: (8)                     assert sorted(G.in_edges(0, keys=True)) == [(1, 0, 0), (2, 0, 0)]
 79: (4)                 def test_in_edges_no_keys(self):
 80: (8)                     G = self.K3
 81: (8)                     assert sorted(G.in_edges()) == [(0, 1), (0, 2), (1, 0), (1, 2), (2,
 0), (2, 1)]
 82: (8)                     assert sorted(G.in_edges(0)) == [(1, 0), (2, 0)]
 83: (8)                     G.add_edge(0, 1, 2)
 84: (8)                     assert sorted(G.in_edges()) == [
 85: (12)                        (0, 1),
 86: (12)                        (0, 1),
 87: (12)                        (0, 2),
 88: (12)                        (1, 0),
 89: (12)                        (1, 2),
 90: (12)                        (2, 0),
 91: (12)                        (2, 1),
 92: (8)                     ]
 93: (8)                     assert sorted(G.in_edges(data=True, keys=False)) == [
 94: (12)                        (0, 1, {}),
 95: (12)                        (0, 1, {}),
 96: (12)                        (0, 2, {}),
 97: (12)                        (1, 0, {}),
 98: (12)                        (1, 2, {}),
 99: (12)                        (2, 0, {}),
 100: (12)                       (2, 1, {}),
 101: (8)                    ]
 102: (4)                def test_in_edges_data(self):
 103: (8)                    G = self.K3
 104: (8)                    assert sorted(G.in_edges(0, data=True)) == [(1, 0, {}), (2, 0, {})]
 105: (8)                    G.remove_edge(1, 0)
 106: (8)                    G.add_edge(1, 0, data=1)
 107: (8)                    assert sorted(G.in_edges(0, data=True)) == [(1, 0, {"data": 1}), (2,
 0, {})]
 108: (8)                    assert sorted(G.in_edges(0, data="data")) == [(1, 0, 1), (2, 0, None)]
 109: (8)                    assert sorted(G.in_edges(0, data="data", default=-1)) == [(1, 0, 1),
 (2, 0, -1)]
 110: (4)                def is_shallow(self, H, G):
 111: (8)                    assert G.graph["foo"] == H.graph["foo"]
 112: (8)                    G.graph["foo"].append(1)
 113: (8)                    assert G.graph["foo"] == H.graph["foo"]
 114: (8)                    assert G.nodes[0]["foo"] == H.nodes[0]["foo"]
 115: (8)                    G.nodes[0]["foo"].append(1)
 116: (8)                    assert G.nodes[0]["foo"] == H.nodes[0]["foo"]
 117: (8)                    assert G[1][2][0]["foo"] == H[1][2][0]["foo"]
 118: (8)                    G[1][2][0]["foo"].append(1)
 119: (8)                    assert G[1][2][0]["foo"] == H[1][2][0]["foo"]
 120: (4)                def is_deep(self, H, G):
 121: (8)                    assert G.graph["foo"] == H.graph["foo"]
 122: (8)                    G.graph["foo"].append(1)
 123: (8)                    assert G.graph["foo"] != H.graph["foo"]
 124: (8)                    assert G.nodes[0]["foo"] == H.nodes[0]["foo"]
 125: (8)                    G.nodes[0]["foo"].append(1)
 126: (8)                    assert G.nodes[0]["foo"] != H.nodes[0]["foo"]
 127: (8)                    assert G[1][2][0]["foo"] == H[1][2][0]["foo"]
 128: (8)                    G[1][2][0]["foo"].append(1)
 129: (8)                    assert G[1][2][0]["foo"] != H[1][2][0]["foo"]
 130: (4)                def test_to_undirected(self):
 131: (8)                    G = self.K3
 132: (8)                    self.add_attributes(G)
 133: (8)                    H = nx.MultiGraph(G)
 134: (8)                    try:
 135: (12)                       assert edges_equal(H.edges(), [(0, 1), (1, 2), (2, 0)])
 136: (8)                    except AssertionError:
 137: (12)                       assert edges_equal(H.edges(), [(0, 1), (1, 2), (1, 2), (2, 0)])
 138: (8)                    H = G.to_undirected()
 139: (8)                    self.is_deep(H, G)
 140: (4)                def test_has_successor(self):
 141: (8)                    G = self.K3
```

```
(8)                 assert G.has_successor(0, 1)
(8)                 assert not G.has_successor(0, -1)
(4)             def test_successors(self):
(8)                 G = self.K3
(8)                 assert sorted(G.successors(0)) == [1, 2]
(8)                 pytest.raises((KeyError, nx.NetworkXError), G.successors, -1)
(4)             def test_has_predecessor(self):
(8)                 G = self.K3
(8)                 assert G.has_predecessor(0, 1)
(8)                 assert not G.has_predecessor(0, -1)
(4)             def test_predecessors(self):
(8)                 G = self.K3
(8)                 assert sorted(G.predecessors(0)) == [1, 2]
(8)                 pytest.raises((KeyError, nx.NetworkXError), G.predecessors, -1)
(4)             def test_degree(self):
(8)                 G = self.K3
(8)                 assert sorted(G.degree()) == [(0, 4), (1, 4), (2, 4)]
(8)                 assert dict(G.degree()) == {0: 4, 1: 4, 2: 4}
(8)                 assert G.degree(0) == 4
(8)                 assert list(G.degree(iter([0]))) == [(0, 4)]
(8)                 G.add_edge(0, 1, weight=0.3, other=1.2)
(8)                 assert sorted(G.degree(weight="weight")) == [(0, 4.3), (1, 4.3), (2,
4)]
(8)                 assert sorted(G.degree(weight="other")) == [(0, 5.2), (1, 5.2), (2,
4)]
(4)             def test_in_degree(self):
(8)                 G = self.K3
(8)                 assert sorted(G.in_degree()) == [(0, 2), (1, 2), (2, 2)]
(8)                 assert dict(G.in_degree()) == {0: 2, 1: 2, 2: 2}
(8)                 assert G.in_degree(0) == 2
(8)                 assert list(G.in_degree(iter([0]))) == [(0, 2)]
(8)                 assert G.in_degree(0, weight="weight") == 2
(4)             def test_out_degree(self):
(8)                 G = self.K3
(8)                 assert sorted(G.out_degree()) == [(0, 2), (1, 2), (2, 2)]
(8)                 assert dict(G.out_degree()) == {0: 2, 1: 2, 2: 2}
(8)                 assert G.out_degree(0) == 2
(8)                 assert list(G.out_degree(iter([0]))) == [(0, 2)]
(8)                 assert G.out_degree(0, weight="weight") == 2
(4)             def test_size(self):
(8)                 G = self.K3
(8)                 assert G.size() == 6
(8)                 assert G.number_of_edges() == 6
(8)                 G.add_edge(0, 1, weight=0.3, other=1.2)
(8)                 assert round(G.size(weight="weight"), 2) == 6.3
(8)                 assert round(G.size(weight="other"), 2) == 7.2
(4)             def test_to_undirected_reciprocal(self):
(8)                 G = self.Graph()
(8)                 G.add_edge(1, 2)
(8)                 assert G.to_undirected().has_edge(1, 2)
(8)                 assert not G.to_undirected(reciprocal=True).has_edge(1, 2)
(8)                 G.add_edge(2, 1)
(8)                 assert G.to_undirected(reciprocal=True).has_edge(1, 2)
(4)             def test_reverse_copy(self):
(8)                 G = nx.MultiDiGraph([(0, 1), (0, 1)])
(8)                 R = G.reverse()
(8)                 assert sorted(R.edges()) == [(1, 0), (1, 0)]
(8)                 R.remove_edge(1, 0)
(8)                 assert sorted(R.edges()) == [(1, 0)]
(8)                 assert sorted(G.edges()) == [(0, 1), (0, 1)]
(4)             def test_reverse_nocopy(self):
(8)                 G = nx.MultiDiGraph([(0, 1), (0, 1)])
(8)                 R = G.reverse(copy=False)
(8)                 assert sorted(R.edges()) == [(1, 0), (1, 0)]
(8)                 pytest.raises(nx.NetworkXError, R.remove_edge, 1, 0)
(4)             def test_di_attributes_cached(self):
(8)                 G = self.K3.copy()
(8)                 assert id(G.in_edges) == id(G.in_edges)
(8)                 assert id(G.out_edges) == id(G.out_edges)
```

```
209: (8)                    assert id(G.in_degree) == id(G.in_degree)
210: (8)                    assert id(G.out_degree) == id(G.out_degree)
211: (8)                    assert id(G.succ) == id(G.succ)
212: (8)                    assert id(G.pred) == id(G.pred)
213: (0)            class TestMultiDiGraph(BaseMultiDiGraphTester, _TestMultiGraph):
214: (4)                def setup_method(self):
215: (8)                    self.Graph = nx.MultiDiGraph
216: (8)                    self.k3edges = [(0, 1), (0, 2), (1, 2)]
217: (8)                    self.k3nodes = [0, 1, 2]
218: (8)                    self.K3 = self.Graph()
219: (8)                    self.K3._succ = {0: {}, 1: {}, 2: {}}
220: (8)                    self.K3._pred = {0: {}, 1: {}, 2: {}}
221: (8)                    for u in self.k3nodes:
222: (12)                       for v in self.k3nodes:
223: (16)                           if u == v:
224: (20)                               continue
225: (16)                           d = {0: {}}
226: (16)                           self.K3._succ[u][v] = d
227: (16)                           self.K3._pred[v][u] = d
228: (8)                    self.K3._node = {}
229: (8)                    self.K3._node[0] = {}
230: (8)                    self.K3._node[1] = {}
231: (8)                    self.K3._node[2] = {}
232: (4)                def test_add_edge(self):
233: (8)                    G = self.Graph()
234: (8)                    G.add_edge(0, 1)
235: (8)                    assert G._adj == {0: {1: {0: {}}}, 1: {}}
236: (8)                    assert G._succ == {0: {1: {0: {}}}, 1: {}}
237: (8)                    assert G._pred == {0: {}, 1: {0: {0: {}}}}
238: (8)                    G = self.Graph()
239: (8)                    G.add_edge(*(0, 1))
240: (8)                    assert G._adj == {0: {1: {0: {}}}, 1: {}}
241: (8)                    assert G._succ == {0: {1: {0: {}}}, 1: {}}
242: (8)                    assert G._pred == {0: {}, 1: {0: {0: {}}}}
243: (8)                    with pytest.raises(ValueError, match="None cannot be a node"):
244: (12)                       G.add_edge(None, 3)
245: (4)                def test_add_edges_from(self):
246: (8)                    G = self.Graph()
247: (8)                    G.add_edges_from([(0, 1), (0, 1, {"weight": 3})])
248: (8)                    assert G._adj == {0: {1: {0: {}, 1: {"weight": 3}}}, 1: {}}
249: (8)                    assert G._succ == {0: {1: {0: {}, 1: {"weight": 3}}}, 1: {}}
250: (8)                    assert G._pred == {0: {}, 1: {0: {0: {}, 1: {"weight": 3}}}}
251: (8)                    G.add_edges_from([(0, 1), (0, 1, {"weight": 3})], weight=2)
252: (8)                    assert G._succ == {
253: (12)                       0: {1: {0: {}, 1: {"weight": 3}, 2: {"weight": 2}, 3: {"weight":
3}}},
254: (12)                       1: {},
255: (8)                    }
256: (8)                    assert G._pred == {
257: (12)                       0: {},
258: (12)                       1: {0: {0: {}, 1: {"weight": 3}, 2: {"weight": 2}, 3: {"weight":
3}}},
259: (8)                    }
260: (8)                    G = self.Graph()
261: (8)                    edges = [
262: (12)                       (0, 1, {"weight": 3}),
263: (12)                       (0, 1, (("weight", 2),)),
264: (12)                       (0, 1, 5),
265: (12)                       (0, 1, "s"),
266: (8)                    ]
267: (8)                    G.add_edges_from(edges)
268: (8)                    keydict = {0: {"weight": 3}, 1: {"weight": 2}, 5: {}, "s": {}}
269: (8)                    assert G._succ == {0: {1: keydict}, 1: {}}
270: (8)                    assert G._pred == {1: {0: keydict}, 0: {}}
271: (8)                    pytest.raises(nx.NetworkXError, G.add_edges_from, [(0,)])
272: (8)                    pytest.raises(nx.NetworkXError, G.add_edges_from, [(0, 1, 2, 3, 4)])
273: (8)                    pytest.raises(TypeError, G.add_edges_from, [0])
274: (8)                    with pytest.raises(ValueError, match="None cannot be a node"):
275: (12)                       G.add_edges_from([(None, 3), (3, 2)])
```

```
    def test_remove_edge(self):
        G = self.K3
        G.remove_edge(0, 1)
        assert G._succ == {
            0: {2: {0: {}}},
            1: {0: {0: {}}, 2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        assert G._pred == {
            0: {1: {0: {}}, 2: {0: {}}},
            1: {2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        pytest.raises((KeyError, nx.NetworkXError), G.remove_edge, -1, 0)
        pytest.raises((KeyError, nx.NetworkXError), G.remove_edge, 0, 2, key=1)
    def test_remove_multiedge(self):
        G = self.K3
        G.add_edge(0, 1, key="parallel edge")
        G.remove_edge(0, 1, key="parallel edge")
        assert G._adj == {
            0: {1: {0: {}}, 2: {0: {}}},
            1: {0: {0: {}}, 2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        assert G._succ == {
            0: {1: {0: {}}, 2: {0: {}}},
            1: {0: {0: {}}, 2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        assert G._pred == {
            0: {1: {0: {}}, 2: {0: {}}},
            1: {0: {0: {}}, 2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        G.remove_edge(0, 1)
        assert G._succ == {
            0: {2: {0: {}}},
            1: {0: {0: {}}, 2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        assert G._pred == {
            0: {1: {0: {}}, 2: {0: {}}},
            1: {2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        pytest.raises((KeyError, nx.NetworkXError), G.remove_edge, -1, 0)
    def test_remove_edges_from(self):
        G = self.K3
        G.remove_edges_from([(0, 1)])
        assert G._succ == {
            0: {2: {0: {}}},
            1: {0: {0: {}}, 2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        assert G._pred == {
            0: {1: {0: {}}, 2: {0: {}}},
            1: {2: {0: {}}},
            2: {0: {0: {}}, 1: {0: {}}},
        }
        G.remove_edges_from([(0, 0)])  # silent fail
class TestEdgeSubgraph(_TestMultiGraphEdgeSubgraph):
    """Unit tests for the :meth:`MultiDiGraph.edge_subgraph` method."""
    def setup_method(self):
        G = nx.MultiDiGraph()
        nx.add_path(G, range(5))
        nx.add_path(G, range(5))
        nx.add_path(G, reversed(range(5)))
        nx.add_path(G, reversed(range(5)))
```

```
344: (8)                    for i in range(5):
345: (12)                       G.nodes[i]["name"] = f"node{i}"
346: (8)                    G.adj[0][1][0]["name"] = "edge010"
347: (8)                    G.adj[0][1][1]["name"] = "edge011"
348: (8)                    G.adj[3][4][0]["name"] = "edge340"
349: (8)                    G.adj[3][4][1]["name"] = "edge341"
350: (8)                    G.graph["name"] = "graph"
351: (8)                    self.G = G
352: (8)                    self.H = G.edge_subgraph([(0, 1, 0), (3, 4, 1)])
353: (0)            class CustomDictClass(UserDict):
354: (4)                pass
355: (0)            class MultiDiGraphSubClass(nx.MultiDiGraph):
356: (4)                node_dict_factory = CustomDictClass  # type: ignore[assignment]
357: (4)                node_attr_dict_factory = CustomDictClass  # type: ignore[assignment]
358: (4)                adjlist_outer_dict_factory = CustomDictClass  # type: ignore[assignment]
359: (4)                adjlist_inner_dict_factory = CustomDictClass  # type: ignore[assignment]
360: (4)                edge_key_dict_factory = CustomDictClass  # type: ignore[assignment]
361: (4)                edge_attr_dict_factory = CustomDictClass  # type: ignore[assignment]
362: (4)                graph_attr_dict_factory = CustomDictClass  # type: ignore[assignment]
363: (0)            class TestMultiDiGraphSubclass(TestMultiDiGraph):
364: (4)                def setup_method(self):
365: (8)                    self.Graph = MultiDiGraphSubClass
366: (8)                    self.k3edges = [(0, 1), (0, 2), (1, 2)]
367: (8)                    self.k3nodes = [0, 1, 2]
368: (8)                    self.K3 = self.Graph()
369: (8)                    self.K3._succ = self.K3.adjlist_outer_dict_factory(
370: (12)                       {
371: (16)                           0: self.K3.adjlist_inner_dict_factory(),
372: (16)                           1: self.K3.adjlist_inner_dict_factory(),
373: (16)                           2: self.K3.adjlist_inner_dict_factory(),
374: (12)                       }
375: (8)                    )
376: (8)                    self.K3._pred = {0: {}, 1: {}, 2: {}}
377: (8)                    for u in self.k3nodes:
378: (12)                       for v in self.k3nodes:
379: (16)                           if u == v:
380: (20)                               continue
381: (16)                           d = {0: {}}
382: (16)                           self.K3._succ[u][v] = d
383: (16)                           self.K3._pred[v][u] = d
384: (8)                    self.K3._node = self.K3.node_dict_factory()
385: (8)                    self.K3._node[0] = self.K3.node_attr_dict_factory()
386: (8)                    self.K3._node[1] = self.K3.node_attr_dict_factory()
387: (8)                    self.K3._node[2] = self.K3.node_attr_dict_factory()
```

File 410 - test_multigraph.py:

```
1: (0)              from collections import UserDict
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import edges_equal
5: (0)              from .test_graph import BaseAttrGraphTester
6: (0)              from .test_graph import TestGraph as _TestGraph
7: (0)              class BaseMultiGraphTester(BaseAttrGraphTester):
8: (4)                  def test_has_edge(self):
9: (8)                      G = self.K3
10: (8)                     assert G.has_edge(0, 1)
11: (8)                     assert not G.has_edge(0, -1)
12: (8)                     assert G.has_edge(0, 1, 0)
13: (8)                     assert not G.has_edge(0, 1, 1)
14: (4)                 def test_get_edge_data(self):
15: (8)                     G = self.K3
16: (8)                     assert G.get_edge_data(0, 1) == {0: {}}
17: (8)                     assert G[0][1] == {0: {}}
18: (8)                     assert G[0][1][0] == {}
19: (8)                     assert G.get_edge_data(10, 20) is None
20: (8)                     assert G.get_edge_data(0, 1, 0) == {}
```

```
(4)     def test_adjacency(self):
(8)         G = self.K3
(8)         assert dict(G.adjacency()) == {
(12)            0: {1: {0: {}}, 2: {0: {}}},
(12)            1: {0: {0: {}}, 2: {0: {}}},
(12)            2: {0: {0: {}}, 1: {0: {}}},
(8)         }
(4)     def deepcopy_edge_attr(self, H, G):
(8)         assert G[1][2][0]["foo"] == H[1][2][0]["foo"]
(8)         G[1][2][0]["foo"].append(1)
(8)         assert G[1][2][0]["foo"] != H[1][2][0]["foo"]
(4)     def shallow_copy_edge_attr(self, H, G):
(8)         assert G[1][2][0]["foo"] == H[1][2][0]["foo"]
(8)         G[1][2][0]["foo"].append(1)
(8)         assert G[1][2][0]["foo"] == H[1][2][0]["foo"]
(4)     def graphs_equal(self, H, G):
(8)         assert G._adj == H._adj
(8)         assert G._node == H._node
(8)         assert G.graph == H.graph
(8)         assert G.name == H.name
(8)         if not G.is_directed() and not H.is_directed():
(12)            assert H._adj[1][2][0] is H._adj[2][1][0]
(12)            assert G._adj[1][2][0] is G._adj[2][1][0]
(8)         else:  # at least one is directed
(12)            if not G.is_directed():
(16)                G._pred = G._adj
(16)                G._succ = G._adj
(12)            if not H.is_directed():
(16)                H._pred = H._adj
(16)                H._succ = H._adj
(12)            assert G._pred == H._pred
(12)            assert G._succ == H._succ
(12)            assert H._succ[1][2][0] is H._pred[2][1][0]
(12)            assert G._succ[1][2][0] is G._pred[2][1][0]
(4)     def same_attrdict(self, H, G):
(8)         old_foo = H[1][2][0]["foo"]
(8)         H.adj[1][2][0]["foo"] = "baz"
(8)         assert G._adj == H._adj
(8)         H.adj[1][2][0]["foo"] = old_foo
(8)         assert G._adj == H._adj
(8)         old_foo = H.nodes[0]["foo"]
(8)         H.nodes[0]["foo"] = "baz"
(8)         assert G._node == H._node
(8)         H.nodes[0]["foo"] = old_foo
(8)         assert G._node == H._node
(4)     def different_attrdict(self, H, G):
(8)         old_foo = H[1][2][0]["foo"]
(8)         H.adj[1][2][0]["foo"] = "baz"
(8)         assert G._adj != H._adj
(8)         H.adj[1][2][0]["foo"] = old_foo
(8)         assert G._adj == H._adj
(8)         old_foo = H.nodes[0]["foo"]
(8)         H.nodes[0]["foo"] = "baz"
(8)         assert G._node != H._node
(8)         H.nodes[0]["foo"] = old_foo
(8)         assert G._node == H._node
(4)     def test_to_undirected(self):
(8)         G = self.K3
(8)         self.add_attributes(G)
(8)         H = nx.MultiGraph(G)
(8)         self.is_shallow_copy(H, G)
(8)         H = G.to_undirected()
(8)         self.is_deepcopy(H, G)
(4)     def test_to_directed(self):
(8)         G = self.K3
(8)         self.add_attributes(G)
(8)         H = nx.MultiDiGraph(G)
(8)         self.is_shallow_copy(H, G)
(8)         H = G.to_directed()
```

```
 90: (8)                    self.is_deepcopy(H, G)
 91: (4)                def test_number_of_edges_selfloops(self):
 92: (8)                    G = self.K3
 93: (8)                    G.add_edge(0, 0)
 94: (8)                    G.add_edge(0, 0)
 95: (8)                    G.add_edge(0, 0, key="parallel edge")
 96: (8)                    G.remove_edge(0, 0, key="parallel edge")
 97: (8)                    assert G.number_of_edges(0, 0) == 2
 98: (8)                    G.remove_edge(0, 0)
 99: (8)                    assert G.number_of_edges(0, 0) == 1
 100: (4)                def test_edge_lookup(self):
 101: (8)                    G = self.Graph()
 102: (8)                    G.add_edge(1, 2, foo="bar")
 103: (8)                    G.add_edge(1, 2, "key", foo="biz")
 104: (8)                    assert edges_equal(G.edges[1, 2, 0], {"foo": "bar"})
 105: (8)                    assert edges_equal(G.edges[1, 2, "key"], {"foo": "biz"})
 106: (4)                def test_edge_attr(self):
 107: (8)                    G = self.Graph()
 108: (8)                    G.add_edge(1, 2, key="k1", foo="bar")
 109: (8)                    G.add_edge(1, 2, key="k2", foo="baz")
 110: (8)                    assert isinstance(G.get_edge_data(1, 2), G.edge_key_dict_factory)
 111: (8)                    assert all(
 112: (12)                       isinstance(d, G.edge_attr_dict_factory) for u, v, d in
G.edges(data=True)
 113: (8)                    )
 114: (8)                    assert edges_equal(
 115: (12)                       G.edges(keys=True, data=True),
 116: (12)                       [(1, 2, "k1", {"foo": "bar"}), (1, 2, "k2", {"foo": "baz"})],
 117: (8)                    )
 118: (8)                    assert edges_equal(
 119: (12)                       G.edges(keys=True, data="foo"), [(1, 2, "k1", "bar"), (1, 2, "k2",
"baz")]
 120: (8)                    )
 121: (4)                def test_edge_attr4(self):
 122: (8)                    G = self.Graph()
 123: (8)                    G.add_edge(1, 2, key=0, data=7, spam="bar", bar="foo")
 124: (8)                    assert edges_equal(
 125: (12)                       G.edges(data=True), [(1, 2, {"data": 7, "spam": "bar", "bar":
"foo"})]
 126: (8)                    )
 127: (8)                    G[1][2][0]["data"] = 10  # OK to set data like this
 128: (8)                    assert edges_equal(
 129: (12)                       G.edges(data=True), [(1, 2, {"data": 10, "spam": "bar", "bar":
"foo"})]
 130: (8)                    )
 131: (8)                    G.adj[1][2][0]["data"] = 20
 132: (8)                    assert edges_equal(
 133: (12)                       G.edges(data=True), [(1, 2, {"data": 20, "spam": "bar", "bar":
"foo"})]
 134: (8)                    )
 135: (8)                    G.edges[1, 2, 0]["data"] = 21  # another spelling, "edge"
 136: (8)                    assert edges_equal(
 137: (12)                       G.edges(data=True), [(1, 2, {"data": 21, "spam": "bar", "bar":
"foo"})]
 138: (8)                    )
 139: (8)                    G.adj[1][2][0]["listdata"] = [20, 200]
 140: (8)                    G.adj[1][2][0]["weight"] = 20
 141: (8)                    assert edges_equal(
 142: (12)                       G.edges(data=True),
 143: (12)                       [
 144: (16)                           (
 145: (20)                               1,
 146: (20)                               2,
 147: (20)                               {
 148: (24)                                   "data": 21,
 149: (24)                                   "spam": "bar",
 150: (24)                                   "bar": "foo",
 151: (24)                                   "listdata": [20, 200],
 152: (24)                                   "weight": 20,
```

```
(20)                                },
(16)                            )
(12)                        ],
(8)                     )
(0)             class TestMultiGraph(BaseMultiGraphTester, _TestGraph):
(4)                 def setup_method(self):
(8)                     self.Graph = nx.MultiGraph
(8)                     ed1, ed2, ed3 = ({0: {}}, {0: {}}, {0: {}})
(8)                     self.k3adj = {0: {1: ed1, 2: ed2}, 1: {0: ed1, 2: ed3}, 2: {0: ed2, 1:
ed3}}
(8)                     self.k3edges = [(0, 1), (0, 2), (1, 2)]
(8)                     self.k3nodes = [0, 1, 2]
(8)                     self.K3 = self.Graph()
(8)                     self.K3._adj = self.k3adj
(8)                     self.K3._node = {}
(8)                     self.K3._node[0] = {}
(8)                     self.K3._node[1] = {}
(8)                     self.K3._node[2] = {}
(4)                 def test_data_input(self):
(8)                     G = self.Graph({1: [2], 2: [1]}, name="test")
(8)                     assert G.name == "test"
(8)                     expected = [(1, {2: {0: {}}}), (2, {1: {0: {}}})]
(8)                     assert sorted(G.adj.items()) == expected
(4)                 def test_data_multigraph_input(self):
(8)                     edata0 = {"w": 200, "s": "foo"}
(8)                     edata1 = {"w": 201, "s": "bar"}
(8)                     keydict = {0: edata0, 1: edata1}
(8)                     dododod = {"a": {"b": keydict}}
(8)                     multiple_edge = [("a", "b", 0, edata0), ("a", "b", 1, edata1)]
(8)                     single_edge = [("a", "b", 0, keydict)]
(8)                     G = self.Graph(dododod, multigraph_input=True)
(8)                     assert list(G.edges(keys=True, data=True)) == multiple_edge
(8)                     G = self.Graph(dododod, multigraph_input=None)
(8)                     assert list(G.edges(keys=True, data=True)) == multiple_edge
(8)                     G = self.Graph(dododod, multigraph_input=False)
(8)                     assert list(G.edges(keys=True, data=True)) == single_edge
(8)                     G = self.Graph(dododod, multigraph_input=True)
(8)                     H = self.Graph(nx.to_dict_of_dicts(G))
(8)                     assert nx.is_isomorphic(G, H) is True  # test that default is True
(8)                     for mgi in [True, False]:
(12)                        H = self.Graph(nx.to_dict_of_dicts(G), multigraph_input=mgi)
(12)                        assert nx.is_isomorphic(G, H) == mgi
(4)                 etraits = {"w": 200, "s": "foo"}
(4)                 egraphics = {"color": "blue", "shape": "box"}
(4)                 edata = {"traits": etraits, "graphics": egraphics}
(4)                 dododd1 = {"a": {"b": edata}}
(4)                 dododd2 = {"a": {"b": etraits}}
(4)                 dododd3 = {"a": {"b": {"traits": etraits, "s": "foo"}}}
(4)                 dol = {"a": ["b"]}
(4)                 multiple_edge = [("a", "b", "traits", etraits), ("a", "b", "graphics",
egraphics)]
(4)                 single_edge = [("a", "b", 0, {})]  # type: ignore[var-annotated]
(4)                 single_edge1 = [("a", "b", 0, edata)]
(4)                 single_edge2 = [("a", "b", 0, etraits)]
(4)                 single_edge3 = [("a", "b", 0, {"traits": etraits, "s": "foo"})]
(4)                 cases = [  # (dod, mgi, edges)
(8)                     (dododd1, True, multiple_edge),
(8)                     (dododd1, False, single_edge1),
(8)                     (dododd2, False, single_edge2),
(8)                     (dododd3, False, single_edge3),
(8)                     (dol, False, single_edge),
(4)                 ]
(4)                 @pytest.mark.parametrize("dod, mgi, edges", cases)
(4)                 def test_non_multigraph_input(self, dod, mgi, edges):
(8)                     G = self.Graph(dod, multigraph_input=mgi)
(8)                     assert list(G.edges(keys=True, data=True)) == edges
(8)                     G = nx.to_networkx_graph(dod, create_using=self.Graph,
multigraph_input=mgi)
(8)                     assert list(G.edges(keys=True, data=True)) == edges
```

```
    mgi_none_cases = [
        (dodod1, multiple_edge),
        (dodod2, single_edge2),
        (dodod3, single_edge3),
    ]
    @pytest.mark.parametrize("dod, edges", mgi_none_cases)
    def test_non_multigraph_input_mgi_none(self, dod, edges):
        G = self.Graph(dod)
        assert list(G.edges(keys=True, data=True)) == edges
    raise_cases = [dodod2, dodod3, dol]
    @pytest.mark.parametrize("dod", raise_cases)
    def test_non_multigraph_input_raise(self, dod):
        pytest.raises(nx.NetworkXError, self.Graph, dod, multigraph_input=True)
        pytest.raises(
            nx.NetworkXError,
            nx.to_networkx_graph,
            dod,
            create_using=self.Graph,
            multigraph_input=True,
        )
    def test_getitem(self):
        G = self.K3
        assert G[0] == {1: {0: {}}, 2: {0: {}}}
        with pytest.raises(KeyError):
            G.__getitem__("j")
        with pytest.raises(TypeError):
            G.__getitem__(["A"])
    def test_remove_node(self):
        G = self.K3
        G.remove_node(0)
        assert G.adj == {1: {2: {0: {}}}, 2: {1: {0: {}}}}
        with pytest.raises(nx.NetworkXError):
            G.remove_node(-1)
    def test_add_edge(self):
        G = self.Graph()
        G.add_edge(0, 1)
        assert G.adj == {0: {1: {0: {}}}, 1: {0: {0: {}}}}
        G = self.Graph()
        G.add_edge(*(0, 1))
        assert G.adj == {0: {1: {0: {}}}, 1: {0: {0: {}}}}
        G = self.Graph()
        with pytest.raises(ValueError):
            G.add_edge(None, "anything")
    def test_add_edge_conflicting_key(self):
        G = self.Graph()
        G.add_edge(0, 1, key=1)
        G.add_edge(0, 1)
        assert G.number_of_edges() == 2
        G = self.Graph()
        G.add_edges_from([(0, 1, 1, {})])
        G.add_edges_from([(0, 1)])
        assert G.number_of_edges() == 2
    def test_add_edges_from(self):
        G = self.Graph()
        G.add_edges_from([(0, 1), (0, 1, {"weight": 3})])
        assert G.adj == {
            0: {1: {0: {}, 1: {"weight": 3}}},
            1: {0: {0: {}, 1: {"weight": 3}}},
        }
        G.add_edges_from([(0, 1), (0, 1, {"weight": 3})], weight=2)
        assert G.adj == {
            0: {1: {0: {}, 1: {"weight": 3}, 2: {"weight": 2}, 3: {"weight": 3}}},
            1: {0: {0: {}, 1: {"weight": 3}, 2: {"weight": 2}, 3: {"weight": 3}}},
        }
        G = self.Graph()
        edges = [
```

```
(12)                        (0, 1, {"weight": 3}),
(12)                        (0, 1, (("weight", 2),)),
(12)                        (0, 1, 5),
(12)                        (0, 1, "s"),
(8)                     ]
(8)                     G.add_edges_from(edges)
(8)                     keydict = {0: {"weight": 3}, 1: {"weight": 2}, 5: {}, "s": {}}
(8)                     assert G._adj == {0: {1: keydict}, 1: {0: keydict}}
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.add_edges_from([(0,)])
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.add_edges_from([(0, 1, 2, 3, 4)])
(8)                     with pytest.raises(TypeError):
(12)                        G.add_edges_from([0])
(4)                 def test_multigraph_add_edges_from_four_tuple_misordered(self):
(8)                     """add_edges_from expects 4-tuples of the format (u, v, key,
data_dict).
(8)                     Ensure 4-tuples of form (u, v, data_dict, key) raise exception.
(8)                     """
(8)                     G = nx.MultiGraph()
(8)                     with pytest.raises(TypeError):
(12)                        G.add_edges_from([(0, 1, {"color": "red"}, 0)])
(4)                 def test_remove_edge(self):
(8)                     G = self.K3
(8)                     G.remove_edge(0, 1)
(8)                     assert G.adj == {0: {2: {0: {}}}, 1: {2: {0: {}}}, 2: {0: {0: {}}, 1:
{0: {}}}}
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.remove_edge(-1, 0)
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.remove_edge(0, 2, key=1)
(4)                 def test_remove_edges_from(self):
(8)                     G = self.K3.copy()
(8)                     G.remove_edges_from([(0, 1)])
(8)                     kd = {0: {}}
(8)                     assert G.adj == {0: {2: kd}, 1: {2: kd}, 2: {0: kd, 1: kd}}
(8)                     G.remove_edges_from([(0, 0)])  # silent fail
(8)                     self.K3.add_edge(0, 1)
(8)                     G = self.K3.copy()
(8)                     G.remove_edges_from(list(G.edges(data=True, keys=True)))
(8)                     assert G.adj == {0: {}, 1: {}, 2: {}}
(8)                     G = self.K3.copy()
(8)                     G.remove_edges_from(list(G.edges(data=False, keys=True)))
(8)                     assert G.adj == {0: {}, 1: {}, 2: {}}
(8)                     G = self.K3.copy()
(8)                     G.remove_edges_from(list(G.edges(data=False, keys=False)))
(8)                     assert G.adj == {0: {}, 1: {}, 2: {}}
(8)                     G = self.K3.copy()
(8)                     G.remove_edges_from([(0, 1, 0), (0, 2, 0, {}), (1, 2)])
(8)                     assert G.adj == {0: {1: {1: {}}}, 1: {0: {1: {}}}, 2: {}}
(4)                 def test_remove_multiedge(self):
(8)                     G = self.K3
(8)                     G.add_edge(0, 1, key="parallel edge")
(8)                     G.remove_edge(0, 1, key="parallel edge")
(8)                     assert G.adj == {
(12)                        0: {1: {0: {}}, 2: {0: {}}},
(12)                        1: {0: {0: {}}, 2: {0: {}}},
(12)                        2: {0: {0: {}}, 1: {0: {}}},
(8)                     }
(8)                     G.remove_edge(0, 1)
(8)                     kd = {0: {}}
(8)                     assert G.adj == {0: {2: kd}, 1: {2: kd}, 2: {0: kd, 1: kd}}
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.remove_edge(-1, 0)
(0)             class TestEdgeSubgraph:
(4)                 """Unit tests for the :meth:`MultiGraph.edge_subgraph` method."""
(4)                 def setup_method(self):
(8)                     G = nx.MultiGraph()
(8)                     nx.add_path(G, range(5))
```

```
(8)                 nx.add_path(G, range(5))
(8)                 for i in range(5):
(12)                    G.nodes[i]["name"] = f"node{i}"
(8)                 G.adj[0][1][0]["name"] = "edge010"
(8)                 G.adj[0][1][1]["name"] = "edge011"
(8)                 G.adj[3][4][0]["name"] = "edge340"
(8)                 G.adj[3][4][1]["name"] = "edge341"
(8)                 G.graph["name"] = "graph"
(8)                 self.G = G
(8)                 self.H = G.edge_subgraph([(0, 1, 0), (3, 4, 1)])
(4)             def test_correct_nodes(self):
(8)                 """Tests that the subgraph has the correct nodes."""
(8)                 assert [0, 1, 3, 4] == sorted(self.H.nodes())
(4)             def test_correct_edges(self):
(8)                 """Tests that the subgraph has the correct edges."""
(8)                 assert [(0, 1, 0, "edge010"), (3, 4, 1, "edge341")] == sorted(
(12)                    self.H.edges(keys=True, data="name")
(8)                 )
(4)             def test_add_node(self):
(8)                 """Tests that adding a node to the original graph does not
(8)                 affect the nodes of the subgraph.
(8)                 """
(8)                 self.G.add_node(5)
(8)                 assert [0, 1, 3, 4] == sorted(self.H.nodes())
(4)             def test_remove_node(self):
(8)                 """Tests that removing a node in the original graph does
(8)                 affect the nodes of the subgraph.
(8)                 """
(8)                 self.G.remove_node(0)
(8)                 assert [1, 3, 4] == sorted(self.H.nodes())
(4)             def test_node_attr_dict(self):
(8)                 """Tests that the node attribute dictionary of the two graphs is
(8)                 the same object.
(8)                 """
(8)                 for v in self.H:
(12)                    assert self.G.nodes[v] == self.H.nodes[v]
(8)                 self.G.nodes[0]["name"] = "foo"
(8)                 assert self.G.nodes[0] == self.H.nodes[0]
(8)                 self.H.nodes[1]["name"] = "bar"
(8)                 assert self.G.nodes[1] == self.H.nodes[1]
(4)             def test_edge_attr_dict(self):
(8)                 """Tests that the edge attribute dictionary of the two graphs is
(8)                 the same object.
(8)                 """
(8)                 for u, v, k in self.H.edges(keys=True):
(12)                    assert self.G._adj[u][v][k] == self.H._adj[u][v][k]
(8)                 self.G._adj[0][1][0]["name"] = "foo"
(8)                 assert self.G._adj[0][1][0]["name"] == self.H._adj[0][1][0]["name"]
(8)                 self.H._adj[3][4][1]["name"] = "bar"
(8)                 assert self.G._adj[3][4][1]["name"] == self.H._adj[3][4][1]["name"]
(4)             def test_graph_attr_dict(self):
(8)                 """Tests that the graph attribute dictionary of the two graphs
(8)                 is the same object.
(8)                 """
(8)                 assert self.G.graph is self.H.graph
(0)         class CustomDictClass(UserDict):
(4)             pass
(0)         class MultiGraphSubClass(nx.MultiGraph):
(4)             node_dict_factory = CustomDictClass  # type: ignore[assignment]
(4)             node_attr_dict_factory = CustomDictClass  # type: ignore[assignment]
(4)             adjlist_outer_dict_factory = CustomDictClass  # type: ignore[assignment]
(4)             adjlist_inner_dict_factory = CustomDictClass  # type: ignore[assignment]
(4)             edge_key_dict_factory = CustomDictClass  # type: ignore[assignment]
(4)             edge_attr_dict_factory = CustomDictClass  # type: ignore[assignment]
(4)             graph_attr_dict_factory = CustomDictClass  # type: ignore[assignment]
(0)         class TestMultiGraphSubclass(TestMultiGraph):
(4)             def setup_method(self):
(8)                 self.Graph = MultiGraphSubClass
(8)                 self.k3edges = [(0, 1), (0, 2), (1, 2)]
```

```
421: (8)                    self.k3nodes = [0, 1, 2]
422: (8)                    self.K3 = self.Graph()
423: (8)                    self.K3._adj = self.K3.adjlist_outer_dict_factory(
424: (12)                       {
425: (16)                           0: self.K3.adjlist_inner_dict_factory(),
426: (16)                           1: self.K3.adjlist_inner_dict_factory(),
427: (16)                           2: self.K3.adjlist_inner_dict_factory(),
428: (12)                       }
429: (8)                    )
430: (8)                    self.K3._pred = {0: {}, 1: {}, 2: {}}
431: (8)                    for u in self.k3nodes:
432: (12)                       for v in self.k3nodes:
433: (16)                           if u != v:
434: (20)                               d = {0: {}}
435: (20)                               self.K3._adj[u][v] = d
436: (20)                               self.K3._adj[v][u] = d
437: (8)                    self.K3._node = self.K3.node_dict_factory()
438: (8)                    self.K3._node[0] = self.K3.node_attr_dict_factory()
439: (8)                    self.K3._node[1] = self.K3.node_attr_dict_factory()
440: (8)                    self.K3._node[2] = self.K3.node_attr_dict_factory()

----------------------------------------

File 411 - test_reportviews.py:

1: (0)              import pickle
2: (0)              from copy import deepcopy
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.classes import reportviews as rv
6: (0)              from networkx.classes.reportviews import NodeDataView
7: (0)              class TestNodeView:
8: (4)                  @classmethod
9: (4)                  def setup_class(cls):
10: (8)                     cls.G = nx.path_graph(9)
11: (8)                     cls.nv = cls.G.nodes  # NodeView(G)
12: (4)                 def test_pickle(self):
13: (8)                     import pickle
14: (8)                     nv = self.nv
15: (8)                     pnv = pickle.loads(pickle.dumps(nv, -1))
16: (8)                     assert nv == pnv
17: (8)                     assert nv.__slots__ == pnv.__slots__
18: (4)                 def test_str(self):
19: (8)                     assert str(self.nv) == "[0, 1, 2, 3, 4, 5, 6, 7, 8]"
20: (4)                 def test_repr(self):
21: (8)                     assert repr(self.nv) == "NodeView((0, 1, 2, 3, 4, 5, 6, 7, 8))"
22: (4)                 def test_contains(self):
23: (8)                     G = self.G.copy()
24: (8)                     nv = G.nodes
25: (8)                     assert 7 in nv
26: (8)                     assert 9 not in nv
27: (8)                     G.remove_node(7)
28: (8)                     G.add_node(9)
29: (8)                     assert 7 not in nv
30: (8)                     assert 9 in nv
31: (4)                 def test_getitem(self):
32: (8)                     G = self.G.copy()
33: (8)                     nv = G.nodes
34: (8)                     G.nodes[3]["foo"] = "bar"
35: (8)                     assert nv[7] == {}
36: (8)                     assert nv[3] == {"foo": "bar"}
37: (8)                     with pytest.raises(nx.NetworkXError):
38: (12)                        G.nodes[0:5]
39: (4)                 def test_iter(self):
40: (8)                     nv = self.nv
41: (8)                     for i, n in enumerate(nv):
42: (12)                        assert i == n
43: (8)                     inv = iter(nv)
44: (8)                     assert next(inv) == 0
```

```
45: (8)                     assert iter(nv) != nv
46: (8)                     assert iter(inv) == inv
47: (8)                     inv2 = iter(nv)
48: (8)                     next(inv2)
49: (8)                     assert list(inv) == list(inv2)
50: (8)                     nnv = nv(data=False)
51: (8)                     for i, n in enumerate(nnv):
52: (12)                        assert i == n
53: (4)                 def test_call(self):
54: (8)                     nodes = self.nv
55: (8)                     assert nodes is nodes()
56: (8)                     assert nodes is not nodes(data=True)
57: (8)                     assert nodes is not nodes(data="weight")
58: (0)             class TestNodeDataView:
59: (4)                 @classmethod
60: (4)                 def setup_class(cls):
61: (8)                     cls.G = nx.path_graph(9)
62: (8)                     cls.nv = NodeDataView(cls.G)
63: (8)                     cls.ndv = cls.G.nodes.data(True)
64: (8)                     cls.nwv = cls.G.nodes.data("foo")
65: (4)                 def test_viewtype(self):
66: (8)                     nv = self.G.nodes
67: (8)                     ndvfalse = nv.data(False)
68: (8)                     assert nv is ndvfalse
69: (8)                     assert nv is not self.ndv
70: (4)                 def test_pickle(self):
71: (8)                     import pickle
72: (8)                     nv = self.nv
73: (8)                     pnv = pickle.loads(pickle.dumps(nv, -1))
74: (8)                     assert nv == pnv
75: (8)                     assert nv.__slots__ == pnv.__slots__
76: (4)                 def test_str(self):
77: (8)                     msg = str([(n, {}) for n in range(9)])
78: (8)                     assert str(self.ndv) == msg
79: (4)                 def test_repr(self):
80: (8)                     expected = "NodeDataView((0, 1, 2, 3, 4, 5, 6, 7, 8))"
81: (8)                     assert repr(self.nv) == expected
82: (8)                     expected = (
83: (12)                        "NodeDataView({0: {}, 1: {}, 2: {}, 3: {}, "
84: (12)                        + "4: {}, 5: {}, 6: {}, 7: {}, 8: {}})"
85: (8)                     )
86: (8)                     assert repr(self.ndv) == expected
87: (8)                     expected = (
88: (12)                        "NodeDataView({0: None, 1: None, 2: None, 3: None, 4: None, "
89: (12)                        + "5: None, 6: None, 7: None, 8: None}, data='foo')"
90: (8)                     )
91: (8)                     assert repr(self.nwv) == expected
92: (4)                 def test_contains(self):
93: (8)                     G = self.G.copy()
94: (8)                     nv = G.nodes.data()
95: (8)                     nwv = G.nodes.data("foo")
96: (8)                     G.nodes[3]["foo"] = "bar"
97: (8)                     assert (7, {}) in nv
98: (8)                     assert (3, {"foo": "bar"}) in nv
99: (8)                     assert (3, "bar") in nwv
100: (8)                    assert (7, None) in nwv
101: (8)                    nwv_def = G.nodes(data="foo", default="biz")
102: (8)                    assert (7, "biz") in nwv_def
103: (8)                    assert (3, "bar") in nwv_def
104: (4)                def test_getitem(self):
105: (8)                    G = self.G.copy()
106: (8)                    nv = G.nodes
107: (8)                    G.nodes[3]["foo"] = "bar"
108: (8)                    assert nv[3] == {"foo": "bar"}
109: (8)                    nwv_def = G.nodes(data="foo", default="biz")
110: (8)                    assert nwv_def[7], "biz"
111: (8)                    assert nwv_def[3] == "bar"
112: (8)                    with pytest.raises(nx.NetworkXError):
113: (12)                       G.nodes.data()[0:5]
```

```
114: (4)                def test_iter(self):
115: (8)                    G = self.G.copy()
116: (8)                    nv = G.nodes.data()
117: (8)                    ndv = G.nodes.data(True)
118: (8)                    nwv = G.nodes.data("foo")
119: (8)                    for i, (n, d) in enumerate(nv):
120: (12)                       assert i == n
121: (12)                       assert d == {}
122: (8)                    inv = iter(nv)
123: (8)                    assert next(inv) == (0, {})
124: (8)                    G.nodes[3]["foo"] = "bar"
125: (8)                    for n, d in nv:
126: (12)                       if n == 3:
127: (16)                           assert d == {"foo": "bar"}
128: (12)                       else:
129: (16)                           assert d == {}
130: (8)                    for n, d in ndv:
131: (12)                       if n == 3:
132: (16)                           assert d == {"foo": "bar"}
133: (12)                       else:
134: (16)                           assert d == {}
135: (8)                    for n, d in nwv:
136: (12)                       if n == 3:
137: (16)                           assert d == "bar"
138: (12)                       else:
139: (16)                           assert d is None
140: (8)                    for n, d in G.nodes.data("foo", default=1):
141: (12)                       if n == 3:
142: (16)                           assert d == "bar"
143: (12)                       else:
144: (16)                           assert d == 1
145: (0)            def test_nodedataview_unhashable():
146: (4)                G = nx.path_graph(9)
147: (4)                G.nodes[3]["foo"] = "bar"
148: (4)                nvs = [G.nodes.data()]
149: (4)                nvs.append(G.nodes.data(True))
150: (4)                H = G.copy()
151: (4)                H.nodes[4]["foo"] = {1, 2, 3}
152: (4)                nvs.append(H.nodes.data(True))
153: (4)                for nv in nvs:
154: (8)                    pytest.raises(TypeError, set, nv)
155: (8)                    pytest.raises(TypeError, eval, "nv | nv", locals())
156: (4)                Gn = G.nodes.data(False)
157: (4)                set(Gn)
158: (4)                Gn | Gn
159: (4)                Gn = G.nodes.data("foo")
160: (4)                set(Gn)
161: (4)                Gn | Gn
162: (0)            class TestNodeViewSetOps:
163: (4)                @classmethod
164: (4)                def setup_class(cls):
165: (8)                    cls.G = nx.path_graph(9)
166: (8)                    cls.G.nodes[3]["foo"] = "bar"
167: (8)                    cls.nv = cls.G.nodes
168: (4)                def n_its(self, nodes):
169: (8)                    return set(nodes)
170: (4)                def test_len(self):
171: (8)                    G = self.G.copy()
172: (8)                    nv = G.nodes
173: (8)                    assert len(nv) == 9
174: (8)                    G.remove_node(7)
175: (8)                    assert len(nv) == 8
176: (8)                    G.add_node(9)
177: (8)                    assert len(nv) == 9
178: (4)                def test_and(self):
179: (8)                    nv = self.nv
180: (8)                    some_nodes = self.n_its(range(5, 12))
181: (8)                    assert nv & some_nodes == self.n_its(range(5, 9))
182: (8)                    assert some_nodes & nv == self.n_its(range(5, 9))
```

```
(4)     def test_or(self):
(8)         nv = self.nv
(8)         some_nodes = self.n_its(range(5, 12))
(8)         assert nv | some_nodes == self.n_its(range(12))
(8)         assert some_nodes | nv == self.n_its(range(12))
(4)     def test_xor(self):
(8)         nv = self.nv
(8)         some_nodes = self.n_its(range(5, 12))
(8)         nodes = {0, 1, 2, 3, 4, 9, 10, 11}
(8)         assert nv ^ some_nodes == self.n_its(nodes)
(8)         assert some_nodes ^ nv == self.n_its(nodes)
(4)     def test_sub(self):
(8)         nv = self.nv
(8)         some_nodes = self.n_its(range(5, 12))
(8)         assert nv - some_nodes == self.n_its(range(5))
(8)         assert some_nodes - nv == self.n_its(range(9, 12))
(0) class TestNodeDataViewSetOps(TestNodeViewSetOps):
(4)     @classmethod
(4)     def setup_class(cls):
(8)         cls.G = nx.path_graph(9)
(8)         cls.G.nodes[3]["foo"] = "bar"
(8)         cls.nv = cls.G.nodes.data("foo")
(4)     def n_its(self, nodes):
(8)         return {(node, "bar" if node == 3 else None) for node in nodes}
(0) class TestNodeDataViewDefaultSetOps(TestNodeDataViewSetOps):
(4)     @classmethod
(4)     def setup_class(cls):
(8)         cls.G = nx.path_graph(9)
(8)         cls.G.nodes[3]["foo"] = "bar"
(8)         cls.nv = cls.G.nodes.data("foo", default=1)
(4)     def n_its(self, nodes):
(8)         return {(node, "bar" if node == 3 else 1) for node in nodes}
(0) class TestEdgeDataView:
(4)     @classmethod
(4)     def setup_class(cls):
(8)         cls.G = nx.path_graph(9)
(8)         cls.eview = nx.reportviews.EdgeView
(4)     def test_pickle(self):
(8)         import pickle
(8)         ev = self.eview(self.G)(data=True)
(8)         pev = pickle.loads(pickle.dumps(ev, -1))
(8)         assert list(ev) == list(pev)
(8)         assert ev.__slots__ == pev.__slots__
(4)     def modify_edge(self, G, e, **kwds):
(8)         G._adj[e[0]][e[1]].update(kwds)
(4)     def test_str(self):
(8)         ev = self.eview(self.G)(data=True)
(8)         rep = str([(n, n + 1, {}) for n in range(8)])
(8)         assert str(ev) == rep
(4)     def test_repr(self):
(8)         ev = self.eview(self.G)(data=True)
(8)         rep = (
(12)            "EdgeDataView([(0, 1, {}), (1, 2, {}), "
(12)            + "(2, 3, {}), (3, 4, {}), "
(12)            + "(4, 5, {}), (5, 6, {}), "
(12)            + "(6, 7, {}), (7, 8, {})])"
(8)         )
(8)         assert repr(ev) == rep
(4)     def test_iterdata(self):
(8)         G = self.G.copy()
(8)         evr = self.eview(G)
(8)         ev = evr(data=True)
(8)         ev_def = evr(data="foo", default=1)
(8)         for u, v, d in ev:
(12)            pass
(8)         assert d == {}
(8)         for u, v, wt in ev_def:
(12)            pass
(8)         assert wt == 1
```

```
252: (8)                    self.modify_edge(G, (2, 3), foo="bar")
253: (8)                    for e in ev:
254: (12)                       assert len(e) == 3
255: (12)                       if set(e[:2]) == {2, 3}:
256: (16)                           assert e[2] == {"foo": "bar"}
257: (16)                           checked = True
258: (12)                       else:
259: (16)                           assert e[2] == {}
260: (8)                    assert checked
261: (8)                    for e in ev_def:
262: (12)                       assert len(e) == 3
263: (12)                       if set(e[:2]) == {2, 3}:
264: (16)                           assert e[2] == "bar"
265: (16)                           checked_wt = True
266: (12)                       else:
267: (16)                           assert e[2] == 1
268: (8)                    assert checked_wt
269: (4)                def test_iter(self):
270: (8)                    evr = self.eview(self.G)
271: (8)                    ev = evr()
272: (8)                    for u, v in ev:
273: (12)                       pass
274: (8)                    iev = iter(ev)
275: (8)                    assert next(iev) == (0, 1)
276: (8)                    assert iter(ev) != ev
277: (8)                    assert iter(iev) == iev
278: (4)                def test_contains(self):
279: (8)                    evr = self.eview(self.G)
280: (8)                    ev = evr()
281: (8)                    if self.G.is_directed():
282: (12)                       assert (1, 2) in ev and (2, 1) not in ev
283: (8)                    else:
284: (12)                       assert (1, 2) in ev and (2, 1) in ev
285: (8)                    assert (1, 4) not in ev
286: (8)                    assert (1, 90) not in ev
287: (8)                    assert (90, 1) not in ev
288: (4)                def test_contains_with_nbunch(self):
289: (8)                    evr = self.eview(self.G)
290: (8)                    ev = evr(nbunch=[0, 2])
291: (8)                    if self.G.is_directed():
292: (12)                       assert (0, 1) in ev
293: (12)                       assert (1, 2) not in ev
294: (12)                       assert (2, 3) in ev
295: (8)                    else:
296: (12)                       assert (0, 1) in ev
297: (12)                       assert (1, 2) in ev
298: (12)                       assert (2, 3) in ev
299: (8)                    assert (3, 4) not in ev
300: (8)                    assert (4, 5) not in ev
301: (8)                    assert (5, 6) not in ev
302: (8)                    assert (7, 8) not in ev
303: (8)                    assert (8, 9) not in ev
304: (4)                def test_len(self):
305: (8)                    evr = self.eview(self.G)
306: (8)                    ev = evr(data="foo")
307: (8)                    assert len(ev) == 8
308: (8)                    assert len(evr(1)) == 2
309: (8)                    assert len(evr([1, 2, 3])) == 4
310: (8)                    assert len(self.G.edges(1)) == 2
311: (8)                    assert len(self.G.edges()) == 8
312: (8)                    assert len(self.G.edges) == 8
313: (8)                    H = self.G.copy()
314: (8)                    H.add_edge(1, 1)
315: (8)                    assert len(H.edges(1)) == 3
316: (8)                    assert len(H.edges()) == 9
317: (8)                    assert len(H.edges) == 9
318: (0)            class TestOutEdgeDataView(TestEdgeDataView):
319: (4)                @classmethod
320: (4)                def setup_class(cls):
```

```
(8)                 cls.G = nx.path_graph(9, create_using=nx.DiGraph())
(8)                 cls.eview = nx.reportviews.OutEdgeView
(4)             def test_repr(self):
(8)                 ev = self.eview(self.G)(data=True)
(8)                 rep = (
(12)                    "OutEdgeDataView([(0, 1, {}), (1, 2, {}), "
(12)                    + "(2, 3, {}), (3, 4, {}), "
(12)                    + "(4, 5, {}), (5, 6, {}), "
(12)                    + "(6, 7, {}), (7, 8, {})])"
(8)                 )
(8)                 assert repr(ev) == rep
(4)             def test_len(self):
(8)                 evr = self.eview(self.G)
(8)                 ev = evr(data="foo")
(8)                 assert len(ev) == 8
(8)                 assert len(evr(1)) == 1
(8)                 assert len(evr([1, 2, 3])) == 3
(8)                 assert len(self.G.edges(1)) == 1
(8)                 assert len(self.G.edges()) == 8
(8)                 assert len(self.G.edges) == 8
(8)                 H = self.G.copy()
(8)                 H.add_edge(1, 1)
(8)                 assert len(H.edges(1)) == 2
(8)                 assert len(H.edges()) == 9
(8)                 assert len(H.edges) == 9
(4)             def test_contains_with_nbunch(self):
(8)                 evr = self.eview(self.G)
(8)                 ev = evr(nbunch=[0, 2])
(8)                 assert (0, 1) in ev
(8)                 assert (1, 2) not in ev
(8)                 assert (2, 3) in ev
(8)                 assert (3, 4) not in ev
(8)                 assert (4, 5) not in ev
(8)                 assert (5, 6) not in ev
(8)                 assert (7, 8) not in ev
(8)                 assert (8, 9) not in ev
(0)         class TestInEdgeDataView(TestOutEdgeDataView):
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 cls.G = nx.path_graph(9, create_using=nx.DiGraph())
(8)                 cls.eview = nx.reportviews.InEdgeView
(4)             def test_repr(self):
(8)                 ev = self.eview(self.G)(data=True)
(8)                 rep = (
(12)                    "InEdgeDataView([(0, 1, {}), (1, 2, {}), "
(12)                    + "(2, 3, {}), (3, 4, {}), "
(12)                    + "(4, 5, {}), (5, 6, {}), "
(12)                    + "(6, 7, {}), (7, 8, {})])"
(8)                 )
(8)                 assert repr(ev) == rep
(4)             def test_contains_with_nbunch(self):
(8)                 evr = self.eview(self.G)
(8)                 ev = evr(nbunch=[0, 2])
(8)                 assert (0, 1) not in ev
(8)                 assert (1, 2) in ev
(8)                 assert (2, 3) not in ev
(8)                 assert (3, 4) not in ev
(8)                 assert (4, 5) not in ev
(8)                 assert (5, 6) not in ev
(8)                 assert (7, 8) not in ev
(8)                 assert (8, 9) not in ev
(0)         class TestMultiEdgeDataView(TestEdgeDataView):
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 cls.G = nx.path_graph(9, create_using=nx.MultiGraph())
(8)                 cls.eview = nx.reportviews.MultiEdgeView
(4)             def modify_edge(self, G, e, **kwds):
(8)                 G._adj[e[0]][e[1]][0].update(kwds)
(4)             def test_repr(self):
```

```
(8)                     ev = self.eview(self.G)(data=True)
(8)                     rep = (
(12)                        "MultiEdgeDataView([(0, 1, {}), (1, 2, {}), "
(12)                        + "(2, 3, {}), (3, 4, {}), "
(12)                        + "(4, 5, {}), (5, 6, {}), "
(12)                        + "(6, 7, {}), (7, 8, {})])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     evr = self.eview(self.G)
(8)                     ev = evr(nbunch=[0, 2])
(8)                     assert (0, 1) in ev
(8)                     assert (1, 2) in ev
(8)                     assert (2, 3) in ev
(8)                     assert (3, 4) not in ev
(8)                     assert (4, 5) not in ev
(8)                     assert (5, 6) not in ev
(8)                     assert (7, 8) not in ev
(8)                     assert (8, 9) not in ev
(0)             class TestOutMultiEdgeDataView(TestOutEdgeDataView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, create_using=nx.MultiDiGraph())
(8)                     cls.eview = nx.reportviews.OutMultiEdgeView
(4)                 def modify_edge(self, G, e, **kwds):
(8)                     G._adj[e[0]][e[1]][0].update(kwds)
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)(data=True)
(8)                     rep = (
(12)                        "OutMultiEdgeDataView([(0, 1, {}), (1, 2, {}), "
(12)                        + "(2, 3, {}), (3, 4, {}), "
(12)                        + "(4, 5, {}), (5, 6, {}), "
(12)                        + "(6, 7, {}), (7, 8, {})])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     evr = self.eview(self.G)
(8)                     ev = evr(nbunch=[0, 2])
(8)                     assert (0, 1) in ev
(8)                     assert (1, 2) not in ev
(8)                     assert (2, 3) in ev
(8)                     assert (3, 4) not in ev
(8)                     assert (4, 5) not in ev
(8)                     assert (5, 6) not in ev
(8)                     assert (7, 8) not in ev
(8)                     assert (8, 9) not in ev
(0)             class TestInMultiEdgeDataView(TestOutMultiEdgeDataView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, create_using=nx.MultiDiGraph())
(8)                     cls.eview = nx.reportviews.InMultiEdgeView
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)(data=True)
(8)                     rep = (
(12)                        "InMultiEdgeDataView([(0, 1, {}), (1, 2, {}), "
(12)                        + "(2, 3, {}), (3, 4, {}), "
(12)                        + "(4, 5, {}), (5, 6, {}), "
(12)                        + "(6, 7, {}), (7, 8, {})])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     evr = self.eview(self.G)
(8)                     ev = evr(nbunch=[0, 2])
(8)                     assert (0, 1) not in ev
(8)                     assert (1, 2) in ev
(8)                     assert (2, 3) not in ev
(8)                     assert (3, 4) not in ev
(8)                     assert (4, 5) not in ev
(8)                     assert (5, 6) not in ev
```

```
459: (8)                    assert (7, 8) not in ev
460: (8)                    assert (8, 9) not in ev
461: (0)            class TestEdgeView:
462: (4)                @classmethod
463: (4)                def setup_class(cls):
464: (8)                    cls.G = nx.path_graph(9)
465: (8)                    cls.eview = nx.reportviews.EdgeView
466: (4)                def test_pickle(self):
467: (8)                    import pickle
468: (8)                    ev = self.eview(self.G)
469: (8)                    pev = pickle.loads(pickle.dumps(ev, -1))
470: (8)                    assert ev == pev
471: (8)                    assert ev.__slots__ == pev.__slots__
472: (4)                def modify_edge(self, G, e, **kwds):
473: (8)                    G._adj[e[0]][e[1]].update(kwds)
474: (4)                def test_str(self):
475: (8)                    ev = self.eview(self.G)
476: (8)                    rep = str([(n, n + 1) for n in range(8)])
477: (8)                    assert str(ev) == rep
478: (4)                def test_repr(self):
479: (8)                    ev = self.eview(self.G)
480: (8)                    rep = (
481: (12)                       "EdgeView([(0, 1), (1, 2), (2, 3), (3, 4), "
482: (12)                       + "(4, 5), (5, 6), (6, 7), (7, 8)])"
483: (8)                    )
484: (8)                    assert repr(ev) == rep
485: (4)                def test_getitem(self):
486: (8)                    G = self.G.copy()
487: (8)                    ev = G.edges
488: (8)                    G.edges[0, 1]["foo"] = "bar"
489: (8)                    assert ev[0, 1] == {"foo": "bar"}
490: (8)                    with pytest.raises(nx.NetworkXError, match=".*does not support
slicing"):
491: (12)                       G.edges[0:5]
492: (8)                    with pytest.raises(KeyError, match=r".*edge.*is not in the graph."):
493: (12)                       G.edges[0, 9]
494: (4)                def test_call(self):
495: (8)                    ev = self.eview(self.G)
496: (8)                    assert id(ev) == id(ev())
497: (8)                    assert id(ev) == id(ev(data=False))
498: (8)                    assert id(ev) != id(ev(data=True))
499: (8)                    assert id(ev) != id(ev(nbunch=1))
500: (4)                def test_data(self):
501: (8)                    ev = self.eview(self.G)
502: (8)                    assert id(ev) != id(ev.data())
503: (8)                    assert id(ev) == id(ev.data(data=False))
504: (8)                    assert id(ev) != id(ev.data(data=True))
505: (8)                    assert id(ev) != id(ev.data(nbunch=1))
506: (4)                def test_iter(self):
507: (8)                    ev = self.eview(self.G)
508: (8)                    for u, v in ev:
509: (12)                       pass
510: (8)                    iev = iter(ev)
511: (8)                    assert next(iev) == (0, 1)
512: (8)                    assert iter(ev) != ev
513: (8)                    assert iter(iev) == iev
514: (4)                def test_contains(self):
515: (8)                    ev = self.eview(self.G)
516: (8)                    edv = ev()
517: (8)                    if self.G.is_directed():
518: (12)                       assert (1, 2) in ev and (2, 1) not in ev
519: (12)                       assert (1, 2) in edv and (2, 1) not in edv
520: (8)                    else:
521: (12)                       assert (1, 2) in ev and (2, 1) in ev
522: (12)                       assert (1, 2) in edv and (2, 1) in edv
523: (8)                    assert (1, 4) not in ev
524: (8)                    assert (1, 4) not in edv
525: (8)                    assert (1, 90) not in ev
526: (8)                    assert (90, 1) not in ev
```

```
(8)                     assert (1, 90) not in edv
(8)                     assert (90, 1) not in edv
(4)                 def test_contains_with_nbunch(self):
(8)                     ev = self.eview(self.G)
(8)                     evn = ev(nbunch=[0, 2])
(8)                     assert (0, 1) in evn
(8)                     assert (1, 2) in evn
(8)                     assert (2, 3) in evn
(8)                     assert (3, 4) not in evn
(8)                     assert (4, 5) not in evn
(8)                     assert (5, 6) not in evn
(8)                     assert (7, 8) not in evn
(8)                     assert (8, 9) not in evn
(4)                 def test_len(self):
(8)                     ev = self.eview(self.G)
(8)                     num_ed = 9 if self.G.is_multigraph() else 8
(8)                     assert len(ev) == num_ed
(8)                     H = self.G.copy()
(8)                     H.add_edge(1, 1)
(8)                     assert len(H.edges(1)) == 3 + H.is_multigraph() - H.is_directed()
(8)                     assert len(H.edges()) == num_ed + 1
(8)                     assert len(H.edges) == num_ed + 1
(4)                 def test_and(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1), (1, 0), (0, 2)}
(8)                     if self.G.is_directed():
(12)                        assert some_edges & ev, {(0, 1)}
(12)                        assert ev & some_edges, {(0, 1)}
(8)                     else:
(12)                        assert ev & some_edges == {(0, 1), (1, 0)}
(12)                        assert some_edges & ev == {(0, 1), (1, 0)}
(8)                     return
(4)                 def test_or(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1), (1, 0), (0, 2)}
(8)                     result1 = {(n, n + 1) for n in range(8)}
(8)                     result1.update(some_edges)
(8)                     result2 = {(n + 1, n) for n in range(8)}
(8)                     result2.update(some_edges)
(8)                     assert (ev | some_edges) in (result1, result2)
(8)                     assert (some_edges | ev) in (result1, result2)
(4)                 def test_xor(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1), (1, 0), (0, 2)}
(8)                     if self.G.is_directed():
(12)                        result = {(n, n + 1) for n in range(1, 8)}
(12)                        result.update({(1, 0), (0, 2)})
(12)                        assert ev ^ some_edges == result
(8)                     else:
(12)                        result = {(n, n + 1) for n in range(1, 8)}
(12)                        result.update({(0, 2)})
(12)                        assert ev ^ some_edges == result
(8)                     return
(4)                 def test_sub(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1), (1, 0), (0, 2)}
(8)                     result = {(n, n + 1) for n in range(8)}
(8)                     result.remove((0, 1))
(8)                     assert ev - some_edges, result
(0)             class TestOutEdgeView(TestEdgeView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, nx.DiGraph())
(8)                     cls.eview = nx.reportviews.OutEdgeView
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)
(8)                     rep = (
(12)                        "OutEdgeView([(0, 1), (1, 2), (2, 3), (3, 4), "
(12)                        + "(4, 5), (5, 6), (6, 7), (7, 8)])"
```

```
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     ev = self.eview(self.G)
(8)                     evn = ev(nbunch=[0, 2])
(8)                     assert (0, 1) in evn
(8)                     assert (1, 2) not in evn
(8)                     assert (2, 3) in evn
(8)                     assert (3, 4) not in evn
(8)                     assert (4, 5) not in evn
(8)                     assert (5, 6) not in evn
(8)                     assert (7, 8) not in evn
(8)                     assert (8, 9) not in evn
(0)             class TestInEdgeView(TestEdgeView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, nx.DiGraph())
(8)                     cls.eview = nx.reportviews.InEdgeView
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)
(8)                     rep = (
(12)                        "InEdgeView([(0, 1), (1, 2), (2, 3), (3, 4), "
(12)                        + "(4, 5), (5, 6), (6, 7), (7, 8)])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     ev = self.eview(self.G)
(8)                     evn = ev(nbunch=[0, 2])
(8)                     assert (0, 1) not in evn
(8)                     assert (1, 2) in evn
(8)                     assert (2, 3) not in evn
(8)                     assert (3, 4) not in evn
(8)                     assert (4, 5) not in evn
(8)                     assert (5, 6) not in evn
(8)                     assert (7, 8) not in evn
(8)                     assert (8, 9) not in evn
(0)             class TestMultiEdgeView(TestEdgeView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, nx.MultiGraph())
(8)                     cls.G.add_edge(1, 2, key=3, foo="bar")
(8)                     cls.eview = nx.reportviews.MultiEdgeView
(4)                 def modify_edge(self, G, e, **kwds):
(8)                     if len(e) == 2:
(12)                        e = e + (0,)
(8)                     G._adj[e[0]][e[1]][e[2]].update(kwds)
(4)                 def test_str(self):
(8)                     ev = self.eview(self.G)
(8)                     replist = [(n, n + 1, 0) for n in range(8)]
(8)                     replist.insert(2, (1, 2, 3))
(8)                     rep = str(replist)
(8)                     assert str(ev) == rep
(4)                 def test_getitem(self):
(8)                     G = self.G.copy()
(8)                     ev = G.edges
(8)                     G.edges[0, 1, 0]["foo"] = "bar"
(8)                     assert ev[0, 1, 0] == {"foo": "bar"}
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        G.edges[0:5]
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)
(8)                     rep = (
(12)                        "MultiEdgeView([(0, 1, 0), (1, 2, 0), (1, 2, 3), (2, 3, 0), "
(12)                        + "(3, 4, 0), (4, 5, 0), (5, 6, 0), (6, 7, 0), (7, 8, 0)])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_call(self):
(8)                     ev = self.eview(self.G)
(8)                     assert id(ev) == id(ev(keys=True))
```

```
(8)                 assert id(ev) == id(ev(data=False, keys=True))
(8)                 assert id(ev) != id(ev(keys=False))
(8)                 assert id(ev) != id(ev(data=True))
(8)                 assert id(ev) != id(ev(nbunch=1))
(4)             def test_data(self):
(8)                 ev = self.eview(self.G)
(8)                 assert id(ev) != id(ev.data())
(8)                 assert id(ev) == id(ev.data(data=False, keys=True))
(8)                 assert id(ev) != id(ev.data(keys=False))
(8)                 assert id(ev) != id(ev.data(data=True))
(8)                 assert id(ev) != id(ev.data(nbunch=1))
(4)             def test_iter(self):
(8)                 ev = self.eview(self.G)
(8)                 for u, v, k in ev:
(12)                    pass
(8)                 iev = iter(ev)
(8)                 assert next(iev) == (0, 1, 0)
(8)                 assert iter(ev) != ev
(8)                 assert iter(iev) == iev
(4)             def test_iterkeys(self):
(8)                 G = self.G
(8)                 evr = self.eview(G)
(8)                 ev = evr(keys=True)
(8)                 for u, v, k in ev:
(12)                    pass
(8)                 assert k == 0
(8)                 ev = evr(keys=True, data="foo", default=1)
(8)                 for u, v, k, wt in ev:
(12)                    pass
(8)                 assert wt == 1
(8)                 self.modify_edge(G, (2, 3, 0), foo="bar")
(8)                 ev = evr(keys=True, data=True)
(8)                 for e in ev:
(12)                    assert len(e) == 4
(12)                    print("edge:", e)
(12)                    if set(e[:2]) == {2, 3}:
(16)                        print(self.G._adj[2][3])
(16)                        assert e[2] == 0
(16)                        assert e[3] == {"foo": "bar"}
(16)                        checked = True
(12)                    elif set(e[:3]) == {1, 2, 3}:
(16)                        assert e[2] == 3
(16)                        assert e[3] == {"foo": "bar"}
(16)                        checked_multi = True
(12)                    else:
(16)                        assert e[2] == 0
(16)                        assert e[3] == {}
(8)                 assert checked
(8)                 assert checked_multi
(8)                 ev = evr(keys=True, data="foo", default=1)
(8)                 for e in ev:
(12)                    if set(e[:2]) == {1, 2} and e[2] == 3:
(16)                        assert e[3] == "bar"
(12)                    if set(e[:2]) == {1, 2} and e[2] == 0:
(16)                        assert e[3] == 1
(12)                    if set(e[:2]) == {2, 3}:
(16)                        assert e[2] == 0
(16)                        assert e[3] == "bar"
(16)                        assert len(e) == 4
(16)                        checked_wt = True
(8)                 assert checked_wt
(8)                 ev = evr(keys=True)
(8)                 for e in ev:
(12)                    assert len(e) == 3
(8)                 elist = sorted([(i, i + 1, 0) for i in range(8)] + [(1, 2, 3)])
(8)                 assert sorted(ev) == elist
(8)                 ev = evr((1, 2), "foo", keys=True, default=1)
(8)                 with pytest.raises(TypeError):
(12)                    evr((1, 2), "foo", True, 1)
```

```
(8)                     with pytest.raises(TypeError):
(12)                        evr((1, 2), "foo", True, default=1)
(8)                     for e in ev:
(12)                        if set(e[:2]) == {1, 2}:
(16)                            assert e[2] in {0, 3}
(16)                            if e[2] == 3:
(20)                                assert e[3] == "bar"
(16)                            else:  # e[2] == 0
(20)                                assert e[3] == 1
(8)                     if G.is_directed():
(12)                        assert len(list(ev)) == 3
(8)                     else:
(12)                        assert len(list(ev)) == 4
(4)                 def test_or(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1, 0), (1, 0, 0), (0, 2, 0)}
(8)                     result = {(n, n + 1, 0) for n in range(8)}
(8)                     result.update(some_edges)
(8)                     result.update({(1, 2, 3)})
(8)                     assert ev | some_edges == result
(8)                     assert some_edges | ev == result
(4)                 def test_sub(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1, 0), (1, 0, 0), (0, 2, 0)}
(8)                     result = {(n, n + 1, 0) for n in range(8)}
(8)                     result.remove((0, 1, 0))
(8)                     result.update({(1, 2, 3)})
(8)                     assert ev - some_edges, result
(8)                     assert some_edges - ev, result
(4)                 def test_xor(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1, 0), (1, 0, 0), (0, 2, 0)}
(8)                     if self.G.is_directed():
(12)                        result = {(n, n + 1, 0) for n in range(1, 8)}
(12)                        result.update({(1, 0, 0), (0, 2, 0), (1, 2, 3)})
(12)                        assert ev ^ some_edges == result
(12)                        assert some_edges ^ ev == result
(8)                     else:
(12)                        result = {(n, n + 1, 0) for n in range(1, 8)}
(12)                        result.update({(0, 2, 0), (1, 2, 3)})
(12)                        assert ev ^ some_edges == result
(12)                        assert some_edges ^ ev == result
(4)                 def test_and(self):
(8)                     ev = self.eview(self.G)
(8)                     some_edges = {(0, 1, 0), (1, 0, 0), (0, 2, 0)}
(8)                     if self.G.is_directed():
(12)                        assert ev & some_edges == {(0, 1, 0)}
(12)                        assert some_edges & ev == {(0, 1, 0)}
(8)                     else:
(12)                        assert ev & some_edges == {(0, 1, 0), (1, 0, 0)}
(12)                        assert some_edges & ev == {(0, 1, 0), (1, 0, 0)}
(4)                 def test_contains_with_nbunch(self):
(8)                     ev = self.eview(self.G)
(8)                     evn = ev(nbunch=[0, 2])
(8)                     assert (0, 1) in evn
(8)                     assert (1, 2) in evn
(8)                     assert (2, 3) in evn
(8)                     assert (3, 4) not in evn
(8)                     assert (4, 5) not in evn
(8)                     assert (5, 6) not in evn
(8)                     assert (7, 8) not in evn
(8)                     assert (8, 9) not in evn
(0)             class TestOutMultiEdgeView(TestMultiEdgeView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, nx.MultiDiGraph())
(8)                     cls.G.add_edge(1, 2, key=3, foo="bar")
(8)                     cls.eview = nx.reportviews.OutMultiEdgeView
(4)                 def modify_edge(self, G, e, **kwds):
```

```
(8)                     if len(e) == 2:
(12)                        e = e + (0,)
(8)                     G._adj[e[0]][e[1]][e[2]].update(kwds)
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)
(8)                     rep = (
(12)                        "OutMultiEdgeView([(0, 1, 0), (1, 2, 0), (1, 2, 3), (2, 3, 0),"
(12)                        + " (3, 4, 0), (4, 5, 0), (5, 6, 0), (6, 7, 0), (7, 8, 0)])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     ev = self.eview(self.G)
(8)                     evn = ev(nbunch=[0, 2])
(8)                     assert (0, 1) in evn
(8)                     assert (1, 2) not in evn
(8)                     assert (2, 3) in evn
(8)                     assert (3, 4) not in evn
(8)                     assert (4, 5) not in evn
(8)                     assert (5, 6) not in evn
(8)                     assert (7, 8) not in evn
(8)                     assert (8, 9) not in evn
(0)             class TestInMultiEdgeView(TestMultiEdgeView):
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(9, nx.MultiDiGraph())
(8)                     cls.G.add_edge(1, 2, key=3, foo="bar")
(8)                     cls.eview = nx.reportviews.InMultiEdgeView
(4)                 def modify_edge(self, G, e, **kwds):
(8)                     if len(e) == 2:
(12)                        e = e + (0,)
(8)                     G._adj[e[0]][e[1]][e[2]].update(kwds)
(4)                 def test_repr(self):
(8)                     ev = self.eview(self.G)
(8)                     rep = (
(12)                        "InMultiEdgeView([(0, 1, 0), (1, 2, 0), (1, 2, 3), (2, 3, 0), "
(12)                        + "(3, 4, 0), (4, 5, 0), (5, 6, 0), (6, 7, 0), (7, 8, 0)])"
(8)                     )
(8)                     assert repr(ev) == rep
(4)                 def test_contains_with_nbunch(self):
(8)                     ev = self.eview(self.G)
(8)                     evn = ev(nbunch=[0, 2])
(8)                     assert (0, 1) not in evn
(8)                     assert (1, 2) in evn
(8)                     assert (2, 3) not in evn
(8)                     assert (3, 4) not in evn
(8)                     assert (4, 5) not in evn
(8)                     assert (5, 6) not in evn
(8)                     assert (7, 8) not in evn
(8)                     assert (8, 9) not in evn
(0)             class TestDegreeView:
(4)                 GRAPH = nx.Graph
(4)                 dview = nx.reportviews.DegreeView
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.G = nx.path_graph(6, cls.GRAPH())
(8)                     cls.G.add_edge(1, 3, foo=2)
(8)                     cls.G.add_edge(1, 3, foo=3)
(4)                 def test_pickle(self):
(8)                     import pickle
(8)                     deg = self.G.degree
(8)                     pdeg = pickle.loads(pickle.dumps(deg, -1))
(8)                     assert dict(deg) == dict(pdeg)
(4)                 def test_str(self):
(8)                     dv = self.dview(self.G)
(8)                     rep = str([(0, 1), (1, 3), (2, 2), (3, 3), (4, 2), (5, 1)])
(8)                     assert str(dv) == rep
(8)                     dv = self.G.degree()
(8)                     assert str(dv) == rep
(4)                 def test_repr(self):
```

```
(8)                     dv = self.dview(self.G)
(8)                     rep = "DegreeView({0: 1, 1: 3, 2: 2, 3: 3, 4: 2, 5: 1})"
(8)                     assert repr(dv) == rep
(4)                 def test_iter(self):
(8)                     dv = self.dview(self.G)
(8)                     for n, d in dv:
(12)                        pass
(8)                     idv = iter(dv)
(8)                     assert iter(dv) != dv
(8)                     assert iter(idv) == idv
(8)                     assert next(idv) == (0, dv[0])
(8)                     assert next(idv) == (1, dv[1])
(8)                     dv = self.dview(self.G, weight="foo")
(8)                     for n, d in dv:
(12)                        pass
(8)                     idv = iter(dv)
(8)                     assert iter(dv) != dv
(8)                     assert iter(idv) == idv
(8)                     assert next(idv) == (0, dv[0])
(8)                     assert next(idv) == (1, dv[1])
(4)                 def test_nbunch(self):
(8)                     dv = self.dview(self.G)
(8)                     dvn = dv(0)
(8)                     assert dvn == 1
(8)                     dvn = dv([2, 3])
(8)                     assert sorted(dvn) == [(2, 2), (3, 3)]
(4)                 def test_getitem(self):
(8)                     dv = self.dview(self.G)
(8)                     assert dv[0] == 1
(8)                     assert dv[1] == 3
(8)                     assert dv[2] == 2
(8)                     assert dv[3] == 3
(8)                     dv = self.dview(self.G, weight="foo")
(8)                     assert dv[0] == 1
(8)                     assert dv[1] == 5
(8)                     assert dv[2] == 2
(8)                     assert dv[3] == 5
(4)                 def test_weight(self):
(8)                     dv = self.dview(self.G)
(8)                     dvw = dv(0, weight="foo")
(8)                     assert dvw == 1
(8)                     dvw = dv(1, weight="foo")
(8)                     assert dvw == 5
(8)                     dvw = dv([2, 3], weight="foo")
(8)                     assert sorted(dvw) == [(2, 2), (3, 5)]
(8)                     dvd = dict(dv(weight="foo"))
(8)                     assert dvd[0] == 1
(8)                     assert dvd[1] == 5
(8)                     assert dvd[2] == 2
(8)                     assert dvd[3] == 5
(4)                 def test_len(self):
(8)                     dv = self.dview(self.G)
(8)                     assert len(dv) == 6
(0)             class TestDiDegreeView(TestDegreeView):
(4)                 GRAPH = nx.DiGraph
(4)                 dview = nx.reportviews.DiDegreeView
(4)                 def test_repr(self):
(8)                     dv = self.G.degree()
(8)                     rep = "DiDegreeView({0: 1, 1: 3, 2: 2, 3: 3, 4: 2, 5: 1})"
(8)                     assert repr(dv) == rep
(0)             class TestOutDegreeView(TestDegreeView):
(4)                 GRAPH = nx.DiGraph
(4)                 dview = nx.reportviews.OutDegreeView
(4)                 def test_str(self):
(8)                     dv = self.dview(self.G)
(8)                     rep = str([(0, 1), (1, 2), (2, 1), (3, 1), (4, 1), (5, 0)])
(8)                     assert str(dv) == rep
(8)                     dv = self.G.out_degree()
(8)                     assert str(dv) == rep
```

```
(4)                 def test_repr(self):
(8)                     dv = self.G.out_degree()
(8)                     rep = "OutDegreeView({0: 1, 1: 2, 2: 1, 3: 1, 4: 1, 5: 0})"
(8)                     assert repr(dv) == rep
(4)                 def test_nbunch(self):
(8)                     dv = self.dview(self.G)
(8)                     dvn = dv(0)
(8)                     assert dvn == 1
(8)                     dvn = dv([2, 3])
(8)                     assert sorted(dvn) == [(2, 1), (3, 1)]
(4)                 def test_getitem(self):
(8)                     dv = self.dview(self.G)
(8)                     assert dv[0] == 1
(8)                     assert dv[1] == 2
(8)                     assert dv[2] == 1
(8)                     assert dv[3] == 1
(8)                     dv = self.dview(self.G, weight="foo")
(8)                     assert dv[0] == 1
(8)                     assert dv[1] == 4
(8)                     assert dv[2] == 1
(8)                     assert dv[3] == 1
(4)                 def test_weight(self):
(8)                     dv = self.dview(self.G)
(8)                     dvw = dv(0, weight="foo")
(8)                     assert dvw == 1
(8)                     dvw = dv(1, weight="foo")
(8)                     assert dvw == 4
(8)                     dvw = dv([2, 3], weight="foo")
(8)                     assert sorted(dvw) == [(2, 1), (3, 1)]
(8)                     dvd = dict(dv(weight="foo"))
(8)                     assert dvd[0] == 1
(8)                     assert dvd[1] == 4
(8)                     assert dvd[2] == 1
(8)                     assert dvd[3] == 1
(0)             class TestInDegreeView(TestDegreeView):
(4)                 GRAPH = nx.DiGraph
(4)                 dview = nx.reportviews.InDegreeView
(4)                 def test_str(self):
(8)                     dv = self.dview(self.G)
(8)                     rep = str([(0, 0), (1, 1), (2, 1), (3, 2), (4, 1), (5, 1)])
(8)                     assert str(dv) == rep
(8)                     dv = self.G.in_degree()
(8)                     assert str(dv) == rep
(4)                 def test_repr(self):
(8)                     dv = self.G.in_degree()
(8)                     rep = "InDegreeView({0: 0, 1: 1, 2: 1, 3: 2, 4: 1, 5: 1})"
(8)                     assert repr(dv) == rep
(4)                 def test_nbunch(self):
(8)                     dv = self.dview(self.G)
(8)                     dvn = dv(0)
(8)                     assert dvn == 0
(8)                     dvn = dv([2, 3])
(8)                     assert sorted(dvn) == [(2, 1), (3, 2)]
(4)                 def test_getitem(self):
(8)                     dv = self.dview(self.G)
(8)                     assert dv[0] == 0
(8)                     assert dv[1] == 1
(8)                     assert dv[2] == 1
(8)                     assert dv[3] == 2
(8)                     dv = self.dview(self.G, weight="foo")
(8)                     assert dv[0] == 0
(8)                     assert dv[1] == 1
(8)                     assert dv[2] == 1
(8)                     assert dv[3] == 4
(4)                 def test_weight(self):
(8)                     dv = self.dview(self.G)
(8)                     dvw = dv(0, weight="foo")
(8)                     assert dvw == 0
(8)                     dvw = dv(1, weight="foo")
```

```
(8)                 assert dvw == 1
(8)                 dvw = dv([2, 3], weight="foo")
(8)                 assert sorted(dvw) == [(2, 1), (3, 4)]
(8)                 dvd = dict(dv(weight="foo"))
(8)                 assert dvd[0] == 0
(8)                 assert dvd[1] == 1
(8)                 assert dvd[2] == 1
(8)                 assert dvd[3] == 4
(0)         class TestMultiDegreeView(TestDegreeView):
(4)             GRAPH = nx.MultiGraph
(4)             dview = nx.reportviews.MultiDegreeView
(4)             def test_str(self):
(8)                 dv = self.dview(self.G)
(8)                 rep = str([(0, 1), (1, 4), (2, 2), (3, 4), (4, 2), (5, 1)])
(8)                 assert str(dv) == rep
(8)                 dv = self.G.degree()
(8)                 assert str(dv) == rep
(4)             def test_repr(self):
(8)                 dv = self.G.degree()
(8)                 rep = "MultiDegreeView({0: 1, 1: 4, 2: 2, 3: 4, 4: 2, 5: 1})"
(8)                 assert repr(dv) == rep
(4)             def test_nbunch(self):
(8)                 dv = self.dview(self.G)
(8)                 dvn = dv(0)
(8)                 assert dvn == 1
(8)                 dvn = dv([2, 3])
(8)                 assert sorted(dvn) == [(2, 2), (3, 4)]
(4)             def test_getitem(self):
(8)                 dv = self.dview(self.G)
(8)                 assert dv[0] == 1
(8)                 assert dv[1] == 4
(8)                 assert dv[2] == 2
(8)                 assert dv[3] == 4
(8)                 dv = self.dview(self.G, weight="foo")
(8)                 assert dv[0] == 1
(8)                 assert dv[1] == 7
(8)                 assert dv[2] == 2
(8)                 assert dv[3] == 7
(4)             def test_weight(self):
(8)                 dv = self.dview(self.G)
(8)                 dvw = dv(0, weight="foo")
(8)                 assert dvw == 1
(8)                 dvw = dv(1, weight="foo")
(8)                 assert dvw == 7
(8)                 dvw = dv([2, 3], weight="foo")
(8)                 assert sorted(dvw) == [(2, 2), (3, 7)]
(8)                 dvd = dict(dv(weight="foo"))
(8)                 assert dvd[0] == 1
(8)                 assert dvd[1] == 7
(8)                 assert dvd[2] == 2
(8)                 assert dvd[3] == 7
(0)         class TestDiMultiDegreeView(TestMultiDegreeView):
(4)             GRAPH = nx.MultiDiGraph
(4)             dview = nx.reportviews.DiMultiDegreeView
(4)             def test_repr(self):
(8)                 dv = self.G.degree()
(8)                 rep = "DiMultiDegreeView({0: 1, 1: 4, 2: 2, 3: 4, 4: 2, 5: 1})"
(8)                 assert repr(dv) == rep
(0)         class TestOutMultiDegreeView(TestDegreeView):
(4)             GRAPH = nx.MultiDiGraph
(4)             dview = nx.reportviews.OutMultiDegreeView
(4)             def test_str(self):
(8)                 dv = self.dview(self.G)
(8)                 rep = str([(0, 1), (1, 3), (2, 1), (3, 1), (4, 1), (5, 0)])
(8)                 assert str(dv) == rep
(8)                 dv = self.G.out_degree()
(8)                 assert str(dv) == rep
(4)             def test_repr(self):
(8)                 dv = self.G.out_degree()
```

```
(8)                 rep = "OutMultiDegreeView({0: 1, 1: 3, 2: 1, 3: 1, 4: 1, 5: 0})"
(8)                 assert repr(dv) == rep
(4)             def test_nbunch(self):
(8)                 dv = self.dview(self.G)
(8)                 dvn = dv(0)
(8)                 assert dvn == 1
(8)                 dvn = dv([2, 3])
(8)                 assert sorted(dvn) == [(2, 1), (3, 1)]
(4)             def test_getitem(self):
(8)                 dv = self.dview(self.G)
(8)                 assert dv[0] == 1
(8)                 assert dv[1] == 3
(8)                 assert dv[2] == 1
(8)                 assert dv[3] == 1
(8)                 dv = self.dview(self.G, weight="foo")
(8)                 assert dv[0] == 1
(8)                 assert dv[1] == 6
(8)                 assert dv[2] == 1
(8)                 assert dv[3] == 1
(4)             def test_weight(self):
(8)                 dv = self.dview(self.G)
(8)                 dvw = dv(0, weight="foo")
(8)                 assert dvw == 1
(8)                 dvw = dv(1, weight="foo")
(8)                 assert dvw == 6
(8)                 dvw = dv([2, 3], weight="foo")
(8)                 assert sorted(dvw) == [(2, 1), (3, 1)]
(8)                 dvd = dict(dv(weight="foo"))
(8)                 assert dvd[0] == 1
(8)                 assert dvd[1] == 6
(8)                 assert dvd[2] == 1
(8)                 assert dvd[3] == 1
(0)         class TestInMultiDegreeView(TestDegreeView):
(4)             GRAPH = nx.MultiDiGraph
(4)             dview = nx.reportviews.InMultiDegreeView
(4)             def test_str(self):
(8)                 dv = self.dview(self.G)
(8)                 rep = str([(0, 0), (1, 1), (2, 1), (3, 3), (4, 1), (5, 1)])
(8)                 assert str(dv) == rep
(8)                 dv = self.G.in_degree()
(8)                 assert str(dv) == rep
(4)             def test_repr(self):
(8)                 dv = self.G.in_degree()
(8)                 rep = "InMultiDegreeView({0: 0, 1: 1, 2: 1, 3: 3, 4: 1, 5: 1})"
(8)                 assert repr(dv) == rep
(4)             def test_nbunch(self):
(8)                 dv = self.dview(self.G)
(8)                 dvn = dv(0)
(8)                 assert dvn == 0
(8)                 dvn = dv([2, 3])
(8)                 assert sorted(dvn) == [(2, 1), (3, 3)]
(4)             def test_getitem(self):
(8)                 dv = self.dview(self.G)
(8)                 assert dv[0] == 0
(8)                 assert dv[1] == 1
(8)                 assert dv[2] == 1
(8)                 assert dv[3] == 3
(8)                 dv = self.dview(self.G, weight="foo")
(8)                 assert dv[0] == 0
(8)                 assert dv[1] == 1
(8)                 assert dv[2] == 1
(8)                 assert dv[3] == 6
(4)             def test_weight(self):
(8)                 dv = self.dview(self.G)
(8)                 dvw = dv(0, weight="foo")
(8)                 assert dvw == 0
(8)                 dvw = dv(1, weight="foo")
(8)                 assert dvw == 1
(8)                 dvw = dv([2, 3], weight="foo")
```

```
1148: (8)                   assert sorted(dvw) == [(2, 1), (3, 6)]
1149: (8)                   dvd = dict(dv(weight="foo"))
1150: (8)                   assert dvd[0] == 0
1151: (8)                   assert dvd[1] == 1
1152: (8)                   assert dvd[2] == 1
1153: (8)                   assert dvd[3] == 6
1154: (0)           @pytest.mark.parametrize(
1155: (4)               ("reportview", "err_msg_terms"),
1156: (4)               (
1157: (8)                   (rv.NodeView, "list(G.nodes"),
1158: (8)                   (rv.NodeDataView, "list(G.nodes.data"),
1159: (8)                   (rv.EdgeView, "list(G.edges"),
1160: (8)                   (rv.InEdgeView, "list(G.in_edges"),
1161: (8)                   (rv.OutEdgeView, "list(G.edges"),
1162: (8)                   (rv.MultiEdgeView, "list(G.edges"),
1163: (8)                   (rv.InMultiEdgeView, "list(G.in_edges"),
1164: (8)                   (rv.OutMultiEdgeView, "list(G.edges"),
1165: (4)               ),
1166: (0)           )
1167: (0)           def test_slicing_reportviews(reportview, err_msg_terms):
1168: (4)               G = nx.complete_graph(3)
1169: (4)               view = reportview(G)
1170: (4)               with pytest.raises(nx.NetworkXError) as exc:
1171: (8)                   view[0:2]
1172: (4)               errmsg = str(exc.value)
1173: (4)               assert type(view).__name__ in errmsg
1174: (4)               assert err_msg_terms in errmsg
1175: (0)           @pytest.mark.parametrize(
1176: (4)               "graph", [nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph]
1177: (0)           )
1178: (0)           def test_cache_dict_get_set_state(graph):
1179: (4)               G = nx.path_graph(5, graph())
1180: (4)               G.nodes, G.edges, G.adj, G.degree
1181: (4)               if G.is_directed():
1182: (8)                   G.pred, G.succ, G.in_edges, G.out_edges, G.in_degree, G.out_degree
1183: (4)               cached_dict = G.__dict__
1184: (4)               assert "nodes" in cached_dict
1185: (4)               assert "edges" in cached_dict
1186: (4)               assert "adj" in cached_dict
1187: (4)               assert "degree" in cached_dict
1188: (4)               if G.is_directed():
1189: (8)                   assert "pred" in cached_dict
1190: (8)                   assert "succ" in cached_dict
1191: (8)                   assert "in_edges" in cached_dict
1192: (8)                   assert "out_edges" in cached_dict
1193: (8)                   assert "in_degree" in cached_dict
1194: (8)                   assert "out_degree" in cached_dict
1195: (4)               pickle.loads(pickle.dumps(G, -1))
1196: (4)               deepcopy(G)
```

File 412 - test_special.py:

```
1: (0)              import networkx as nx
2: (0)              from .test_digraph import BaseDiGraphTester
3: (0)              from .test_digraph import TestDiGraph as _TestDiGraph
4: (0)              from .test_graph import BaseGraphTester
5: (0)              from .test_graph import TestGraph as _TestGraph
6: (0)              from .test_multidigraph import TestMultiDiGraph as _TestMultiDiGraph
7: (0)              from .test_multigraph import TestMultiGraph as _TestMultiGraph
8: (0)              def test_factories():
9: (4)                  class mydict1(dict):
10: (8)                     pass
11: (4)                 class mydict2(dict):
12: (8)                     pass
13: (4)                 class mydict3(dict):
14: (8)                     pass
15: (4)                 class mydict4(dict):
```

```
16: (8)                     pass
17: (4)                 class mydict5(dict):
18: (8)                     pass
19: (4)                 for Graph in (nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph):
20: (8)                     class MyGraph(Graph):
21: (12)                        node_dict_factory = mydict1
22: (12)                        adjlist_outer_dict_factory = mydict2
23: (12)                        adjlist_inner_dict_factory = mydict3
24: (12)                        edge_key_dict_factory = mydict4
25: (12)                        edge_attr_dict_factory = mydict5
26: (8)                     G = MyGraph()
27: (8)                     assert isinstance(G._node, mydict1)
28: (8)                     assert isinstance(G._adj, mydict2)
29: (8)                     G.add_node(1)
30: (8)                     assert isinstance(G._adj[1], mydict3)
31: (8)                     if G.is_directed():
32: (12)                        assert isinstance(G._pred, mydict2)
33: (12)                        assert isinstance(G._succ, mydict2)
34: (12)                        assert isinstance(G._pred[1], mydict3)
35: (8)                     G.add_edge(1, 2)
36: (8)                     if G.is_multigraph():
37: (12)                        assert isinstance(G._adj[1][2], mydict4)
38: (12)                        assert isinstance(G._adj[1][2][0], mydict5)
39: (8)                     else:
40: (12)                        assert isinstance(G._adj[1][2], mydict5)
41: (0)             class TestSpecialGraph(_TestGraph):
42: (4)                 def setup_method(self):
43: (8)                     _TestGraph.setup_method(self)
44: (8)                     self.Graph = nx.Graph
45: (0)             class TestThinGraph(BaseGraphTester):
46: (4)                 def setup_method(self):
47: (8)                     all_edge_dict = {"weight": 1}
48: (8)                     class MyGraph(nx.Graph):
49: (12)                        def edge_attr_dict_factory(self):
50: (16)                            return all_edge_dict
51: (8)                     self.Graph = MyGraph
52: (8)                     ed1, ed2, ed3 = (all_edge_dict, all_edge_dict, all_edge_dict)
53: (8)                     self.k3adj = {0: {1: ed1, 2: ed2}, 1: {0: ed1, 2: ed3}, 2: {0: ed2, 1:
ed3}}
54: (8)                     self.k3edges = [(0, 1), (0, 2), (1, 2)]
55: (8)                     self.k3nodes = [0, 1, 2]
56: (8)                     self.K3 = self.Graph()
57: (8)                     self.K3._adj = self.k3adj
58: (8)                     self.K3._node = {}
59: (8)                     self.K3._node[0] = {}
60: (8)                     self.K3._node[1] = {}
61: (8)                     self.K3._node[2] = {}
62: (0)             class TestSpecialDiGraph(_TestDiGraph):
63: (4)                 def setup_method(self):
64: (8)                     _TestDiGraph.setup_method(self)
65: (8)                     self.Graph = nx.DiGraph
66: (0)             class TestThinDiGraph(BaseDiGraphTester):
67: (4)                 def setup_method(self):
68: (8)                     all_edge_dict = {"weight": 1}
69: (8)                     class MyGraph(nx.DiGraph):
70: (12)                        def edge_attr_dict_factory(self):
71: (16)                            return all_edge_dict
72: (8)                     self.Graph = MyGraph
73: (8)                     ed1, ed2, ed3 = (all_edge_dict, all_edge_dict, all_edge_dict)
74: (8)                     ed4, ed5, ed6 = (all_edge_dict, all_edge_dict, all_edge_dict)
75: (8)                     self.k3adj = {0: {1: ed1, 2: ed2}, 1: {0: ed3, 2: ed4}, 2: {0: ed5, 1:
ed6}}
76: (8)                     self.k3edges = [(0, 1), (0, 2), (1, 2)]
77: (8)                     self.k3nodes = [0, 1, 2]
78: (8)                     self.K3 = self.Graph()
79: (8)                     self.K3._succ = self.k3adj
80: (8)                     self.K3._pred = {0: {1: ed3, 2: ed5}, 1: {0: ed1, 2: ed6}, 2: {0: ed2,
1: ed4}}
81: (8)                     self.K3._node = {}
```

```
82: (8)                     self.K3._node[0] = {}
83: (8)                     self.K3._node[1] = {}
84: (8)                     self.K3._node[2] = {}
85: (8)                     ed1, ed2 = (all_edge_dict, all_edge_dict)
86: (8)                     self.P3 = self.Graph()
87: (8)                     self.P3._succ = {0: {1: ed1}, 1: {2: ed2}, 2: {}}
88: (8)                     self.P3._pred = {0: {}, 1: {0: ed1}, 2: {1: ed2}}
89: (8)                     self.P3._node = {}
90: (8)                     self.P3._node[0] = {}
91: (8)                     self.P3._node[1] = {}
92: (8)                     self.P3._node[2] = {}
93: (0)             class TestSpecialMultiGraph(_TestMultiGraph):
94: (4)                 def setup_method(self):
95: (8)                     _TestMultiGraph.setup_method(self)
96: (8)                     self.Graph = nx.MultiGraph
97: (0)             class TestSpecialMultiDiGraph(_TestMultiDiGraph):
98: (4)                 def setup_method(self):
99: (8)                     _TestMultiDiGraph.setup_method(self)
100: (8)                    self.Graph = nx.MultiDiGraph

----------------------------------------

File 413 - __init__.py:

1: (0)              from .layout import *
2: (0)              from .nx_latex import *
3: (0)              from .nx_pylab import *
4: (0)              from . import nx_agraph
5: (0)              from . import nx_pydot

----------------------------------------

File 414 - test_subgraphviews.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import edges_equal
4: (0)              class TestSubGraphView:
5: (4)                  gview = staticmethod(nx.subgraph_view)
6: (4)                  graph = nx.Graph
7: (4)                  hide_edges_filter = staticmethod(nx.filters.hide_edges)
8: (4)                  show_edges_filter = staticmethod(nx.filters.show_edges)
9: (4)                  @classmethod
10: (4)                 def setup_class(cls):
11: (8)                     cls.G = nx.path_graph(9, create_using=cls.graph())
12: (8)                     cls.hide_edges_w_hide_nodes = {(3, 4), (4, 5), (5, 6)}
13: (4)                 def test_hidden_nodes(self):
14: (8)                     hide_nodes = [4, 5, 111]
15: (8)                     nodes_gone = nx.filters.hide_nodes(hide_nodes)
16: (8)                     gview = self.gview
17: (8)                     G = gview(self.G, filter_node=nodes_gone)
18: (8)                     assert self.G.nodes - G.nodes == {4, 5}
19: (8)                     assert self.G.edges - G.edges == self.hide_edges_w_hide_nodes
20: (8)                     if G.is_directed():
21: (12)                        assert list(G[3]) == []
22: (12)                        assert list(G[2]) == [3]
23: (8)                     else:
24: (12)                        assert list(G[3]) == [2]
25: (12)                        assert set(G[2]) == {1, 3}
26: (8)                     pytest.raises(KeyError, G.__getitem__, 4)
27: (8)                     pytest.raises(KeyError, G.__getitem__, 112)
28: (8)                     pytest.raises(KeyError, G.__getitem__, 111)
29: (8)                     assert G.degree(3) == (3 if G.is_multigraph() else 1)
30: (8)                     assert G.size() == (7 if G.is_multigraph() else 5)
31: (4)                 def test_hidden_edges(self):
32: (8)                     hide_edges = [(2, 3), (8, 7), (222, 223)]
33: (8)                     edges_gone = self.hide_edges_filter(hide_edges)
34: (8)                     gview = self.gview
35: (8)                     G = gview(self.G, filter_edge=edges_gone)
```

```
36: (8)                     assert self.G.nodes == G.nodes
37: (8)                     if G.is_directed():
38: (12)                        assert self.G.edges - G.edges == {(2, 3)}
39: (12)                        assert list(G[2]) == []
40: (12)                        assert list(G.pred[3]) == []
41: (12)                        assert list(G.pred[2]) == [1]
42: (12)                        assert G.size() == 7
43: (8)                     else:
44: (12)                        assert self.G.edges - G.edges == {(2, 3), (7, 8)}
45: (12)                        assert list(G[2]) == [1]
46: (12)                        assert G.size() == 6
47: (8)                     assert list(G[3]) == [4]
48: (8)                     pytest.raises(KeyError, G.__getitem__, 221)
49: (8)                     pytest.raises(KeyError, G.__getitem__, 222)
50: (8)                     assert G.degree(3) == 1
51: (4)                 def test_shown_node(self):
52: (8)                     induced_subgraph = nx.filters.show_nodes([2, 3, 111])
53: (8)                     gview = self.gview
54: (8)                     G = gview(self.G, filter_node=induced_subgraph)
55: (8)                     assert set(G.nodes) == {2, 3}
56: (8)                     if G.is_directed():
57: (12)                        assert list(G[3]) == []
58: (8)                     else:
59: (12)                        assert list(G[3]) == [2]
60: (8)                     assert list(G[2]) == [3]
61: (8)                     pytest.raises(KeyError, G.__getitem__, 4)
62: (8)                     pytest.raises(KeyError, G.__getitem__, 112)
63: (8)                     pytest.raises(KeyError, G.__getitem__, 111)
64: (8)                     assert G.degree(3) == (3 if G.is_multigraph() else 1)
65: (8)                     assert G.size() == (3 if G.is_multigraph() else 1)
66: (4)                 def test_shown_edges(self):
67: (8)                     show_edges = [(2, 3), (8, 7), (222, 223)]
68: (8)                     edge_subgraph = self.show_edges_filter(show_edges)
69: (8)                     G = self.gview(self.G, filter_edge=edge_subgraph)
70: (8)                     assert self.G.nodes == G.nodes
71: (8)                     if G.is_directed():
72: (12)                        assert G.edges == {(2, 3)}
73: (12)                        assert list(G[3]) == []
74: (12)                        assert list(G[2]) == [3]
75: (12)                        assert list(G.pred[3]) == [2]
76: (12)                        assert list(G.pred[2]) == []
77: (12)                        assert G.size() == 1
78: (8)                     else:
79: (12)                        assert G.edges == {(2, 3), (7, 8)}
80: (12)                        assert list(G[3]) == [2]
81: (12)                        assert list(G[2]) == [3]
82: (12)                        assert G.size() == 2
83: (8)                     pytest.raises(KeyError, G.__getitem__, 221)
84: (8)                     pytest.raises(KeyError, G.__getitem__, 222)
85: (8)                     assert G.degree(3) == 1
86: (0)             class TestSubDiGraphView(TestSubGraphView):
87: (4)                 gview = staticmethod(nx.subgraph_view)
88: (4)                 graph = nx.DiGraph
89: (4)                 hide_edges_filter = staticmethod(nx.filters.hide_diedges)
90: (4)                 show_edges_filter = staticmethod(nx.filters.show_diedges)
91: (4)                 hide_edges = [(2, 3), (8, 7), (222, 223)]
92: (4)                 excluded = {(2, 3), (3, 4), (4, 5), (5, 6)}
93: (4)                 def test_inoutedges(self):
94: (8)                     edges_gone = self.hide_edges_filter(self.hide_edges)
95: (8)                     hide_nodes = [4, 5, 111]
96: (8)                     nodes_gone = nx.filters.hide_nodes(hide_nodes)
97: (8)                     G = self.gview(self.G, filter_node=nodes_gone, filter_edge=edges_gone)
98: (8)                     assert self.G.in_edges - G.in_edges == self.excluded
99: (8)                     assert self.G.out_edges - G.out_edges == self.excluded
100: (4)                 def test_pred(self):
101: (8)                     edges_gone = self.hide_edges_filter(self.hide_edges)
102: (8)                     hide_nodes = [4, 5, 111]
103: (8)                     nodes_gone = nx.filters.hide_nodes(hide_nodes)
104: (8)                     G = self.gview(self.G, filter_node=nodes_gone, filter_edge=edges_gone)
```

```
(8)                 assert list(G.pred[2]) == [1]
(8)                 assert list(G.pred[6]) == []
(4)             def test_inout_degree(self):
(8)                 edges_gone = self.hide_edges_filter(self.hide_edges)
(8)                 hide_nodes = [4, 5, 111]
(8)                 nodes_gone = nx.filters.hide_nodes(hide_nodes)
(8)                 G = self.gview(self.G, filter_node=nodes_gone, filter_edge=edges_gone)
(8)                 assert G.degree(2) == 1
(8)                 assert G.out_degree(2) == 0
(8)                 assert G.in_degree(2) == 1
(8)                 assert G.size() == 4
(0)         class TestMultiGraphView(TestSubGraphView):
(4)             gview = staticmethod(nx.subgraph_view)
(4)             graph = nx.MultiGraph
(4)             hide_edges_filter = staticmethod(nx.filters.hide_multiedges)
(4)             show_edges_filter = staticmethod(nx.filters.show_multiedges)
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 cls.G = nx.path_graph(9, create_using=cls.graph())
(8)                 multiedges = {(2, 3, 4), (2, 3, 5)}
(8)                 cls.G.add_edges_from(multiedges)
(8)                 cls.hide_edges_w_hide_nodes = {(3, 4, 0), (4, 5, 0), (5, 6, 0)}
(4)             def test_hidden_edges(self):
(8)                 hide_edges = [(2, 3, 4), (2, 3, 3), (8, 7, 0), (222, 223, 0)]
(8)                 edges_gone = self.hide_edges_filter(hide_edges)
(8)                 G = self.gview(self.G, filter_edge=edges_gone)
(8)                 assert self.G.nodes == G.nodes
(8)                 if G.is_directed():
(12)                    assert self.G.edges - G.edges == {(2, 3, 4)}
(12)                    assert list(G[3]) == [4]
(12)                    assert list(G[2]) == [3]
(12)                    assert list(G.pred[3]) == [2]  # only one 2 but two edges
(12)                    assert list(G.pred[2]) == [1]
(12)                    assert G.size() == 9
(8)                 else:
(12)                    assert self.G.edges - G.edges == {(2, 3, 4), (7, 8, 0)}
(12)                    assert list(G[3]) == [2, 4]
(12)                    assert list(G[2]) == [1, 3]
(12)                    assert G.size() == 8
(8)                 assert G.degree(3) == 3
(8)                 pytest.raises(KeyError, G.__getitem__, 221)
(8)                 pytest.raises(KeyError, G.__getitem__, 222)
(4)             def test_shown_edges(self):
(8)                 show_edges = [(2, 3, 4), (2, 3, 3), (8, 7, 0), (222, 223, 0)]
(8)                 edge_subgraph = self.show_edges_filter(show_edges)
(8)                 G = self.gview(self.G, filter_edge=edge_subgraph)
(8)                 assert self.G.nodes == G.nodes
(8)                 if G.is_directed():
(12)                    assert G.edges == {(2, 3, 4)}
(12)                    assert list(G[3]) == []
(12)                    assert list(G.pred[3]) == [2]
(12)                    assert list(G.pred[2]) == []
(12)                    assert G.size() == 1
(8)                 else:
(12)                    assert G.edges == {(2, 3, 4), (7, 8, 0)}
(12)                    assert G.size() == 2
(12)                    assert list(G[3]) == [2]
(8)                 assert G.degree(3) == 1
(8)                 assert list(G[2]) == [3]
(8)                 pytest.raises(KeyError, G.__getitem__, 221)
(8)                 pytest.raises(KeyError, G.__getitem__, 222)
(0)         class TestMultiDiGraphView(TestMultiGraphView, TestSubDiGraphView):
(4)             gview = staticmethod(nx.subgraph_view)
(4)             graph = nx.MultiDiGraph
(4)             hide_edges_filter = staticmethod(nx.filters.hide_multidiedges)
(4)             show_edges_filter = staticmethod(nx.filters.show_multidiedges)
(4)             hide_edges = [(2, 3, 0), (8, 7, 0), (222, 223, 0)]
(4)             excluded = {(2, 3, 0), (3, 4, 0), (4, 5, 0), (5, 6, 0)}
(4)             def test_inout_degree(self):
```

```
(8)                 edges_gone = self.hide_edges_filter(self.hide_edges)
(8)                 hide_nodes = [4, 5, 111]
(8)                 nodes_gone = nx.filters.hide_nodes(hide_nodes)
(8)                 G = self.gview(self.G, filter_node=nodes_gone, filter_edge=edges_gone)
(8)                 assert G.degree(2) == 3
(8)                 assert G.out_degree(2) == 2
(8)                 assert G.in_degree(2) == 1
(8)                 assert G.size() == 6
(0)         class TestInducedSubGraph:
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 cls.K3 = G = nx.complete_graph(3)
(8)                 G.graph["foo"] = []
(8)                 G.nodes[0]["foo"] = []
(8)                 G.remove_edge(1, 2)
(8)                 ll = []
(8)                 G.add_edge(1, 2, foo=ll)
(8)                 G.add_edge(2, 1, foo=ll)
(4)             def test_full_graph(self):
(8)                 G = self.K3
(8)                 H = nx.induced_subgraph(G, [0, 1, 2, 5])
(8)                 assert H.name == G.name
(8)                 self.graphs_equal(H, G)
(8)                 self.same_attrdict(H, G)
(4)             def test_partial_subgraph(self):
(8)                 G = self.K3
(8)                 H = nx.induced_subgraph(G, 0)
(8)                 assert dict(H.adj) == {0: {}}
(8)                 assert dict(G.adj) != {0: {}}
(8)                 H = nx.induced_subgraph(G, [0, 1])
(8)                 assert dict(H.adj) == {0: {1: {}}, 1: {0: {}}}
(4)             def same_attrdict(self, H, G):
(8)                 old_foo = H[1][2]["foo"]
(8)                 H.edges[1, 2]["foo"] = "baz"
(8)                 assert G.edges == H.edges
(8)                 H.edges[1, 2]["foo"] = old_foo
(8)                 assert G.edges == H.edges
(8)                 old_foo = H.nodes[0]["foo"]
(8)                 H.nodes[0]["foo"] = "baz"
(8)                 assert G.nodes == H.nodes
(8)                 H.nodes[0]["foo"] = old_foo
(8)                 assert G.nodes == H.nodes
(4)             def graphs_equal(self, H, G):
(8)                 assert G._adj == H._adj
(8)                 assert G._node == H._node
(8)                 assert G.graph == H.graph
(8)                 assert G.name == H.name
(8)                 if not G.is_directed() and not H.is_directed():
(12)                    assert H._adj[1][2] is H._adj[2][1]
(12)                    assert G._adj[1][2] is G._adj[2][1]
(8)                 else:  # at least one is directed
(12)                    if not G.is_directed():
(16)                        G._pred = G._adj
(16)                        G._succ = G._adj
(12)                    if not H.is_directed():
(16)                        H._pred = H._adj
(16)                        H._succ = H._adj
(12)                    assert G._pred == H._pred
(12)                    assert G._succ == H._succ
(12)                    assert H._succ[1][2] is H._pred[2][1]
(12)                    assert G._succ[1][2] is G._pred[2][1]
(0)         class TestEdgeSubGraph:
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 cls.G = G = nx.path_graph(5)
(8)                 for i in range(5):
(12)                    G.nodes[i]["name"] = f"node{i}"
(8)                 G.edges[0, 1]["name"] = "edge01"
(8)                 G.edges[3, 4]["name"] = "edge34"
```

```
243: (8)                    G.graph["name"] = "graph"
244: (8)                    cls.H = nx.edge_subgraph(G, [(0, 1), (3, 4)])
245: (4)                def test_correct_nodes(self):
246: (8)                    """Tests that the subgraph has the correct nodes."""
247: (8)                    assert [(0, "node0"), (1, "node1"), (3, "node3"), (4, "node4")] ==
sorted(
248: (12)                       self.H.nodes.data("name")
249: (8)                    )
250: (4)                def test_correct_edges(self):
251: (8)                    """Tests that the subgraph has the correct edges."""
252: (8)                    assert edges_equal(
253: (12)                       [(0, 1, "edge01"), (3, 4, "edge34")], self.H.edges.data("name")
254: (8)                    )
255: (4)                def test_add_node(self):
256: (8)                    """Tests that adding a node to the original graph does not
257: (8)                    affect the nodes of the subgraph.
258: (8)                    """
259: (8)                    self.G.add_node(5)
260: (8)                    assert [0, 1, 3, 4] == sorted(self.H.nodes)
261: (8)                    self.G.remove_node(5)
262: (4)                def test_remove_node(self):
263: (8)                    """Tests that removing a node in the original graph
264: (8)                    removes the nodes of the subgraph.
265: (8)                    """
266: (8)                    self.G.remove_node(0)
267: (8)                    assert [1, 3, 4] == sorted(self.H.nodes)
268: (8)                    self.G.add_node(0, name="node0")
269: (8)                    self.G.add_edge(0, 1, name="edge01")
270: (4)                def test_node_attr_dict(self):
271: (8)                    """Tests that the node attribute dictionary of the two graphs is
272: (8)                    the same object.
273: (8)                    """
274: (8)                    for v in self.H:
275: (12)                       assert self.G.nodes[v] == self.H.nodes[v]
276: (8)                    self.G.nodes[0]["name"] = "foo"
277: (8)                    assert self.G.nodes[0] == self.H.nodes[0]
278: (8)                    self.H.nodes[1]["name"] = "bar"
279: (8)                    assert self.G.nodes[1] == self.H.nodes[1]
280: (8)                    self.G.nodes[0]["name"] = "node0"
281: (8)                    self.H.nodes[1]["name"] = "node1"
282: (4)                def test_edge_attr_dict(self):
283: (8)                    """Tests that the edge attribute dictionary of the two graphs is
284: (8)                    the same object.
285: (8)                    """
286: (8)                    for u, v in self.H.edges():
287: (12)                       assert self.G.edges[u, v] == self.H.edges[u, v]
288: (8)                    self.G.edges[0, 1]["name"] = "foo"
289: (8)                    assert self.G.edges[0, 1]["name"] == self.H.edges[0, 1]["name"]
290: (8)                    self.H.edges[3, 4]["name"] = "bar"
291: (8)                    assert self.G.edges[3, 4]["name"] == self.H.edges[3, 4]["name"]
292: (8)                    self.G.edges[0, 1]["name"] = "edge01"
293: (8)                    self.H.edges[3, 4]["name"] = "edge34"
294: (4)                def test_graph_attr_dict(self):
295: (8)                    """Tests that the graph attribute dictionary of the two graphs
296: (8)                    is the same object.
297: (8)                    """
298: (8)                    assert self.G.graph is self.H.graph
299: (4)                def test_readonly(self):
300: (8)                    """Tests that the subgraph cannot change the graph structure"""
301: (8)                    pytest.raises(nx.NetworkXError, self.H.add_node, 5)
302: (8)                    pytest.raises(nx.NetworkXError, self.H.remove_node, 0)
303: (8)                    pytest.raises(nx.NetworkXError, self.H.add_edge, 5, 6)
304: (8)                    pytest.raises(nx.NetworkXError, self.H.remove_edge, 0, 1)

----------------------------------------

File 415 - layout.py:

1: (0)              """
```

```
2: (0)              ******
3: (0)              Layout
4: (0)              ******
5: (0)              Node positioning algorithms for graph drawing.
6: (0)              For `random_layout()` the possible resulting shape
7: (0)              is a square of side [0, scale] (default: [0, 1])
8: (0)              Changing `center` shifts the layout by that amount.
9: (0)              For the other layout routines, the extent is
10: (0)             [center - scale, center + scale] (default: [-1, 1]).
11: (0)             Warning: Most layout routines have only been tested in 2-dimensions.
12: (0)             """
13: (0)             import networkx as nx
14: (0)             from networkx.utils import np_random_state
15: (0)             __all__ = [
16: (4)                 "bipartite_layout",
17: (4)                 "circular_layout",
18: (4)                 "kamada_kawai_layout",
19: (4)                 "random_layout",
20: (4)                 "rescale_layout",
21: (4)                 "rescale_layout_dict",
22: (4)                 "shell_layout",
23: (4)                 "spring_layout",
24: (4)                 "spectral_layout",
25: (4)                 "planar_layout",
26: (4)                 "fruchterman_reingold_layout",
27: (4)                 "spiral_layout",
28: (4)                 "multipartite_layout",
29: (4)                 "bfs_layout",
30: (4)                 "arf_layout",
31: (0)             ]
32: (0)             def _process_params(G, center, dim):
33: (4)                 import numpy as np
34: (4)                 if not isinstance(G, nx.Graph):
35: (8)                     empty_graph = nx.Graph()
36: (8)                     empty_graph.add_nodes_from(G)
37: (8)                     G = empty_graph
38: (4)                 if center is None:
39: (8)                     center = np.zeros(dim)
40: (4)                 else:
41: (8)                     center = np.asarray(center)
42: (4)                 if len(center) != dim:
43: (8)                     msg = "length of center coordinates must match dimension of layout"
44: (8)                     raise ValueError(msg)
45: (4)                 return G, center
46: (0)             @np_random_state(3)
47: (0)             def random_layout(G, center=None, dim=2, seed=None):
48: (4)                 """Position nodes uniformly at random in the unit square.
49: (4)                 For every node, a position is generated by choosing each of dim
50: (4)                 coordinates uniformly at random on the interval [0.0, 1.0).
51: (4)                 NumPy (http://scipy.org) is required for this function.
52: (4)                 Parameters
53: (4)                 ----------
54: (4)                 G : NetworkX graph or list of nodes
55: (8)                     A position will be assigned to every node in G.
56: (4)                 center : array-like or None
57: (8)                     Coordinate pair around which to center the layout.
58: (4)                 dim : int
59: (8)                     Dimension of layout.
60: (4)                 seed : int, RandomState instance or None  optional (default=None)
61: (8)                     Set the random state for deterministic node layouts.
62: (8)                     If int, `seed` is the seed used by the random number generator,
63: (8)                     if numpy.random.RandomState instance, `seed` is the random
64: (8)                     number generator,
65: (8)                     if None, the random number generator is the RandomState instance used
66: (8)                     by numpy.random.
67: (4)                 Returns
68: (4)                 -------
69: (4)                 pos : dict
70: (8)                     A dictionary of positions keyed by node
```

```
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.lollipop_graph(4, 3)
(4)                 >>> pos = nx.random_layout(G)
(4)                 """
(4)                 import numpy as np
(4)                 G, center = _process_params(G, center, dim)
(4)                 pos = seed.rand(len(G), dim) + center
(4)                 pos = pos.astype(np.float32)
(4)                 pos = dict(zip(G, pos))
(4)                 return pos
(0)             def circular_layout(G, scale=1, center=None, dim=2):
(4)                 """Position nodes on a circle.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph or list of nodes
(8)                     A position will be assigned to every node in G.
(4)                 scale : number (default: 1)
(8)                     Scale factor for positions.
(4)                 center : array-like or None
(8)                     Coordinate pair around which to center the layout.
(4)                 dim : int
(8)                     Dimension of layout.
(8)                     If dim>2, the remaining dimensions are set to zero
(8)                     in the returned positions.
(8)                     If dim<2, a ValueError is raised.
(4)                 Returns
(4)                 -------
(4)                 pos : dict
(8)                     A dictionary of positions keyed by node
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If dim < 2
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> pos = nx.circular_layout(G)
(4)                 Notes
(4)                 -----
(4)                 This algorithm currently only works in two dimensions and does not
(4)                 try to minimize edge crossings.
(4)                 """
(4)                 import numpy as np
(4)                 if dim < 2:
(8)                     raise ValueError("cannot handle dimensions < 2")
(4)                 G, center = _process_params(G, center, dim)
(4)                 paddims = max(0, (dim - 2))
(4)                 if len(G) == 0:
(8)                     pos = {}
(4)                 elif len(G) == 1:
(8)                     pos = {nx.utils.arbitrary_element(G): center}
(4)                 else:
(8)                     theta = np.linspace(0, 1, len(G) + 1)[:-1] * 2 * np.pi
(8)                     theta = theta.astype(np.float32)
(8)                     pos = np.column_stack(
(12)                        [np.cos(theta), np.sin(theta), np.zeros((len(G), paddims))]
(8)                     )
(8)                     pos = rescale_layout(pos, scale=scale) + center
(8)                     pos = dict(zip(G, pos))
(4)                 return pos
(0)             def shell_layout(G, nlist=None, rotate=None, scale=1, center=None, dim=2):
(4)                 """Position nodes in concentric circles.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph or list of nodes
(8)                     A position will be assigned to every node in G.
(4)                 nlist : list of lists
(7)                    List of node lists for each shell.
```

```
140: (4)                rotate : angle in radians (default=pi/len(nlist))
141: (7)                   Angle by which to rotate the starting position of each shell
142: (7)                   relative to the starting position of the previous shell.
143: (7)                   To recreate behavior before v2.5 use rotate=0.
144: (4)                scale : number (default: 1)
145: (8)                    Scale factor for positions.
146: (4)                center : array-like or None
147: (8)                    Coordinate pair around which to center the layout.
148: (4)                dim : int
149: (8)                    Dimension of layout, currently only dim=2 is supported.
150: (8)                    Other dimension values result in a ValueError.
151: (4)                Returns
152: (4)                -------
153: (4)                pos : dict
154: (8)                    A dictionary of positions keyed by node
155: (4)                Raises
156: (4)                ------
157: (4)                ValueError
158: (8)                    If dim != 2
159: (4)                Examples
160: (4)                --------
161: (4)                >>> G = nx.path_graph(4)
162: (4)                >>> shells = [[0], [1, 2, 3]]
163: (4)                >>> pos = nx.shell_layout(G, shells)
164: (4)                Notes
165: (4)                -----
166: (4)                This algorithm currently only works in two dimensions and does not
167: (4)                try to minimize edge crossings.
168: (4)                """
169: (4)                import numpy as np
170: (4)                if dim != 2:
171: (8)                    raise ValueError("can only handle 2 dimensions")
172: (4)                G, center = _process_params(G, center, dim)
173: (4)                if len(G) == 0:
174: (8)                    return {}
175: (4)                if len(G) == 1:
176: (8)                    return {nx.utils.arbitrary_element(G): center}
177: (4)                if nlist is None:
178: (8)                    nlist = [list(G)]
179: (4)                radius_bump = scale / len(nlist)
180: (4)                if len(nlist[0]) == 1:
181: (8)                    radius = 0.0
182: (4)                else:
183: (8)                    radius = radius_bump
184: (4)                if rotate is None:
185: (8)                    rotate = np.pi / len(nlist)
186: (4)                first_theta = rotate
187: (4)                npos = {}
188: (4)                for nodes in nlist:
189: (8)                    theta = (
190: (12)                       np.linspace(0, 2 * np.pi, len(nodes), endpoint=False,
dtype=np.float32)
191: (12)                       + first_theta
192: (8)                    )
193: (8)                    pos = radius * np.column_stack([np.cos(theta), np.sin(theta)]) +
center
194: (8)                    npos.update(zip(nodes, pos))
195: (8)                    radius += radius_bump
196: (8)                    first_theta += rotate
197: (4)                return npos
198: (0)            def bipartite_layout(
199: (4)                G, nodes, align="vertical", scale=1, center=None, aspect_ratio=4 / 3
200: (0)            ):
201: (4)                """Position nodes in two straight lines.
202: (4)                Parameters
203: (4)                ----------
204: (4)                G : NetworkX graph or list of nodes
205: (8)                    A position will be assigned to every node in G.
206: (4)                nodes : list or container
```

```
207: (8)                    Nodes in one node set of the bipartite graph.
208: (8)                    This set will be placed on left or top.
209: (4)                align : string (default='vertical')
210: (8)                    The alignment of nodes. Vertical or horizontal.
211: (4)                scale : number (default: 1)
212: (8)                    Scale factor for positions.
213: (4)                center : array-like or None
214: (8)                    Coordinate pair around which to center the layout.
215: (4)                aspect_ratio : number (default=4/3):
216: (8)                    The ratio of the width to the height of the layout.
217: (4)                Returns
218: (4)                -------
219: (4)                pos : dict
220: (8)                    A dictionary of positions keyed by node.
221: (4)                Examples
222: (4)                --------
223: (4)                >>> G = nx.bipartite.gnmk_random_graph(3, 5, 10, seed=123)
224: (4)                >>> top = nx.bipartite.sets(G)[0]
225: (4)                >>> pos = nx.bipartite_layout(G, top)
226: (4)                Notes
227: (4)                -----
228: (4)                This algorithm currently only works in two dimensions and does not
229: (4)                try to minimize edge crossings.
230: (4)                """
231: (4)                import numpy as np
232: (4)                if align not in ("vertical", "horizontal"):
233: (8)                    msg = "align must be either vertical or horizontal."
234: (8)                    raise ValueError(msg)
235: (4)                G, center = _process_params(G, center=center, dim=2)
236: (4)                if len(G) == 0:
237: (8)                    return {}
238: (4)                height = 1
239: (4)                width = aspect_ratio * height
240: (4)                offset = (width / 2, height / 2)
241: (4)                top = dict.fromkeys(nodes)
242: (4)                bottom = [v for v in G if v not in top]
243: (4)                nodes = list(top) + bottom
244: (4)                left_xs = np.repeat(0, len(top))
245: (4)                right_xs = np.repeat(width, len(bottom))
246: (4)                left_ys = np.linspace(0, height, len(top))
247: (4)                right_ys = np.linspace(0, height, len(bottom))
248: (4)                top_pos = np.column_stack([left_xs, left_ys]) - offset
249: (4)                bottom_pos = np.column_stack([right_xs, right_ys]) - offset
250: (4)                pos = np.concatenate([top_pos, bottom_pos])
251: (4)                pos = rescale_layout(pos, scale=scale) + center
252: (4)                if align == "horizontal":
253: (8)                    pos = pos[:, ::-1]  # swap x and y coords
254: (4)                pos = dict(zip(nodes, pos))
255: (4)                return pos
256: (0)            @np_random_state(10)
257: (0)            def spring_layout(
258: (4)                G,
259: (4)                k=None,
260: (4)                pos=None,
261: (4)                fixed=None,
262: (4)                iterations=50,
263: (4)                threshold=1e-4,
264: (4)                weight="weight",
265: (4)                scale=1,
266: (4)                center=None,
267: (4)                dim=2,
268: (4)                seed=None,
269: (0)            ):
270: (4)                """Position nodes using Fruchterman-Reingold force-directed algorithm.
271: (4)                The algorithm simulates a force-directed representation of the network
272: (4)                treating edges as springs holding nodes close, while treating nodes
273: (4)                as repelling objects, sometimes called an anti-gravity force.
274: (4)                Simulation continues until the positions are close to an equilibrium.
275: (4)                There are some hard-coded values: minimal distance between
```

```
    nodes (0.01) and "temperature" of 0.1 to ensure nodes don't fly away.
    During the simulation, `k` helps determine the distance between nodes,
    though `scale` and `center` determine the size and place after
    rescaling occurs at the end of the simulation.
    Fixing some nodes doesn't allow them to move in the simulation.
    It also turns off the rescaling feature at the simulation's end.
    In addition, setting `scale` to `None` turns off rescaling.
    Parameters
    ----------
    G : NetworkX graph or list of nodes
        A position will be assigned to every node in G.
    k : float (default=None)
        Optimal distance between nodes.  If None the distance is set to
        1/sqrt(n) where n is the number of nodes.  Increase this value
        to move nodes farther apart.
    pos : dict or None  optional (default=None)
        Initial positions for nodes as a dictionary with node as keys
        and values as a coordinate list or tuple.  If None, then use
        random initial positions.
    fixed : list or None  optional (default=None)
        Nodes to keep fixed at initial position.
        Nodes not in ``G.nodes`` are ignored.
        ValueError raised if `fixed` specified and `pos` not.
    iterations : int  optional (default=50)
        Maximum number of iterations taken
    threshold: float optional (default = 1e-4)
        Threshold for relative error in node position changes.
        The iteration stops if the error is below this threshold.
    weight : string or None   optional (default='weight')
        The edge attribute that holds the numerical value used for
        the edge weight.  Larger means a stronger attractive force.
        If None, then all edge weights are 1.
    scale : number or None (default: 1)
        Scale factor for positions. Not used unless `fixed is None`.
        If scale is None, no rescaling is performed.
    center : array-like or None
        Coordinate pair around which to center the layout.
        Not used unless `fixed is None`.
    dim : int
        Dimension of layout.
    seed : int, RandomState instance or None  optional (default=None)
        Set the random state for deterministic node layouts.
        If int, `seed` is the seed used by the random number generator,
        if numpy.random.RandomState instance, `seed` is the random
        number generator,
        if None, the random number generator is the RandomState instance used
        by numpy.random.
    Returns
    -------
    pos : dict
        A dictionary of positions keyed by node
    Examples
    --------
    >>> G = nx.path_graph(4)
    >>> pos = nx.spring_layout(G)
    >>> pos = nx.fruchterman_reingold_layout(G)
    """
    import numpy as np
    G, center = _process_params(G, center, dim)
    if fixed is not None:
        if pos is None:
            raise ValueError("nodes are fixed without positions given")
        for node in fixed:
            if node not in pos:
                raise ValueError("nodes are fixed without positions given")
        nfixed = {node: i for i, node in enumerate(G)}
        fixed = np.asarray([nfixed[node] for node in fixed if node in nfixed])
    if pos is not None:
        dom_size = max(coord for pos_tup in pos.values() for coord in pos_tup)
```

```
345: (8)                    if dom_size == 0:
346: (12)                       dom_size = 1
347: (8)                    pos_arr = seed.rand(len(G), dim) * dom_size + center
348: (8)                    for i, n in enumerate(G):
349: (12)                       if n in pos:
350: (16)                           pos_arr[i] = np.asarray(pos[n])
351: (4)                else:
352: (8)                    pos_arr = None
353: (8)                    dom_size = 1
354: (4)                if len(G) == 0:
355: (8)                    return {}
356: (4)                if len(G) == 1:
357: (8)                    return {nx.utils.arbitrary_element(G.nodes()): center}
358: (4)                try:
359: (8)                    if len(G) < 500:  # sparse solver for large graphs
360: (12)                       raise ValueError
361: (8)                    A = nx.to_scipy_sparse_array(G, weight=weight, dtype="f")
362: (8)                    if k is None and fixed is not None:
363: (12)                       nnodes, _ = A.shape
364: (12)                       k = dom_size / np.sqrt(nnodes)
365: (8)                    pos = _sparse_fruchterman_reingold(
366: (12)                       A, k, pos_arr, fixed, iterations, threshold, dim, seed
367: (8)                    )
368: (4)                except ValueError:
369: (8)                    A = nx.to_numpy_array(G, weight=weight)
370: (8)                    if k is None and fixed is not None:
371: (12)                       nnodes, _ = A.shape
372: (12)                       k = dom_size / np.sqrt(nnodes)
373: (8)                    pos = _fruchterman_reingold(
374: (12)                       A, k, pos_arr, fixed, iterations, threshold, dim, seed
375: (8)                    )
376: (4)                if fixed is None and scale is not None:
377: (8)                    pos = rescale_layout(pos, scale=scale) + center
378: (4)                pos = dict(zip(G, pos))
379: (4)                return pos
380: (0)            fruchterman_reingold_layout = spring_layout
381: (0)            @np_random_state(7)
382: (0)            def _fruchterman_reingold(
383: (4)                A, k=None, pos=None, fixed=None, iterations=50, threshold=1e-4, dim=2,
seed=None
384: (0)            ):
385: (4)                import numpy as np
386: (4)                try:
387: (8)                    nnodes, _ = A.shape
388: (4)                except AttributeError as err:
389: (8)                    msg = "fruchterman_reingold() takes an adjacency matrix as input"
390: (8)                    raise nx.NetworkXError(msg) from err
391: (4)                if pos is None:
392: (8)                    pos = np.asarray(seed.rand(nnodes, dim), dtype=A.dtype)
393: (4)                else:
394: (8)                    pos = pos.astype(A.dtype)
395: (4)                if k is None:
396: (8)                    k = np.sqrt(1.0 / nnodes)
397: (4)                t = max(max(pos.T[0]) - min(pos.T[0]), max(pos.T[1]) - min(pos.T[1])) *
0.1
398: (4)                dt = t / (iterations + 1)
399: (4)                delta = np.zeros((pos.shape[0], pos.shape[0], pos.shape[1]),
dtype=A.dtype)
400: (4)                for iteration in range(iterations):
401: (8)                    delta = pos[:, np.newaxis, :] - pos[np.newaxis, :, :]
402: (8)                    distance = np.linalg.norm(delta, axis=-1)
403: (8)                    np.clip(distance, 0.01, None, out=distance)
404: (8)                    displacement = np.einsum(
405: (12)                       "ijk,ij->ik", delta, (k * k / distance**2 - A * distance / k)
406: (8)                    )
407: (8)                    length = np.linalg.norm(displacement, axis=-1)
408: (8)                    length = np.where(length < 0.01, 0.1, length)
409: (8)                    delta_pos = np.einsum("ij,i->ij", displacement, t / length)
410: (8)                    if fixed is not None:
```

```
411: (12)                        delta_pos[fixed] = 0.0
412: (8)                    pos += delta_pos
413: (8)                    t -= dt
414: (8)                    if (np.linalg.norm(delta_pos) / nnodes) < threshold:
415: (12)                        break
416: (4)                return pos
417: (0)            @np_random_state(7)
418: (0)            def _sparse_fruchterman_reingold(
419: (4)                A, k=None, pos=None, fixed=None, iterations=50, threshold=1e-4, dim=2,
seed=None
420: (0)            ):
421: (4)                import numpy as np
422: (4)                import scipy as sp
423: (4)                try:
424: (8)                    nnodes, _ = A.shape
425: (4)                except AttributeError as err:
426: (8)                    msg = "fruchterman_reingold() takes an adjacency matrix as input"
427: (8)                    raise nx.NetworkXError(msg) from err
428: (4)                try:
429: (8)                    A = A.tolil()
430: (4)                except AttributeError:
431: (8)                    A = (sp.sparse.coo_array(A)).tolil()
432: (4)                if pos is None:
433: (8)                    pos = np.asarray(seed.rand(nnodes, dim), dtype=A.dtype)
434: (4)                else:
435: (8)                    pos = pos.astype(A.dtype)
436: (4)                if fixed is None:
437: (8)                    fixed = []
438: (4)                if k is None:
439: (8)                    k = np.sqrt(1.0 / nnodes)
440: (4)                t = max(max(pos.T[0]) - min(pos.T[0]), max(pos.T[1]) - min(pos.T[1])) *
0.1
441: (4)                dt = t / (iterations + 1)
442: (4)                displacement = np.zeros((dim, nnodes))
443: (4)                for iteration in range(iterations):
444: (8)                    displacement *= 0
445: (8)                    for i in range(A.shape[0]):
446: (12)                        if i in fixed:
447: (16)                            continue
448: (12)                        delta = (pos[i] - pos).T
449: (12)                        distance = np.sqrt((delta**2).sum(axis=0))
450: (12)                        distance = np.where(distance < 0.01, 0.01, distance)
451: (12)                        Ai = A.getrowview(i).toarray()  # TODO: revisit w/ sparse 1D
container
452: (12)                        displacement[:, i] += (
453: (16)                            delta * (k * k / distance**2 - Ai * distance / k)
454: (12)                        ).sum(axis=1)
455: (8)                    length = np.sqrt((displacement**2).sum(axis=0))
456: (8)                    length = np.where(length < 0.01, 0.1, length)
457: (8)                    delta_pos = (displacement * t / length).T
458: (8)                    pos += delta_pos
459: (8)                    t -= dt
460: (8)                    if (np.linalg.norm(delta_pos) / nnodes) < threshold:
461: (12)                        break
462: (4)                return pos
463: (0)            def kamada_kawai_layout(
464: (4)                G, dist=None, pos=None, weight="weight", scale=1, center=None, dim=2
465: (0)            ):
466: (4)                """Position nodes using Kamada-Kawai path-length cost-function.
467: (4)                Parameters
468: (4)                ----------
469: (4)                G : NetworkX graph or list of nodes
470: (8)                    A position will be assigned to every node in G.
471: (4)                dist : dict (default=None)
472: (8)                    A two-level dictionary of optimal distances between nodes,
473: (8)                    indexed by source and destination node.
474: (8)                    If None, the distance is computed using shortest_path_length().
475: (4)                pos : dict or None  optional (default=None)
476: (8)                    Initial positions for nodes as a dictionary with node as keys
```

```
477: (8)                    and values as a coordinate list or tuple.  If None, then use
478: (8)                    circular_layout() for dim >= 2 and a linear layout for dim == 1.
479: (4)                weight : string or None   optional (default='weight')
480: (8)                    The edge attribute that holds the numerical value used for
481: (8)                    the edge weight.  If None, then all edge weights are 1.
482: (4)                scale : number (default: 1)
483: (8)                    Scale factor for positions.
484: (4)                center : array-like or None
485: (8)                    Coordinate pair around which to center the layout.
486: (4)                dim : int
487: (8)                    Dimension of layout.
488: (4)                Returns
489: (4)                -------
490: (4)                pos : dict
491: (8)                    A dictionary of positions keyed by node
492: (4)                Examples
493: (4)                --------
494: (4)                >>> G = nx.path_graph(4)
495: (4)                >>> pos = nx.kamada_kawai_layout(G)
496: (4)                """
497: (4)                import numpy as np
498: (4)                G, center = _process_params(G, center, dim)
499: (4)                nNodes = len(G)
500: (4)                if nNodes == 0:
501: (8)                    return {}
502: (4)                if dist is None:
503: (8)                    dist = dict(nx.shortest_path_length(G, weight=weight))
504: (4)                dist_mtx = 1e6 * np.ones((nNodes, nNodes))
505: (4)                for row, nr in enumerate(G):
506: (8)                    if nr not in dist:
507: (12)                       continue
508: (8)                    rdist = dist[nr]
509: (8)                    for col, nc in enumerate(G):
510: (12)                       if nc not in rdist:
511: (16)                           continue
512: (12)                       dist_mtx[row][col] = rdist[nc]
513: (4)                if pos is None:
514: (8)                    if dim >= 3:
515: (12)                       pos = random_layout(G, dim=dim)
516: (8)                    elif dim == 2:
517: (12)                       pos = circular_layout(G, dim=dim)
518: (8)                    else:
519: (12)                       pos = dict(zip(G, np.linspace(0, 1, len(G))))
520: (4)                pos_arr = np.array([pos[n] for n in G])
521: (4)                pos = _kamada_kawai_solve(dist_mtx, pos_arr, dim)
522: (4)                pos = rescale_layout(pos, scale=scale) + center
523: (4)                return dict(zip(G, pos))
524: (0)            def _kamada_kawai_solve(dist_mtx, pos_arr, dim):
525: (4)                import numpy as np
526: (4)                import scipy as sp
527: (4)                meanwt = 1e-3
528: (4)                costargs = (np, 1 / (dist_mtx + np.eye(dist_mtx.shape[0]) * 1e-3), meanwt,
dim)
529: (4)                optresult = sp.optimize.minimize(
530: (8)                    _kamada_kawai_costfn,
531: (8)                    pos_arr.ravel(),
532: (8)                    method="L-BFGS-B",
533: (8)                    args=costargs,
534: (8)                    jac=True,
535: (4)                )
536: (4)                return optresult.x.reshape((-1, dim))
537: (0)            def _kamada_kawai_costfn(pos_vec, np, invdist, meanweight, dim):
538: (4)                nNodes = invdist.shape[0]
539: (4)                pos_arr = pos_vec.reshape((nNodes, dim))
540: (4)                delta = pos_arr[:, np.newaxis, :] - pos_arr[np.newaxis, :, :]
541: (4)                nodesep = np.linalg.norm(delta, axis=-1)
542: (4)                direction = np.einsum("ijk,ij->ijk", delta, 1 / (nodesep + np.eye(nNodes)
* 1e-3))
543: (4)                offset = nodesep * invdist - 1.0
```

```
(4)                 offset[np.diag_indices(nNodes)] = 0
(4)                 cost = 0.5 * np.sum(offset**2)
(4)                 grad = np.einsum("ij,ij,ijk->ik", invdist, offset, direction) - np.einsum(
(8)                     "ij,ij,ijk->jk", invdist, offset, direction
(4)                 )
(4)                 sumpos = np.sum(pos_arr, axis=0)
(4)                 cost += 0.5 * meanweight * np.sum(sumpos**2)
(4)                 grad += meanweight * sumpos
(4)                 return (cost, grad.ravel())
(0)             def spectral_layout(G, weight="weight", scale=1, center=None, dim=2):
(4)                 """Position nodes using the eigenvectors of the graph Laplacian.
(4)                 Using the unnormalized Laplacian, the layout shows possible clusters of
(4)                 nodes which are an approximation of the ratio cut. If dim is the number of
(4)                 dimensions then the positions are the entries of the dim eigenvectors
(4)                 corresponding to the ascending eigenvalues starting from the second one.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph or list of nodes
(8)                     A position will be assigned to every node in G.
(4)                 weight : string or None   optional (default='weight')
(8)                     The edge attribute that holds the numerical value used for
(8)                     the edge weight.  If None, then all edge weights are 1.
(4)                 scale : number (default: 1)
(8)                     Scale factor for positions.
(4)                 center : array-like or None
(8)                     Coordinate pair around which to center the layout.
(4)                 dim : int
(8)                     Dimension of layout.
(4)                 Returns
(4)                 -------
(4)                 pos : dict
(8)                     A dictionary of positions keyed by node
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> pos = nx.spectral_layout(G)
(4)                 Notes
(4)                 -----
(4)                 Directed graphs will be considered as undirected graphs when
(4)                 positioning the nodes.
(4)                 For larger graphs (>500 nodes) this will use the SciPy sparse
(4)                 eigenvalue solver (ARPACK).
(4)                 """
(4)                 import numpy as np
(4)                 G, center = _process_params(G, center, dim)
(4)                 if len(G) <= 2:
(8)                     if len(G) == 0:
(12)                        pos = np.array([])
(8)                     elif len(G) == 1:
(12)                        pos = np.array([center])
(8)                     else:
(12)                        pos = np.array([np.zeros(dim), np.array(center) * 2.0])
(8)                     return dict(zip(G, pos))
(4)                 try:
(8)                     if len(G) < 500:  # dense solver is faster for small graphs
(12)                        raise ValueError
(8)                     A = nx.to_scipy_sparse_array(G, weight=weight, dtype="d")
(8)                     if G.is_directed():
(12)                        A = A + np.transpose(A)
(8)                     pos = _sparse_spectral(A, dim)
(4)                 except (ImportError, ValueError):
(8)                     A = nx.to_numpy_array(G, weight=weight)
(8)                     if G.is_directed():
(12)                        A += A.T
(8)                     pos = _spectral(A, dim)
(4)                 pos = rescale_layout(pos, scale=scale) + center
(4)                 pos = dict(zip(G, pos))
(4)                 return pos
(0)             def _spectral(A, dim=2):
```

```
613: (4)                import numpy as np
614: (4)                try:
615: (8)                    nnodes, _ = A.shape
616: (4)                except AttributeError as err:
617: (8)                    msg = "spectral() takes an adjacency matrix as input"
618: (8)                    raise nx.NetworkXError(msg) from err
619: (4)                D = np.identity(nnodes, dtype=A.dtype) * np.sum(A, axis=1)
620: (4)                L = D - A
621: (4)                eigenvalues, eigenvectors = np.linalg.eig(L)
622: (4)                index = np.argsort(eigenvalues)[1 : dim + 1]  # 0 index is zero eigenvalue
623: (4)                return np.real(eigenvectors[:, index])
624: (0)            def _sparse_spectral(A, dim=2):
625: (4)                import numpy as np
626: (4)                import scipy as sp
627: (4)                try:
628: (8)                    nnodes, _ = A.shape
629: (4)                except AttributeError as err:
630: (8)                    msg = "sparse_spectral() takes an adjacency matrix as input"
631: (8)                    raise nx.NetworkXError(msg) from err
632: (4)                D = sp.sparse.csr_array(sp.sparse.spdiags(A.sum(axis=1), 0, nnodes,
nnodes))
633: (4)                L = D - A
634: (4)                k = dim + 1
635: (4)                ncv = max(2 * k + 1, int(np.sqrt(nnodes)))
636: (4)                eigenvalues, eigenvectors = sp.sparse.linalg.eigsh(L, k, which="SM",
ncv=ncv)
637: (4)                index = np.argsort(eigenvalues)[1:k]  # 0 index is zero eigenvalue
638: (4)                return np.real(eigenvectors[:, index])
639: (0)            def planar_layout(G, scale=1, center=None, dim=2):
640: (4)                """Position nodes without edge intersections.
641: (4)                Parameters
642: (4)                ----------
643: (4)                G : NetworkX graph or list of nodes
644: (8)                    A position will be assigned to every node in G. If G is of type
645: (8)                    nx.PlanarEmbedding, the positions are selected accordingly.
646: (4)                scale : number (default: 1)
647: (8)                    Scale factor for positions.
648: (4)                center : array-like or None
649: (8)                    Coordinate pair around which to center the layout.
650: (4)                dim : int
651: (8)                    Dimension of layout.
652: (4)                Returns
653: (4)                -------
654: (4)                pos : dict
655: (8)                    A dictionary of positions keyed by node
656: (4)                Raises
657: (4)                ------
658: (4)                NetworkXException
659: (8)                    If G is not planar
660: (4)                Examples
661: (4)                --------
662: (4)                >>> G = nx.path_graph(4)
663: (4)                >>> pos = nx.planar_layout(G)
664: (4)                """
665: (4)                import numpy as np
666: (4)                if dim != 2:
667: (8)                    raise ValueError("can only handle 2 dimensions")
668: (4)                G, center = _process_params(G, center, dim)
669: (4)                if len(G) == 0:
670: (8)                    return {}
671: (4)                if isinstance(G, nx.PlanarEmbedding):
672: (8)                    embedding = G
673: (4)                else:
674: (8)                    is_planar, embedding = nx.check_planarity(G)
675: (8)                    if not is_planar:
676: (12)                       raise nx.NetworkXException("G is not planar.")
677: (4)                pos = nx.combinatorial_embedding_to_pos(embedding)
678: (4)                node_list = list(embedding)
679: (4)                pos = np.vstack([pos[x] for x in node_list])
```

```
(4)                 pos = pos.astype(np.float64)
(4)                 pos = rescale_layout(pos, scale=scale) + center
(4)                 return dict(zip(node_list, pos))
(0)             def spiral_layout(G, scale=1, center=None, dim=2, resolution=0.35,
equidistant=False):
(4)                 """Position nodes in a spiral layout.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph or list of nodes
(8)                     A position will be assigned to every node in G.
(4)                 scale : number (default: 1)
(8)                     Scale factor for positions.
(4)                 center : array-like or None
(8)                     Coordinate pair around which to center the layout.
(4)                 dim : int, default=2
(8)                     Dimension of layout, currently only dim=2 is supported.
(8)                     Other dimension values result in a ValueError.
(4)                 resolution : float, default=0.35
(8)                     The compactness of the spiral layout returned.
(8)                     Lower values result in more compressed spiral layouts.
(4)                 equidistant : bool, default=False
(8)                     If True, nodes will be positioned equidistant from each other
(8)                     by decreasing angle further from center.
(8)                     If False, nodes will be positioned at equal angles
(8)                     from each other by increasing separation further from center.
(4)                 Returns
(4)                 -------
(4)                 pos : dict
(8)                     A dictionary of positions keyed by node
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If dim != 2
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> pos = nx.spiral_layout(G)
(4)                 >>> nx.draw(G, pos=pos)
(4)                 Notes
(4)                 -----
(4)                 This algorithm currently only works in two dimensions.
(4)                 """
(4)                 import numpy as np
(4)                 if dim != 2:
(8)                     raise ValueError("can only handle 2 dimensions")
(4)                 G, center = _process_params(G, center, dim)
(4)                 if len(G) == 0:
(8)                     return {}
(4)                 if len(G) == 1:
(8)                     return {nx.utils.arbitrary_element(G): center}
(4)                 pos = []
(4)                 if equidistant:
(8)                     chord = 1
(8)                     step = 0.5
(8)                     theta = resolution
(8)                     theta += chord / (step * theta)
(8)                     for _ in range(len(G)):
(12)                        r = step * theta
(12)                        theta += chord / r
(12)                        pos.append([np.cos(theta) * r, np.sin(theta) * r])
(4)                 else:
(8)                     dist = np.arange(len(G), dtype=float)
(8)                     angle = resolution * dist
(8)                     pos = np.transpose(dist * np.array([np.cos(angle), np.sin(angle)]))
(4)                 pos = rescale_layout(np.array(pos), scale=scale) + center
(4)                 pos = dict(zip(G, pos))
(4)                 return pos
(0)             def multipartite_layout(G, subset_key="subset", align="vertical", scale=1,
center=None):
```

```
    """Position nodes in layers of straight lines.
    Parameters
    ----------
    G : NetworkX graph or list of nodes
        A position will be assigned to every node in G.
    subset_key : string or dict (default='subset')
        If a string, the key of node data in G that holds the node subset.
        If a dict, keyed by layer number to the nodes in that layer/subset.
    align : string (default='vertical')
        The alignment of nodes. Vertical or horizontal.
    scale : number (default: 1)
        Scale factor for positions.
    center : array-like or None
        Coordinate pair around which to center the layout.
    Returns
    -------
    pos : dict
        A dictionary of positions keyed by node.
    Examples
    --------
    >>> G = nx.complete_multipartite_graph(28, 16, 10)
    >>> pos = nx.multipartite_layout(G)
    or use a dict to provide the layers of the layout
    >>> G = nx.Graph([(0, 1), (1, 2), (1, 3), (3, 4)])
    >>> layers = {"a": [0], "b": [1], "c": [2, 3], "d": [4]}
    >>> pos = nx.multipartite_layout(G, subset_key=layers)
    Notes
    -----
    This algorithm currently only works in two dimensions and does not
    try to minimize edge crossings.
    Network does not need to be a complete multipartite graph. As long as nodes
    have subset_key data, they will be placed in the corresponding layers.
    """
    import numpy as np
    if align not in ("vertical", "horizontal"):
        msg = "align must be either vertical or horizontal."
        raise ValueError(msg)
    G, center = _process_params(G, center=center, dim=2)
    if len(G) == 0:
        return {}
    try:
        if len(G) != sum(len(nodes) for nodes in subset_key.values()):
            raise nx.NetworkXError(
                "all nodes must be in one subset of `subset_key` dict"
            )
    except AttributeError:
        node_to_subset = nx.get_node_attributes(G, subset_key)
        if len(node_to_subset) != len(G):
            raise nx.NetworkXError(
                f"all nodes need a subset_key attribute: {subset_key}"
            )
        subset_key = nx.utils.groups(node_to_subset)
    try:
        layers = dict(sorted(subset_key.items()))
    except TypeError:
        layers = subset_key
    pos = None
    nodes = []
    width = len(layers)
    for i, layer in enumerate(layers.values()):
        height = len(layer)
        xs = np.repeat(i, height)
        ys = np.arange(0, height, dtype=float)
        offset = ((width - 1) / 2, (height - 1) / 2)
        layer_pos = np.column_stack([xs, ys]) - offset
        if pos is None:
            pos = layer_pos
        else:
```

```
815: (12)                        pos = np.concatenate([pos, layer_pos])
816: (8)                    nodes.extend(layer)
817: (4)                pos = rescale_layout(pos, scale=scale) + center
818: (4)                if align == "horizontal":
819: (8)                    pos = pos[:, ::-1]  # swap x and y coords
820: (4)                pos = dict(zip(nodes, pos))
821: (4)                return pos
822: (0)            def arf_layout(
823: (4)                G,
824: (4)                pos=None,
825: (4)                scaling=1,
826: (4)                a=1.1,
827: (4)                etol=1e-6,
828: (4)                dt=1e-3,
829: (4)                max_iter=1000,
830: (0)            ):
831: (4)                """Arf layout for networkx
832: (4)                The attractive and repulsive forces (arf) layout [1]
833: (4)                improves the spring layout in three ways. First, it
834: (4)                prevents congestion of highly connected nodes due to
835: (4)                strong forcing between nodes. Second, it utilizes the
836: (4)                layout space more effectively by preventing large gaps
837: (4)                that spring layout tends to create. Lastly, the arf
838: (4)                layout represents symmetries in the layout better than
839: (4)                the default spring layout.
840: (4)                Parameters
841: (4)                ----------
842: (4)                G : nx.Graph or nx.DiGraph
843: (8)                    Networkx graph.
844: (4)                pos : dict
845: (8)                    Initial  position of  the nodes.  If set  to None  a
846: (8)                    random layout will be used.
847: (4)                scaling : float
848: (8)                    Scales the radius of the circular layout space.
849: (4)                a : float
850: (8)                    Strength of springs between connected nodes. Should be larger than 1.
The greater a, the clearer the separation ofunconnected sub clusters.
851: (4)                etol : float
852: (8)                    Gradient sum of spring forces must be larger than `etol` before
successful termination.
853: (4)                dt : float
854: (8)                    Time step for force differential equation simulations.
855: (4)                max_iter : int
856: (8)                    Max iterations before termination of the algorithm.
857: (4)                References
858: (4)                .. [1] "Self-Organization Applied to Dynamic Network Layout", M. Geipel,
859: (12)                       International Journal of Modern Physics C, 2007, Vol 18, No 10,
pp. 1537-1549.
860: (12)                       https://doi.org/10.1142/S0129183107011558
https://arxiv.org/abs/0704.1748
861: (4)                Returns
862: (4)                -------
863: (4)                pos : dict
864: (8)                    A dictionary of positions keyed by node.
865: (4)                Examples
866: (4)                --------
867: (4)                >>> G = nx.grid_graph((5, 5))
868: (4)                >>> pos = nx.arf_layout(G)
869: (4)                """
870: (4)                import warnings
871: (4)                import numpy as np
872: (4)                if a <= 1:
873: (8)                    msg = "The parameter a should be larger than 1"
874: (8)                    raise ValueError(msg)
875: (4)                pos_tmp = nx.random_layout(G)
876: (4)                if pos is None:
877: (8)                    pos = pos_tmp
878: (4)                else:
879: (8)                    for node in G.nodes():
```

```
(12)                        if node not in pos:
(16)                            pos[node] = pos_tmp[node].copy()
(4)                 N = len(G)
(4)                 if N == 0:
(8)                     return pos
(4)                 K = np.ones((N, N)) - np.eye(N)
(4)                 node_order = {node: i for i, node in enumerate(G)}
(4)                 for x, y in G.edges():
(8)                     if x != y:
(12)                        idx, jdx = (node_order[i] for i in (x, y))
(12)                        K[idx, jdx] = a
(4)                 p = np.asarray(list(pos.values()))
(4)                 rho = scaling * np.sqrt(N)
(4)                 error = etol + 1
(4)                 n_iter = 0
(4)                 while error > etol:
(8)                     diff = p[:, np.newaxis] - p[np.newaxis]
(8)                     A = np.linalg.norm(diff, axis=-1)[..., np.newaxis]
(8)                     with warnings.catch_warnings():
(12)                        warnings.simplefilter("ignore")
(12)                        change = K[..., np.newaxis] * diff - rho / A * diff
(8)                     change = np.nansum(change, axis=0)
(8)                     p += change * dt
(8)                     error = np.linalg.norm(change, axis=-1).sum()
(8)                     if n_iter > max_iter:
(12)                        break
(8)                     n_iter += 1
(4)                 return dict(zip(G.nodes(), p))
(0)             def rescale_layout(pos, scale=1):
(4)                 """Returns scaled position array to (-scale, scale) in all axes.
(4)                 The function acts on NumPy arrays which hold position information.
(4)                 Each position is one row of the array. The dimension of the space
(4)                 equals the number of columns. Each coordinate in one column.
(4)                 To rescale, the mean (center) is subtracted from each axis separately.
(4)                 Then all values are scaled so that the largest magnitude value
(4)                 from all axes equals `scale` (thus, the aspect ratio is preserved).
(4)                 The resulting NumPy Array is returned (order of rows unchanged).
(4)                 Parameters
(4)                 ----------
(4)                 pos : numpy array
(8)                     positions to be scaled. Each row is a position.
(4)                 scale : number (default: 1)
(8)                     The size of the resulting extent in all directions.
(4)                 Returns
(4)                 -------
(4)                 pos : numpy array
(8)                     scaled positions. Each row is a position.
(4)                 See Also
(4)                 --------
(4)                 rescale_layout_dict
(4)                 """
(4)                 import numpy as np
(4)                 pos -= pos.mean(axis=0)
(4)                 lim = np.abs(pos).max()  # max coordinate for all axes
(4)                 if lim > 0:
(8)                     pos *= scale / lim
(4)                 return pos
(0)             def rescale_layout_dict(pos, scale=1):
(4)                 """Return a dictionary of scaled positions keyed by node
(4)                 Parameters
(4)                 ----------
(4)                 pos : A dictionary of positions keyed by node
(4)                 scale : number (default: 1)
(8)                     The size of the resulting extent in all directions.
(4)                 Returns
(4)                 -------
(4)                 pos : A dictionary of positions keyed by node
(4)                 Examples
(4)                 --------
```

```
949: (4)                >>> import numpy as np
950: (4)                >>> pos = {0: np.array((0, 0)), 1: np.array((1, 1)), 2: np.array((0.5,
0.5))}
951: (4)                >>> nx.rescale_layout_dict(pos)
952: (4)                {0: array([-1., -1.]), 1: array([1., 1.]), 2: array([0., 0.])}
953: (4)                >>> pos = {0: np.array((0, 0)), 1: np.array((-1, 1)), 2: np.array((-0.5,
0.5))}
954: (4)                >>> nx.rescale_layout_dict(pos, scale=2)
955: (4)                {0: array([ 2., -2.]), 1: array([-2.,  2.]), 2: array([0., 0.])}
956: (4)                See Also
957: (4)                --------
958: (4)                rescale_layout
959: (4)                """
960: (4)                import numpy as np
961: (4)                if not pos:  # empty_graph
962: (8)                    return {}
963: (4)                pos_v = np.array(list(pos.values()))
964: (4)                pos_v = rescale_layout(pos_v, scale=scale)
965: (4)                return dict(zip(pos, pos_v))
966: (0)            def bfs_layout(G, start, *, align="vertical", scale=1, center=None):
967: (4)                """Position nodes according to breadth-first search algorithm.
968: (4)                Parameters
969: (4)                ----------
970: (4)                G : NetworkX graph
971: (8)                    A position will be assigned to every node in G.
972: (4)                start : node in `G`
973: (8)                    Starting node for bfs
974: (4)                center : array-like or None
975: (8)                    Coordinate pair around which to center the layout.
976: (4)                Returns
977: (4)                -------
978: (4)                pos : dict
979: (8)                    A dictionary of positions keyed by node.
980: (4)                Examples
981: (4)                --------
982: (4)                >>> G = nx.path_graph(4)
983: (4)                >>> pos = nx.bfs_layout(G, 0)
984: (4)                Notes
985: (4)                -----
986: (4)                This algorithm currently only works in two dimensions and does not
987: (4)                try to minimize edge crossings.
988: (4)                """
989: (4)                G, center = _process_params(G, center, 2)
990: (4)                layers = dict(enumerate(nx.bfs_layers(G, start)))
991: (4)                if len(G) != sum(len(nodes) for nodes in layers.values()):
992: (8)                    raise nx.NetworkXError(
993: (12)                       "bfs_layout didn't include all nodes. Perhaps use input graph:\n"
994: (12)                       "        G.subgraph(nx.node_connected_component(G, start))"
995: (8)                    )
996: (4)                return multipartite_layout(
997: (8)                    G, subset_key=layers, align=align, scale=scale, center=center
998: (4)                )

----------------------------------------

File 416 - nx_agraph.py:

1: (0)              """
2: (0)              ***************
3: (0)              Graphviz AGraph
4: (0)              ***************
5: (0)              Interface to pygraphviz AGraph class.
6: (0)              Examples
7: (0)              --------
8: (0)              >>> G = nx.complete_graph(5)
9: (0)              >>> A = nx.nx_agraph.to_agraph(G)
10: (0)             >>> H = nx.nx_agraph.from_agraph(A)
11: (0)             See Also
12: (0)             --------
```

```
 - Pygraphviz: http://pygraphviz.github.io/
 - Graphviz:      https://www.graphviz.org
 - DOT Language:  http://www.graphviz.org/doc/info/lang.html
"""
import os
import tempfile
import networkx as nx
__all__ = [
    "from_agraph",
    "to_agraph",
    "write_dot",
    "read_dot",
    "graphviz_layout",
    "pygraphviz_layout",
    "view_pygraphviz",
]
@nx._dispatchable(graphs=None, returns_graph=True)
def from_agraph(A, create_using=None):
    """Returns a NetworkX Graph or DiGraph from a PyGraphviz graph.
    Parameters
    ----------
    A : PyGraphviz AGraph
      A graph created with PyGraphviz
    create_using : NetworkX graph constructor, optional (default=None)
       Graph type to create. If graph instance, then cleared before populated.
       If `None`, then the appropriate Graph type is inferred from `A`.
    Examples
    --------
    >>> K5 = nx.complete_graph(5)
    >>> A = nx.nx_agraph.to_agraph(K5)
    >>> G = nx.nx_agraph.from_agraph(A)
    Notes
    -----
    The Graph G will have a dictionary G.graph_attr containing
    the default graphviz attributes for graphs, nodes and edges.
    Default node attributes will be in the dictionary G.node_attr
    which is keyed by node.
    Edge attributes will be returned as edge data in G.  With
    edge_attr=False the edge data will be the Graphviz edge weight
    attribute or the value 1 if no edge weight attribute is found.
    """
    if create_using is None:
        if A.is_directed():
            if A.is_strict():
                create_using = nx.DiGraph
            else:
                create_using = nx.MultiDiGraph
        else:
            if A.is_strict():
                create_using = nx.Graph
            else:
                create_using = nx.MultiGraph
    N = nx.empty_graph(0, create_using)
    if A.name is not None:
        N.name = A.name
    N.graph.update(A.graph_attr)
    for n in A.nodes():
        str_attr = {str(k): v for k, v in n.attr.items()}
        N.add_node(str(n), **str_attr)
    for e in A.edges():
        u, v = str(e[0]), str(e[1])
        attr = dict(e.attr)
        str_attr = {str(k): v for k, v in attr.items()}
        if not N.is_multigraph():
            if e.name is not None:
                str_attr["key"] = e.name
            N.add_edge(u, v, **str_attr)
        else:
            N.add_edge(u, v, key=e.name, **str_attr)
```

```
 82: (4)                 N.graph["graph"] = dict(A.graph_attr)
 83: (4)                 N.graph["node"] = dict(A.node_attr)
 84: (4)                 N.graph["edge"] = dict(A.edge_attr)
 85: (4)                 return N
 86: (0)             def to_agraph(N):
 87: (4)                 """Returns a pygraphviz graph from a NetworkX graph N.
 88: (4)                 Parameters
 89: (4)                 ----------
 90: (4)                 N : NetworkX graph
 91: (6)                   A graph created with NetworkX
 92: (4)                 Examples
 93: (4)                 --------
 94: (4)                 >>> K5 = nx.complete_graph(5)
 95: (4)                 >>> A = nx.nx_agraph.to_agraph(K5)
 96: (4)                 Notes
 97: (4)                 -----
 98: (4)                 If N has an dict N.graph_attr an attempt will be made first
 99: (4)                 to copy properties attached to the graph (see from_agraph)
 100: (4)                and then updated with the calling arguments if any.
 101: (4)                """
 102: (4)                try:
 103: (8)                    import pygraphviz
 104: (4)                except ImportError as err:
 105: (8)                    raise ImportError("requires pygraphviz http://pygraphviz.github.io/")
 from err
 106: (4)                directed = N.is_directed()
 107: (4)                strict = nx.number_of_selfloops(N) == 0 and not N.is_multigraph()
 108: (4)                for node in N:
 109: (8)                    if "pos" in N.nodes[node]:
 110: (12)                       N.nodes[node]["pos"] = "{},{}!".format(
 111: (16)                           N.nodes[node]["pos"][0], N.nodes[node]["pos"][1]
 112: (12)                       )
 113: (4)                A = pygraphviz.AGraph(name=N.name, strict=strict, directed=directed)
 114: (4)                A.graph_attr.update(N.graph.get("graph", {}))
 115: (4)                A.node_attr.update(N.graph.get("node", {}))
 116: (4)                A.edge_attr.update(N.graph.get("edge", {}))
 117: (4)                A.graph_attr.update(
 118: (8)                    (k, v) for k, v in N.graph.items() if k not in ("graph", "node",
 "edge")
 119: (4)                )
 120: (4)                for n, nodedata in N.nodes(data=True):
 121: (8)                    A.add_node(n)
 122: (8)                    a = A.get_node(n)
 123: (8)                    a.attr.update({k: str(v) for k, v in nodedata.items()})
 124: (4)                if N.is_multigraph():
 125: (8)                    for u, v, key, edgedata in N.edges(data=True, keys=True):
 126: (12)                       str_edgedata = {k: str(v) for k, v in edgedata.items() if k !=
 "key"}
 127: (12)                       A.add_edge(u, v, key=str(key))
 128: (12)                       a = A.get_edge(u, v)
 129: (12)                       a.attr.update(str_edgedata)
 130: (4)                else:
 131: (8)                    for u, v, edgedata in N.edges(data=True):
 132: (12)                       str_edgedata = {k: str(v) for k, v in edgedata.items()}
 133: (12)                       A.add_edge(u, v)
 134: (12)                       a = A.get_edge(u, v)
 135: (12)                       a.attr.update(str_edgedata)
 136: (4)                return A
 137: (0)            def write_dot(G, path):
 138: (4)                """Write NetworkX graph G to Graphviz dot format on path.
 139: (4)                Parameters
 140: (4)                ----------
 141: (4)                G : graph
 142: (7)                   A networkx graph
 143: (4)                path : filename
 144: (7)                   Filename or file handle to write
 145: (4)                Notes
 146: (4)                -----
 147: (4)                To use a specific graph layout, call ``A.layout`` prior to `write_dot`.
```

```
148: (4)                Note that some graphviz layouts are not guaranteed to be deterministic,
149: (4)                see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.
150: (4)                """
151: (4)                A = to_agraph(G)
152: (4)                A.write(path)
153: (4)                A.clear()
154: (4)                return
155: (0)            @nx._dispatchable(name="agraph_read_dot", graphs=None, returns_graph=True)
156: (0)            def read_dot(path):
157: (4)                """Returns a NetworkX graph from a dot file on path.
158: (4)                Parameters
159: (4)                ----------
160: (4)                path : file or string
161: (7)                   File name or file handle to read.
162: (4)                """
163: (4)                try:
164: (8)                    import pygraphviz
165: (4)                except ImportError as err:
166: (8)                    raise ImportError(
167: (12)                       "read_dot() requires pygraphviz http://pygraphviz.github.io/"
168: (8)                    ) from err
169: (4)                A = pygraphviz.AGraph(file=path)
170: (4)                gr = from_agraph(A)
171: (4)                A.clear()
172: (4)                return gr
173: (0)            def graphviz_layout(G, prog="neato", root=None, args=""):
174: (4)                """Create node positions for G using Graphviz.
175: (4)                Parameters
176: (4)                ----------
177: (4)                G : NetworkX graph
178: (6)                  A graph created with NetworkX
179: (4)                prog : string
180: (6)                  Name of Graphviz layout program
181: (4)                root : string, optional
182: (6)                  Root node for twopi layout
183: (4)                args : string, optional
184: (6)                  Extra arguments to Graphviz layout program
185: (4)                Returns
186: (4)                -------
187: (4)                Dictionary of x, y, positions keyed by node.
188: (4)                Examples
189: (4)                --------
190: (4)                >>> G = nx.petersen_graph()
191: (4)                >>> pos = nx.nx_agraph.graphviz_layout(G)
192: (4)                >>> pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
193: (4)                Notes
194: (4)                -----
195: (4)                This is a wrapper for pygraphviz_layout.
196: (4)                Note that some graphviz layouts are not guaranteed to be deterministic,
197: (4)                see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.
198: (4)                """
199: (4)                return pygraphviz_layout(G, prog=prog, root=root, args=args)
200: (0)            def pygraphviz_layout(G, prog="neato", root=None, args=""):
201: (4)                """Create node positions for G using Graphviz.
202: (4)                Parameters
203: (4)                ----------
204: (4)                G : NetworkX graph
205: (6)                  A graph created with NetworkX
206: (4)                prog : string
207: (6)                  Name of Graphviz layout program
208: (4)                root : string, optional
209: (6)                  Root node for twopi layout
210: (4)                args : string, optional
211: (6)                  Extra arguments to Graphviz layout program
212: (4)                Returns
213: (4)                -------
214: (4)                node_pos : dict
215: (6)                  Dictionary of x, y, positions keyed by node.
216: (4)                Examples
```

```
(4)                 --------
(4)                 >>> G = nx.petersen_graph()
(4)                 >>> pos = nx.nx_agraph.graphviz_layout(G)
(4)                 >>> pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
(4)                 Notes
(4)                 -----
(4)                 If you use complex node objects, they may have the same string
(4)                 representation and GraphViz could treat them as the same node.
(4)                 The layout may assign both nodes a single location. See Issue #1568
(4)                 If this occurs in your case, consider relabeling the nodes just
(4)                 for the layout computation using something similar to::
(8)                     >>> H = nx.convert_node_labels_to_integers(G,
label_attribute="node_label")
(8)                     >>> H_layout = nx.nx_agraph.pygraphviz_layout(G, prog="dot")
(8)                     >>> G_layout = {H.nodes[n]["node_label"]: p for n, p in
H_layout.items()}
(4)                 Note that some graphviz layouts are not guaranteed to be deterministic,
(4)                 see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.
(4)                 """
(4)                 try:
(8)                     import pygraphviz
(4)                 except ImportError as err:
(8)                     raise ImportError("requires pygraphviz http://pygraphviz.github.io/")
from err
(4)                 if root is not None:
(8)                     args += f"-Groot={root}"
(4)                 A = to_agraph(G)
(4)                 A.layout(prog=prog, args=args)
(4)                 node_pos = {}
(4)                 for n in G:
(8)                     node = pygraphviz.Node(A, n)
(8)                     try:
(12)                        xs = node.attr["pos"].split(",")
(12)                        node_pos[n] = tuple(float(x) for x in xs)
(8)                     except:
(12)                        print("no position for node", n)
(12)                        node_pos[n] = (0.0, 0.0)
(4)                 return node_pos
(0)             @nx.utils.open_file(5, "w+b")
(0)             def view_pygraphviz(
(4)                 G, edgelabel=None, prog="dot", args="", suffix="", path=None, show=True
(0)             ):
(4)                 """Views the graph G using the specified layout algorithm.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     The machine to draw.
(4)                 edgelabel : str, callable, None
(8)                     If a string, then it specifies the edge attribute to be displayed
(8)                     on the edge labels. If a callable, then it is called for each
(8)                     edge and it should return the string to be displayed on the edges.
(8)                     The function signature of `edgelabel` should be edgelabel(data),
(8)                     where `data` is the edge attribute dictionary.
(4)                 prog : string
(8)                     Name of Graphviz layout program.
(4)                 args : str
(8)                     Additional arguments to pass to the Graphviz layout program.
(4)                 suffix : str
(8)                     If `filename` is None, we save to a temporary file.  The value of
(8)                     `suffix` will appear at the tail end of the temporary filename.
(4)                 path : str, None
(8)                     The filename used to save the image.  If None, save to a temporary
(8)                     file.  File formats are the same as those from pygraphviz.agraph.draw.
(4)                 show : bool, default = True
(8)                     Whether to display the graph with :mod:`PIL.Image.show`,
(8)                     default is `True`. If `False`, the rendered graph is still available
(8)                     at `path`.
(4)                 Returns
(4)                 -------
```

```
    path : str
        The filename of the generated image.
    A : PyGraphviz graph
        The PyGraphviz graph instance used to generate the image.
    Notes
    -----
    If this function is called in succession too quickly, sometimes the
    image is not displayed. So you might consider time.sleep(.5) between
    calls if you experience problems.
    Note that some graphviz layouts are not guaranteed to be deterministic,
    see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.
    """
    if not len(G):
        raise nx.NetworkXException("An empty graph cannot be drawn.")
    attrs = ["edge", "node", "graph"]
    for attr in attrs:
        if attr not in G.graph:
            G.graph[attr] = {}
    edge_attrs = {"fontsize": "10"}
    node_attrs = {
        "style": "filled",
        "fillcolor": "#0000FF40",
        "height": "0.75",
        "width": "0.75",
        "shape": "circle",
    }
    graph_attrs = {}
    def update_attrs(which, attrs):
        added = []
        for k, v in attrs.items():
            if k not in G.graph[which]:
                G.graph[which][k] = v
                added.append(k)
    def clean_attrs(which, added):
        for attr in added:
            del G.graph[which][attr]
        if not G.graph[which]:
            del G.graph[which]
    update_attrs("edge", edge_attrs)
    update_attrs("node", node_attrs)
    update_attrs("graph", graph_attrs)
    A = to_agraph(G)
    clean_attrs("edge", edge_attrs)
    clean_attrs("node", node_attrs)
    clean_attrs("graph", graph_attrs)
    if edgelabel is not None:
        if not callable(edgelabel):
            def func(data):
                return "".join(["  ", str(data[edgelabel]), "  "])
        else:
            func = edgelabel
        if G.is_multigraph():
            for u, v, key, data in G.edges(keys=True, data=True):
                edge = A.get_edge(u, v, str(key))
                edge.attr["label"] = str(func(data))
        else:
            for u, v, data in G.edges(data=True):
                edge = A.get_edge(u, v)
                edge.attr["label"] = str(func(data))
    if path is None:
        ext = "png"
        if suffix:
            suffix = f"_{suffix}.{ext}"
        else:
            suffix = f".{ext}"
        path = tempfile.NamedTemporaryFile(suffix=suffix, delete=False)
    else:
        pass
    A.draw(path=path, format=None, prog=prog, args=args)
```

```
352: (4)                path.close()
353: (4)                if show:
354: (8)                    from PIL import Image
355: (8)                    Image.open(path.name).show()
356: (4)                return path.name, A
```

File 417 - nx_latex.py:

```
1: (0)              r"""
2: (0)              *****
3: (0)              LaTeX
4: (0)              *****
5: (0)              Export NetworkX graphs in LaTeX format using the TikZ library within
TeX/LaTeX.
6: (0)              Usually, you will want the drawing to appear in a figure environment so
7: (0)              you use ``to_latex(G, caption="A caption")``. If you want the raw
8: (0)              drawing commands without a figure environment use :func:`to_latex_raw`.
9: (0)              And if you want to write to a file instead of just returning the latex
10: (0)             code as a string, use ``write_latex(G, "filename.tex", caption="A caption")``.
11: (0)             To construct a figure with subfigures for each graph to be shown, provide
12: (0)             ``to_latex`` or ``write_latex`` a list of graphs, a list of subcaptions,
13: (0)             and a number of rows of subfigures inside the figure.
14: (0)             To be able to refer to the figures or subfigures in latex using ``\\ref``,
15: (0)             the keyword ``latex_label`` is available for figures and `sub_labels` for
16: (0)             a list of labels, one for each subfigure.
17: (0)             We intend to eventually provide an interface to the TikZ Graph
18: (0)             features which include e.g. layout algorithms.
19: (0)             Let us know via github what you'd like to see available, or better yet
20: (0)             give us some code to do it, or even better make a github pull request
21: (0)             to add the feature.
22: (0)             The TikZ approach
23: (0)             ================
24: (0)             Drawing options can be stored on the graph as node/edge attributes, or
25: (0)             can be provided as dicts keyed by node/edge to a string of the options
26: (0)             for that node/edge. Similarly a label can be shown for each node/edge
27: (0)             by specifying the labels as graph node/edge attributes or by providing
28: (0)             a dict keyed by node/edge to the text to be written for that node/edge.
29: (0)             Options for the tikzpicture environment (e.g. "[scale=2]") can be provided
30: (0)             via a keyword argument. Similarly default node and edge options can be
31: (0)             provided through keywords arguments. The default node options are applied
32: (0)             to the single TikZ "path" that draws all nodes (and no edges). The default
edge
33: (0)             options are applied to a TikZ "scope" which contains a path for each edge.
34: (0)             Examples
35: (0)             ========
36: (0)             >>> G = nx.path_graph(3)
37: (0)             >>> nx.write_latex(G, "just_my_figure.tex", as_document=True)
38: (0)             >>> nx.write_latex(G, "my_figure.tex", caption="A path graph",
latex_label="fig1")
39: (0)             >>> latex_code = nx.to_latex(G)  # a string rather than a file
40: (0)             You can change many features of the nodes and edges.
41: (0)             >>> G = nx.path_graph(4, create_using=nx.DiGraph)
42: (0)             >>> pos = {n: (n, n) for n in G}  # nodes set on a line
43: (0)             >>> G.nodes[0]["style"] = "blue"
44: (0)             >>> G.nodes[2]["style"] = "line width=3,draw"
45: (0)             >>> G.nodes[3]["label"] = "Stop"
46: (0)             >>> G.edges[(0, 1)]["label"] = "1st Step"
47: (0)             >>> G.edges[(0, 1)]["label_opts"] = "near start"
48: (0)             >>> G.edges[(1, 2)]["style"] = "line width=3"
49: (0)             >>> G.edges[(1, 2)]["label"] = "2nd Step"
50: (0)             >>> G.edges[(2, 3)]["style"] = "green"
51: (0)             >>> G.edges[(2, 3)]["label"] = "3rd Step"
52: (0)             >>> G.edges[(2, 3)]["label_opts"] = "near end"
53: (0)             >>> nx.write_latex(G, "latex_graph.tex", pos=pos, as_document=True)
54: (0)             Then compile the LaTeX using something like ``pdflatex latex_graph.tex``
55: (0)             and view the pdf file created: ``latex_graph.pdf``.
56: (0)             If you want **subfigures** each containing one graph, you can input a list of
```

```
graphs.
57: (0)             >>> H1 = nx.path_graph(4)
58: (0)             >>> H2 = nx.complete_graph(4)
59: (0)             >>> H3 = nx.path_graph(8)
60: (0)             >>> H4 = nx.complete_graph(8)
61: (0)             >>> graphs = [H1, H2, H3, H4]
62: (0)             >>> caps = ["Path 4", "Complete graph 4", "Path 8", "Complete graph 8"]
63: (0)             >>> lbls = ["fig2a", "fig2b", "fig2c", "fig2d"]
64: (0)             >>> nx.write_latex(graphs, "subfigs.tex", n_rows=2, sub_captions=caps,
sub_labels=lbls)
65: (0)             >>> latex_code = nx.to_latex(graphs, n_rows=2, sub_captions=caps,
sub_labels=lbls)
66: (0)             >>> node_color = {0: "red", 1: "orange", 2: "blue", 3: "gray!90"}
67: (0)             >>> edge_width = {e: "line width=1.5" for e in H3.edges}
68: (0)             >>> pos = nx.circular_layout(H3)
69: (0)             >>> latex_code = nx.to_latex(H3, pos, node_options=node_color,
edge_options=edge_width)
70: (0)             >>> print(latex_code)
71: (0)             \documentclass{report}
72: (0)             \usepackage{tikz}
73: (0)             \usepackage{subcaption}
74: (0)             <BLANKLINE>
75: (0)             \begin{document}
76: (0)             \begin{figure}
77: (2)               \begin{tikzpicture}
78: (6)                   \draw
79: (8)                     (1.0, 0.0) node[red] (0){0}
80: (8)                     (0.707, 0.707) node[orange] (1){1}
81: (8)                     (-0.0, 1.0) node[blue] (2){2}
82: (8)                     (-0.707, 0.707) node[gray!90] (3){3}
83: (8)                     (-1.0, -0.0) node (4){4}
84: (8)                     (-0.707, -0.707) node (5){5}
85: (8)                     (0.0, -1.0) node (6){6}
86: (8)                     (0.707, -0.707) node (7){7};
87: (6)                   \begin{scope}[-]
88: (8)                     \draw[line width=1.5] (0) to (1);
89: (8)                     \draw[line width=1.5] (1) to (2);
90: (8)                     \draw[line width=1.5] (2) to (3);
91: (8)                     \draw[line width=1.5] (3) to (4);
92: (8)                     \draw[line width=1.5] (4) to (5);
93: (8)                     \draw[line width=1.5] (5) to (6);
94: (8)                     \draw[line width=1.5] (6) to (7);
95: (6)                   \end{scope}
96: (4)                 \end{tikzpicture}
97: (0)             \end{figure}
98: (0)             \end{document}
99: (0)             Notes
100: (0)            -----
101: (0)            If you want to change the preamble/postamble of the figure/document/subfigure
102: (0)            environment, use the keyword arguments: `figure_wrapper`, `document_wrapper`,
103: (0)            `subfigure_wrapper`. The default values are stored in private variables
104: (0)            e.g. ``nx.nx_layout._DOCUMENT_WRAPPER``
105: (0)            References
106: (0)            ----------
107: (0)            TikZ:          https://tikz.dev/
108: (0)            TikZ options details:   https://tikz.dev/tikz-actions
109: (0)            """
110: (0)            import numbers
111: (0)            import os
112: (0)            import networkx as nx
113: (0)            __all__ = [
114: (4)                "to_latex_raw",
115: (4)                "to_latex",
116: (4)                "write_latex",
117: (0)            ]
118: (0)            @nx.utils.not_implemented_for("multigraph")
119: (0)            def to_latex_raw(
120: (4)                G,
121: (4)                pos="pos",
```

```
    tikz_options="",
    default_node_options="",
    node_options="node_options",
    node_label="label",
    default_edge_options="",
    edge_options="edge_options",
    edge_label="label",
    edge_label_options="edge_label_options",
):
    """Return a string of the LaTeX/TikZ code to draw `G`
    This function produces just the code for the tikzpicture
    without any enclosing environment.
    Parameters
    ==========
    G : NetworkX graph
        The NetworkX graph to be drawn
    pos : string or dict (default "pos")
        The name of the node attribute on `G` that holds the position of each node.

        Positions can be sequences of length 2 with numbers for (x,y) coordinates.

        They can also be strings to denote positions in TikZ style, such as (x, y)

        or (angle:radius).
        If a dict, it should be keyed by node to a position.
        If an empty dict, a circular layout is computed by TikZ.
    tikz_options : string
        The tikzpicture options description defining the options for the picture.

        Often large scale options like `[scale=2]`.
    default_node_options : string
        The draw options for a path of nodes. Individual node options override these.
    node_options : string or dict
        The name of the node attribute on `G` that holds the options for each node.

        Or a dict keyed by node to a string holding the options for that node.
    node_label : string or dict
        The name of the node attribute on `G` that holds the node label (text)
        displayed for each node. If the attribute is "" or not present, the node

        itself is drawn as a string. LaTeX processing such as ``"$A_1$"`` is allowed.

        Or a dict keyed by node to a string holding the label for that node.
    default_edge_options : string
        The options for the scope drawing all edges. The default is "[-]" for
        undirected graphs and "[->]" for directed graphs.
    edge_options : string or dict
        The name of the edge attribute on `G` that holds the options for each edge.

        If the edge is a self-loop and ``"loop" not in edge_options`` the option

        "loop," is added to the options for the self-loop edge. Hence you can
        use "[loop above]" explicitly, but the default is "[loop]".
        Or a dict keyed by edge to a string holding the options for that edge.
    edge_label : string or dict
        The name of the edge attribute on `G` that holds the edge label (text)
        displayed for each edge. If the attribute is "" or not present, no edge

        label is drawn.
        Or a dict keyed by edge to a string holding the label for that edge.
    edge_label_options : string or dict
        The name of the edge attribute on `G` that holds the label options for
        each edge. For example, "[sloped,above,blue]". The default is no options.

        Or a dict keyed by edge to a string holding the label options for that edge.

    Returns
    =======
```

```
(4)                 latex_code : string
(7)                    The text string which draws the desired graph(s) when compiled by LaTeX.
(4)                 See Also
(4)                 ========
(4)                 to_latex
(4)                 write_latex
(4)                 """
(4)                 i4 = "\n    "
(4)                 i8 = "\n        "
(4)                 if not isinstance(pos, dict):
(8)                     pos = nx.get_node_attributes(G, pos)
(4)                 if not pos:
(8)                     pos = {n: f"({round(360.0 * i / len(G), 3)}:2)" for i, n in enumerate(G)}
(4)                 for node in G:
(8)                     if node not in pos:
(12)                        raise nx.NetworkXError(f"node {node} has no specified pos {pos}")
(8)                     posnode = pos[node]
(8)                     if not isinstance(posnode, str):
(12)                        try:
(16)                            posx, posy = posnode
(16)                            pos[node] = f"({round(posx, 3)}, {round(posy, 3)})"
(12)                        except (TypeError, ValueError):
(16)                            msg = f"position pos[{node}] is not 2-tuple or a string: {posnode}"
(16)                            raise nx.NetworkXError(msg)
(4)                 if not isinstance(node_options, dict):
(8)                     node_options = nx.get_node_attributes(G, node_options)
(4)                 if not isinstance(node_label, dict):
(8)                     node_label = nx.get_node_attributes(G, node_label)
(4)                 if not isinstance(edge_options, dict):
(8)                     edge_options = nx.get_edge_attributes(G, edge_options)
(4)                 if not isinstance(edge_label, dict):
(8)                     edge_label = nx.get_edge_attributes(G, edge_label)
(4)                 if not isinstance(edge_label_options, dict):
(8)                     edge_label_options = nx.get_edge_attributes(G, edge_label_options)
(4)                 topts = "" if tikz_options == "" else f"[{tikz_options.strip('[]')}]"
(4)                 defn = "" if default_node_options == "" else f"[{default_node_options.strip('[]')}]"
(4)                 linestyle = f"{'->' if G.is_directed() else '-'}"
(4)                 if default_edge_options == "":
(8)                     defe = "[" + linestyle + "]"
(4)                 elif "-" in default_edge_options:
(8)                     defe = default_edge_options
(4)                 else:
(8)                     defe = f"[{linestyle},{default_edge_options.strip('[]')}]"
(4)                 result = "  \\begin{tikzpicture}" + topts
(4)                 result += i4 + "  \\draw" + defn
(4)                 for n in G:
(8)                     nopts = f"[{node_options[n].strip('[]')}]" if n in node_options else ""
(8)                     ntext = f"{{{node_label[n]}}}" if n in node_label else f"{{{n}}}"
(8)                     result += i8 + f"{pos[n]} node{nopts} ({n}){ntext}"
(4)                 result += ";\n"
(4)                 result += "      \\begin{scope}" + defe
(4)                 for edge in G.edges:
(8)                     u, v = edge[:2]
(8)                     e_opts = f"{edge_options[edge]}".strip("[]") if edge in edge_options else ""
(8)                     if u == v and "loop" not in e_opts:
(12)                        e_opts = "loop," + e_opts
(8)                     e_opts = f"[{e_opts}]" if e_opts != "" else ""
(8)                     els = edge_label_options[edge] if edge in edge_label_options else ""
(8)                     els = f"[{els.strip('[]')}]"
(8)                     e_label = f" node{els} {{{edge_label[edge]}}}" if edge in edge_label else ""
(8)                     result += i8 + f"\\draw{e_opts} ({u}) to{e_label} ({v});"
(4)                 result += "\n      \\end{scope}\n    \\end{tikzpicture}\n"
```

```
240: (4)                return result
241: (0)            _DOC_WRAPPER_TIKZ = r"""\documentclass{{report}}
242: (0)            \usepackage{{tikz}}
243: (0)            \usepackage{{subcaption}}
244: (0)            \begin{{document}}
245: (0)            {content}
246: (0)            \end{{document}}"""
247: (0)            _FIG_WRAPPER = r"""\begin{{figure}}
248: (0)            {content}{caption}{label}
249: (0)            \end{{figure}}"""
250: (0)            _SUBFIG_WRAPPER = r"""  \begin{{subfigure}}{{{size}\textwidth}}
251: (0)            {content}{caption}{label}
252: (2)              \end{{subfigure}}"""
253: (0)            def to_latex(
254: (4)                Gbunch,
255: (4)                pos="pos",
256: (4)                tikz_options="",
257: (4)                default_node_options="",
258: (4)                node_options="node_options",
259: (4)                node_label="node_label",
260: (4)                default_edge_options="",
261: (4)                edge_options="edge_options",
262: (4)                edge_label="edge_label",
263: (4)                edge_label_options="edge_label_options",
264: (4)                caption="",
265: (4)                latex_label="",
266: (4)                sub_captions=None,
267: (4)                sub_labels=None,
268: (4)                n_rows=1,
269: (4)                as_document=True,
270: (4)                document_wrapper=_DOC_WRAPPER_TIKZ,
271: (4)                figure_wrapper=_FIG_WRAPPER,
272: (4)                subfigure_wrapper=_SUBFIG_WRAPPER,
273: (0)            ):
274: (4)                """Return latex code to draw the graph(s) in `Gbunch`
275: (4)                The TikZ drawing utility in LaTeX is used to draw the graph(s).
276: (4)                If `Gbunch` is a graph, it is drawn in a figure environment.
277: (4)                If `Gbunch` is an iterable of graphs, each is drawn in a subfigure
environment
278: (4)                within a single figure environment.
279: (4)                If `as_document` is True, the figure is wrapped inside a document
environment
280: (4)                so that the resulting string is ready to be compiled by LaTeX. Otherwise,
281: (4)                the string is ready for inclusion in a larger tex document using
``\\include``
282: (4)                or ``\\input`` statements.
283: (4)                Parameters
284: (4)                ==========
285: (4)                Gbunch : NetworkX graph or iterable of NetworkX graphs
286: (8)                    The NetworkX graph to be drawn or an iterable of graphs
287: (8)                    to be drawn inside subfigures of a single figure.
288: (4)                pos : string or list of strings
289: (8)                    The name of the node attribute on `G` that holds the position of each
node.
290: (8)                    Positions can be sequences of length 2 with numbers for (x,y)
coordinates.
291: (8)                    They can also be strings to denote positions in TikZ style, such as
(x, y)
292: (8)                    or (angle:radius).
293: (8)                    If a dict, it should be keyed by node to a position.
294: (8)                    If an empty dict, a circular layout is computed by TikZ.
295: (8)                    If you are drawing many graphs in subfigures, use a list of position
dicts.
296: (4)                tikz_options : string
297: (8)                    The tikzpicture options description defining the options for the
picture.
298: (8)                    Often large scale options like `[scale=2]`.
299: (4)                default_node_options : string
300: (8)                    The draw options for a path of nodes. Individual node options override
```

```
these.
    node_options : string or dict
        The name of the node attribute on `G` that holds the options for each node.

        Or a dict keyed by node to a string holding the options for that node.
    node_label : string or dict
        The name of the node attribute on `G` that holds the node label (text)
        displayed for each node. If the attribute is "" or not present, the node
        itself is drawn as a string. LaTeX processing such as ``"$A_1$"`` is allowed.

        Or a dict keyed by node to a string holding the label for that node.
    default_edge_options : string
        The options for the scope drawing all edges. The default is "[-]" for
        undirected graphs and "[->]" for directed graphs.
    edge_options : string or dict
        The name of the edge attribute on `G` that holds the options for each edge.

        If the edge is a self-loop and ``"loop" not in edge_options`` the option
        "loop," is added to the options for the self-loop edge. Hence you can
        use "[loop above]" explicitly, but the default is "[loop]".
        Or a dict keyed by edge to a string holding the options for that edge.
    edge_label : string or dict
        The name of the edge attribute on `G` that holds the edge label (text)
        displayed for each edge. If the attribute is "" or not present, no edge
        label is drawn.
        Or a dict keyed by edge to a string holding the label for that edge.
    edge_label_options : string or dict
        The name of the edge attribute on `G` that holds the label options for
        each edge. For example, "[sloped,above,blue]". The default is no options.
        Or a dict keyed by edge to a string holding the label options for that edge.
    caption : string
        The caption string for the figure environment
    latex_label : string
        The latex label used for the figure for easy referral from the main text
    sub_captions : list of strings
        The sub_caption string for each subfigure in the figure
    sub_latex_labels : list of strings
        The latex label for each subfigure in the figure
    n_rows : int
        The number of rows of subfigures to arrange for multiple graphs
    as_document : bool
        Whether to wrap the latex code in a document environment for compiling
    document_wrapper : formatted text string with variable ``content``.
        This text is called to evaluate the content embedded in a document
        environment with a preamble setting up TikZ.
    figure_wrapper : formatted text string
        This text is evaluated with variables ``content``, ``caption`` and ``label``.
        It wraps the content and if a caption is provided, adds the latex code for
        that caption, and if a label is provided, adds the latex code for a label.
    subfigure_wrapper : formatted text string
        This text evaluate variables ``size``, ``content``, ``caption`` and ``label``.
        It wraps the content and if a caption is provided, adds the latex code for
        that caption, and if a label is provided, adds the latex code for a label.
        The size is the vertical size of each row of subfigures as a fraction.
    Returns
    =======
    latex_code : string
```

```
354: (8)                    The text string which draws the desired graph(s) when compiled by
LaTeX.
355: (4)                See Also
356: (4)                ========
357: (4)                write_latex
358: (4)                to_latex_raw
359: (4)                """
360: (4)                if hasattr(Gbunch, "adj"):
361: (8)                    raw = to_latex_raw(
362: (12)                       Gbunch,
363: (12)                       pos,
364: (12)                       tikz_options,
365: (12)                       default_node_options,
366: (12)                       node_options,
367: (12)                       node_label,
368: (12)                       default_edge_options,
369: (12)                       edge_options,
370: (12)                       edge_label,
371: (12)                       edge_label_options,
372: (8)                    )
373: (4)                else:  # iterator of graphs
374: (8)                    sbf = subfigure_wrapper
375: (8)                    size = 1 / n_rows
376: (8)                    N = len(Gbunch)
377: (8)                    if isinstance(pos, str | dict):
378: (12)                       pos = [pos] * N
379: (8)                    if sub_captions is None:
380: (12)                       sub_captions = [""] * N
381: (8)                    if sub_labels is None:
382: (12)                       sub_labels = [""] * N
383: (8)                    if not (len(Gbunch) == len(pos) == len(sub_captions) ==
len(sub_labels)):
384: (12)                       raise nx.NetworkXError(
385: (16)                           "length of Gbunch, sub_captions and sub_figures must agree"
386: (12)                       )
387: (8)                    raw = ""
388: (8)                    for G, pos, subcap, sublbl in zip(Gbunch, pos, sub_captions,
sub_labels):
389: (12)                       subraw = to_latex_raw(
390: (16)                           G,
391: (16)                           pos,
392: (16)                           tikz_options,
393: (16)                           default_node_options,
394: (16)                           node_options,
395: (16)                           node_label,
396: (16)                           default_edge_options,
397: (16)                           edge_options,
398: (16)                           edge_label,
399: (16)                           edge_label_options,
400: (12)                       )
401: (12)                       cap = f"    \\caption{{{subcap}}}" if subcap else ""
402: (12)                       lbl = f"\\label{{{sublbl}}}" if sublbl else ""
403: (12)                       raw += sbf.format(size=size, content=subraw, caption=cap,
label=lbl)
404: (12)                       raw += "\n"
405: (4)                raw = raw[:-1]
406: (4)                cap = f"\n  \\caption{{{caption}}}" if caption else ""
407: (4)                lbl = f"\\label{{{latex_label}}}" if latex_label else ""
408: (4)                fig = figure_wrapper.format(content=raw, caption=cap, label=lbl)
409: (4)                if as_document:
410: (8)                    return document_wrapper.format(content=fig)
411: (4)                return fig
412: (0)            @nx.utils.open_file(1, mode="w")
413: (0)            def write_latex(Gbunch, path, **options):
414: (4)                """Write the latex code to draw the graph(s) onto `path`.
415: (4)                This convenience function creates the latex drawing code as a string
416: (4)                and writes that to a file ready to be compiled when `as_document` is True
417: (4)                or ready to be ``import`` ed or ``include`` ed into your main LaTeX
document.
```

```
    The `path` argument can be a string filename or a file handle to write to.
    Parameters
    ----------
    Gbunch : NetworkX graph or iterable of NetworkX graphs
        If Gbunch is a graph, it is drawn in a figure environment.
        If Gbunch is an iterable of graphs, each is drawn in a subfigure
        environment within a single figure environment.
    path : filename
        Filename or file handle to write to
    options : dict
        By default, TikZ is used with options: (others are ignored)::
            pos : string or dict or list
                The name of the node attribute on `G` that holds the position of each node.

                Positions can be sequences of length 2 with numbers for (x,y) coordinates.

                They can also be strings to denote positions in TikZ style, such as (x, y)

                or (angle:radius).
                If a dict, it should be keyed by node to a position.
                If an empty dict, a circular layout is computed by TikZ.
                If you are drawing many graphs in subfigures, use a list of position dicts.
            tikz_options : string
                The tikzpicture options description defining the options for the picture.

                Often large scale options like `[scale=2]`.
            default_node_options : string
                The draw options for a path of nodes. Individual node options override these.
            node_options : string or dict
                The name of the node attribute on `G` that holds the options for each node.

                Or a dict keyed by node to a string holding the options for that node.
            node_label : string or dict
                The name of the node attribute on `G` that holds the node label (text)

                displayed for each node. If the attribute is "" or not present, the node

                itself is drawn as a string. LaTeX processing such as ``"$A_1$"`` is allowed.

                Or a dict keyed by node to a string holding the label for that node.
            default_edge_options : string
                The options for the scope drawing all edges. The default is "[-]" for

                undirected graphs and "[->]" for directed graphs.
            edge_options : string or dict
                The name of the edge attribute on `G` that holds the options for each edge.

                If the edge is a self-loop and ``"loop" not in edge_options`` the option

                "loop," is added to the options for the self-loop edge. Hence you can

                use "[loop above]" explicitly, but the default is "[loop]".
                Or a dict keyed by edge to a string holding the options for that edge.
            edge_label : string or dict
                The name of the edge attribute on `G` that holds the edge label (text)

                displayed for each edge. If the attribute is "" or not present, no edge

                label is drawn.
                Or a dict keyed by edge to a string holding the label for that edge.
            edge_label_options : string or dict
                The name of the edge attribute on `G` that holds the label options for
```

```
466: (16)                           each edge. For example, "[sloped,above,blue]". The default is
no options.
467: (16)                           Or a dict keyed by edge to a string holding the label options
for that edge.
468: (12)                       caption : string
469: (16)                           The caption string for the figure environment
470: (12)                       latex_label : string
471: (16)                           The latex label used for the figure for easy referral from the
main text
472: (12)                       sub_captions : list of strings
473: (16)                           The sub_caption string for each subfigure in the figure
474: (12)                       sub_latex_labels : list of strings
475: (16)                           The latex label for each subfigure in the figure
476: (12)                       n_rows : int
477: (16)                           The number of rows of subfigures to arrange for multiple
graphs
478: (12)                       as_document : bool
479: (16)                           Whether to wrap the latex code in a document environment for
compiling
480: (12)                       document_wrapper : formatted text string with variable
``content``.
481: (16)                           This text is called to evaluate the content embedded in a
document
482: (16)                           environment with a preamble setting up the TikZ syntax.
483: (12)                       figure_wrapper : formatted text string
484: (16)                           This text is evaluated with variables ``content``, ``caption``
and ``label``.
485: (16)                           It wraps the content and if a caption is provided, adds the
latex code for
486: (16)                           that caption, and if a label is provided, adds the latex code
for a label.
487: (12)                       subfigure_wrapper : formatted text string
488: (16)                           This text evaluate variables ``size``, ``content``,
``caption`` and ``label``.
489: (16)                           It wraps the content and if a caption is provided, adds the
latex code for
490: (16)                           that caption, and if a label is provided, adds the latex code
for a label.
491: (16)                           The size is the vertical size of each row of subfigures as a
fraction.
492: (4)                See Also
493: (4)                ========
494: (4)                to_latex
495: (4)                """
496: (4)                path.write(to_latex(Gbunch, **options))

----------------------------------------

File 418 - nx_pydot.py:

1: (0)              """
2: (0)              *****
3: (0)              Pydot
4: (0)              *****
5: (0)              Import and export NetworkX graphs in Graphviz dot format using pydot.
6: (0)              Either this module or nx_agraph can be used to interface with graphviz.
7: (0)              Examples
8: (0)              --------
9: (0)              >>> G = nx.complete_graph(5)
10: (0)             >>> PG = nx.nx_pydot.to_pydot(G)
11: (0)             >>> H = nx.nx_pydot.from_pydot(PG)
12: (0)             See Also
13: (0)             --------
14: (1)              - pydot:         https://github.com/erocarrera/pydot
15: (1)              - Graphviz:      https://www.graphviz.org
16: (1)              - DOT Language:  http://www.graphviz.org/doc/info/lang.html
17: (0)             """
18: (0)             from locale import getpreferredencoding
19: (0)             import networkx as nx
```

```
20: (0)             from networkx.utils import open_file
21: (0)             __all__ = [
22: (4)                 "write_dot",
23: (4)                 "read_dot",
24: (4)                 "graphviz_layout",
25: (4)                 "pydot_layout",
26: (4)                 "to_pydot",
27: (4)                 "from_pydot",
28: (0)             ]
29: (0)             @open_file(1, mode="w")
30: (0)             def write_dot(G, path):
31: (4)                 """Write NetworkX graph G to Graphviz dot format on path.
32: (4)                 Path can be a string or a file handle.
33: (4)                 """
34: (4)                 P = to_pydot(G)
35: (4)                 path.write(P.to_string())
36: (4)                 return
37: (0)             @open_file(0, mode="r")
38: (0)             @nx._dispatchable(name="pydot_read_dot", graphs=None, returns_graph=True)
39: (0)             def read_dot(path):
40: (4)                 """Returns a NetworkX :class:`MultiGraph` or :class:`MultiDiGraph` from
the
41: (4)                 dot file with the passed path.
42: (4)                 If this file contains multiple graphs, only the first such graph is
43: (4)                 returned. All graphs _except_ the first are silently ignored.
44: (4)                 Parameters
45: (4)                 ----------
46: (4)                 path : str or file
47: (8)                     Filename or file handle.
48: (4)                 Returns
49: (4)                 -------
50: (4)                 G : MultiGraph or MultiDiGraph
51: (8)                     A :class:`MultiGraph` or :class:`MultiDiGraph`.
52: (4)                 Notes
53: (4)                 -----
54: (4)                 Use `G = nx.Graph(nx.nx_pydot.read_dot(path))` to return a :class:`Graph`
instead of a
55: (4)                 :class:`MultiGraph`.
56: (4)                 """
57: (4)                 import pydot
58: (4)                 data = path.read()
59: (4)                 P_list = pydot.graph_from_dot_data(data)
60: (4)                 return from_pydot(P_list[0])
61: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
62: (0)             def from_pydot(P):
63: (4)                 """Returns a NetworkX graph from a Pydot graph.
64: (4)                 Parameters
65: (4)                 ----------
66: (4)                 P : Pydot graph
67: (6)                   A graph created with Pydot
68: (4)                 Returns
69: (4)                 -------
70: (4)                 G : NetworkX multigraph
71: (8)                     A MultiGraph or MultiDiGraph.
72: (4)                 Examples
73: (4)                 --------
74: (4)                 >>> K5 = nx.complete_graph(5)
75: (4)                 >>> A = nx.nx_pydot.to_pydot(K5)
76: (4)                 >>> G = nx.nx_pydot.from_pydot(A)  # return MultiGraph
77: (4)                 >>> G = nx.Graph(nx.nx_pydot.from_pydot(A))
78: (4)                 """
79: (4)                 if P.get_strict(None):  # pydot bug: get_strict() shouldn't take argument
80: (8)                     multiedges = False
81: (4)                 else:
82: (8)                     multiedges = True
83: (4)                 if P.get_type() == "graph":  # undirected
84: (8)                     if multiedges:
85: (12)                        N = nx.MultiGraph()
86: (8)                     else:
```

```
87: (12)                        N = nx.Graph()
88: (4)                 else:
89: (8)                     if multiedges:
90: (12)                        N = nx.MultiDiGraph()
91: (8)                     else:
92: (12)                        N = nx.DiGraph()
93: (4)                 name = P.get_name().strip('"')
94: (4)                 if name != "":
95: (8)                     N.name = name
96: (4)                 for p in P.get_node_list():
97: (8)                     n = p.get_name().strip('"')
98: (8)                     if n in ("node", "graph", "edge"):
99: (12)                        continue
100: (8)                    N.add_node(n, **p.get_attributes())
101: (4)                for e in P.get_edge_list():
102: (8)                    u = e.get_source()
103: (8)                    v = e.get_destination()
104: (8)                    attr = e.get_attributes()
105: (8)                    s = []
106: (8)                    d = []
107: (8)                    if isinstance(u, str):
108: (12)                       s.append(u.strip('"'))
109: (8)                    else:
110: (12)                       for unodes in u["nodes"]:
111: (16)                           s.append(unodes.strip('"'))
112: (8)                    if isinstance(v, str):
113: (12)                       d.append(v.strip('"'))
114: (8)                    else:
115: (12)                       for vnodes in v["nodes"]:
116: (16)                           d.append(vnodes.strip('"'))
117: (8)                    for source_node in s:
118: (12)                       for destination_node in d:
119: (16)                           N.add_edge(source_node, destination_node, **attr)
120: (4)                pattr = P.get_attributes()
121: (4)                if pattr:
122: (8)                    N.graph["graph"] = pattr
123: (4)                try:
124: (8)                    N.graph["node"] = P.get_node_defaults()[0]
125: (4)                except (IndexError, TypeError):
126: (8)                    pass  # N.graph['node']={}
127: (4)                try:
128: (8)                    N.graph["edge"] = P.get_edge_defaults()[0]
129: (4)                except (IndexError, TypeError):
130: (8)                    pass  # N.graph['edge']={}
131: (4)                return N
132: (0)            def _check_colon_quotes(s):
133: (4)                return ":" in s and (s[0] != '"' or s[-1] != '"')
134: (0)            def to_pydot(N):
135: (4)                """Returns a pydot graph from a NetworkX graph N.
136: (4)                Parameters
137: (4)                ----------
138: (4)                N : NetworkX graph
139: (6)                  A graph created with NetworkX
140: (4)                Examples
141: (4)                --------
142: (4)                >>> K5 = nx.complete_graph(5)
143: (4)                >>> P = nx.nx_pydot.to_pydot(K5)
144: (4)                Notes
145: (4)                -----
146: (4)                """
147: (4)                import pydot
148: (4)                if N.is_directed():
149: (8)                    graph_type = "digraph"
150: (4)                else:
151: (8)                    graph_type = "graph"
152: (4)                strict = nx.number_of_selfloops(N) == 0 and not N.is_multigraph()
153: (4)                name = N.name
154: (4)                graph_defaults = N.graph.get("graph", {})
155: (4)                if name == "":
```

```
156: (8)                    P = pydot.Dot("", graph_type=graph_type, strict=strict,
**graph_defaults)
157: (4)                else:
158: (8)                    P = pydot.Dot(
159: (12)                       f'"{name}"', graph_type=graph_type, strict=strict,
**graph_defaults
160: (8)                    )
161: (4)                try:
162: (8)                    P.set_node_defaults(**N.graph["node"])
163: (4)                except KeyError:
164: (8)                    pass
165: (4)                try:
166: (8)                    P.set_edge_defaults(**N.graph["edge"])
167: (4)                except KeyError:
168: (8)                    pass
169: (4)                for n, nodedata in N.nodes(data=True):
170: (8)                    str_nodedata = {str(k): str(v) for k, v in nodedata.items()}
171: (8)                    n = str(n)
172: (8)                    raise_error = _check_colon_quotes(n) or (
173: (12)                       any(
174: (16)                           (_check_colon_quotes(k) or _check_colon_quotes(v))
175: (16)                           for k, v in str_nodedata.items()
176: (12)                       )
177: (8)                    )
178: (8)                    if raise_error:
179: (12)                       raise ValueError(
180: (16)                           f'Node names and attributes should not contain ":" unless they
are quoted with "".\
181: (16)                           For example the string \'attribute:data1\' should be written
as \'"attribute:data1"\'.\
182: (16)                           Please refer https://github.com/pydot/pydot/issues/258'
183: (12)                       )
184: (8)                    p = pydot.Node(n, **str_nodedata)
185: (8)                    P.add_node(p)
186: (4)                if N.is_multigraph():
187: (8)                    for u, v, key, edgedata in N.edges(data=True, keys=True):
188: (12)                       str_edgedata = {str(k): str(v) for k, v in edgedata.items() if k
!= "key"}
189: (12)                       u, v = str(u), str(v)
190: (12)                       raise_error = (
191: (16)                           _check_colon_quotes(u)
192: (16)                           or _check_colon_quotes(v)
193: (16)                           or (
194: (20)                               any(
195: (24)                                   (_check_colon_quotes(k) or _check_colon_quotes(val))
196: (24)                                   for k, val in str_edgedata.items()
197: (20)                               )
198: (16)                           )
199: (12)                       )
200: (12)                       if raise_error:
201: (16)                           raise ValueError(
202: (20)                               f'Node names and attributes should not contain ":" unless
they are quoted with "".\
203: (20)                               For example the string \'attribute:data1\' should be
written as \'"attribute:data1"\'.\
204: (20)                               Please refer https://github.com/pydot/pydot/issues/258'
205: (16)                           )
206: (12)                       edge = pydot.Edge(u, v, key=str(key), **str_edgedata)
207: (12)                       P.add_edge(edge)
208: (4)                else:
209: (8)                    for u, v, edgedata in N.edges(data=True):
210: (12)                       str_edgedata = {str(k): str(v) for k, v in edgedata.items()}
211: (12)                       u, v = str(u), str(v)
212: (12)                       raise_error = (
213: (16)                           _check_colon_quotes(u)
214: (16)                           or _check_colon_quotes(v)
215: (16)                           or (
216: (20)                               any(
217: (24)                                   (_check_colon_quotes(k) or _check_colon_quotes(val))
```

```
(24)                                    for k, val in str_edgedata.items()
(20)                                )
(16)                            )
(12)                        )
(12)                        if raise_error:
(16)                            raise ValueError(
(20)                                f'Node names and attributes should not contain ":" unless they are quoted with "".\
(20)                                For example the string \'attribute:data1\' should be written as \'"attribute:data1"\'.\
(20)                                Please refer https://github.com/pydot/pydot/issues/258'
(16)                            )
(12)                        edge = pydot.Edge(u, v, **str_edgedata)
(12)                        P.add_edge(edge)
(4)                 return P
(0)             def graphviz_layout(G, prog="neato", root=None):
(4)                 """Create node positions using Pydot and Graphviz.
(4)                 Returns a dictionary of positions keyed by node.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(8)                     The graph for which the layout is computed.
(4)                 prog : string (default: 'neato')
(8)                     The name of the GraphViz program to use for layout.
(8)                     Options depend on GraphViz version but may include:
(8)                     'dot', 'twopi', 'fdp', 'sfdp', 'circo'
(4)                 root : Node from G or None (default: None)
(8)                     The node of G from which to start some layout algorithms.
(4)                 Returns
(4)                 -------
(6)                   Dictionary of (x, y) positions keyed by node.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.complete_graph(4)
(4)                 >>> pos = nx.nx_pydot.graphviz_layout(G)
(4)                 >>> pos = nx.nx_pydot.graphviz_layout(G, prog="dot")
(4)                 Notes
(4)                 -----
(4)                 This is a wrapper for pydot_layout.
(4)                 """
(4)                 return pydot_layout(G=G, prog=prog, root=root)
(0)             def pydot_layout(G, prog="neato", root=None):
(4)                 """Create node positions using :mod:`pydot` and Graphviz.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph
(8)                     NetworkX graph to be laid out.
(4)                 prog : string  (default: 'neato')
(8)                     Name of the GraphViz command to use for layout.
(8)                     Options depend on GraphViz version but may include:
(8)                     'dot', 'twopi', 'fdp', 'sfdp', 'circo'
(4)                 root : Node from G or None (default: None)
(8)                     The node of G from which to start some layout algorithms.
(4)                 Returns
(4)                 -------
(4)                 dict
(8)                     Dictionary of positions keyed by node.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.complete_graph(4)
(4)                 >>> pos = nx.nx_pydot.pydot_layout(G)
(4)                 >>> pos = nx.nx_pydot.pydot_layout(G, prog="dot")
(4)                 Notes
(4)                 -----
(4)                 If you use complex node objects, they may have the same string
(4)                 representation and GraphViz could treat them as the same node.
(4)                 The layout may assign both nodes a single location. See Issue #1568
(4)                 If this occurs in your case, consider relabeling the nodes just
(4)                 for the layout computation using something similar to::
```

```
285: (8)                    H = nx.convert_node_labels_to_integers(G,
label_attribute="node_label")
286: (8)                    H_layout = nx.nx_pydot.pydot_layout(G, prog="dot")
287: (8)                    G_layout = {H.nodes[n]["node_label"]: p for n, p in H_layout.items()}
288: (4)                """
289: (4)                import pydot
290: (4)                P = to_pydot(G)
291: (4)                if root is not None:
292: (8)                    P.set("root", str(root))
293: (4)                D_bytes = P.create_dot(prog=prog)
294: (4)                D = str(D_bytes, encoding=getpreferredencoding())
295: (4)                if D == "":  # no data returned
296: (8)                    print(f"Graphviz layout with {prog} failed")
297: (8)                    print()
298: (8)                    print("To debug what happened try:")
299: (8)                    print("P = nx.nx_pydot.to_pydot(G)")
300: (8)                    print('P.write_dot("file.dot")')
301: (8)                    print(f"And then run {prog} on file.dot")
302: (8)                    return
303: (4)                Q_list = pydot.graph_from_dot_data(D)
304: (4)                assert len(Q_list) == 1
305: (4)                Q = Q_list[0]
306: (4)                node_pos = {}
307: (4)                for n in G.nodes():
308: (8)                    str_n = str(n)
309: (8)                    if _check_colon_quotes(str_n):
310: (12)                       raise ValueError(
311: (16)                           f'Node names and node attributes should not contain ":" unless
they are quoted with "".\
312: (16)                           For example the string \'attribute:data1\' should be written
as \'"attribute:data1"\'.\
313: (16)                           Please refer https://github.com/pydot/pydot/issues/258'
314: (12)                       )
315: (8)                    pydot_node = pydot.Node(str_n).get_name()
316: (8)                    node = Q.get_node(pydot_node)
317: (8)                    if isinstance(node, list):
318: (12)                       node = node[0]
319: (8)                    pos = node.get_pos()[1:-1]  # strip leading and trailing double quotes
320: (8)                    if pos is not None:
321: (12)                       xx, yy = pos.split(",")
322: (12)                       node_pos[n] = (float(xx), float(yy))
323: (4)                return node_pos
```

File 419 - nx_pylab.py:

```
1: (0)              """
2: (0)              **********
3: (0)              Matplotlib
4: (0)              **********
5: (0)              Draw networks with matplotlib.
6: (0)              Examples
7: (0)              --------
8: (0)              >>> G = nx.complete_graph(5)
9: (0)              >>> nx.draw(G)
10: (0)             See Also
11: (0)             --------
12: (1)              - :doc:`matplotlib <matplotlib:index>`
13: (1)              - :func:`matplotlib.pyplot.scatter`
14: (1)              - :obj:`matplotlib.patches.FancyArrowPatch`
15: (0)             """
16: (0)             import collections
17: (0)             import itertools
18: (0)             from numbers import Number
19: (0)             import networkx as nx
20: (0)             from networkx.drawing.layout import (
21: (4)                 circular_layout,
22: (4)                 kamada_kawai_layout,
```

```
23: (4)                 planar_layout,
24: (4)                 random_layout,
25: (4)                 shell_layout,
26: (4)                 spectral_layout,
27: (4)                 spring_layout,
28: (0)             )
29: (0)             __all__ = [
30: (4)                 "draw",
31: (4)                 "draw_networkx",
32: (4)                 "draw_networkx_nodes",
33: (4)                 "draw_networkx_edges",
34: (4)                 "draw_networkx_labels",
35: (4)                 "draw_networkx_edge_labels",
36: (4)                 "draw_circular",
37: (4)                 "draw_kamada_kawai",
38: (4)                 "draw_random",
39: (4)                 "draw_spectral",
40: (4)                 "draw_spring",
41: (4)                 "draw_planar",
42: (4)                 "draw_shell",
43: (0)             ]
44: (0)             def draw(G, pos=None, ax=None, **kwds):
45: (4)                 """Draw the graph G with Matplotlib.
46: (4)                 Draw the graph as a simple representation with no node
47: (4)                 labels or edge labels and using the full Matplotlib figure area
48: (4)                 and no axis labels by default.  See draw_networkx() for more
49: (4)                 full-featured drawing that allows title, axis labels etc.
50: (4)                 Parameters
51: (4)                 ----------
52: (4)                 G : graph
53: (8)                     A networkx graph
54: (4)                 pos : dictionary, optional
55: (8)                     A dictionary with nodes as keys and positions as values.
56: (8)                     If not specified a spring layout positioning will be computed.
57: (8)                     See :py:mod:`networkx.drawing.layout` for functions that
58: (8)                     compute node positions.
59: (4)                 ax : Matplotlib Axes object, optional
60: (8)                     Draw the graph in specified Matplotlib axes.
61: (4)                 kwds : optional keywords
62: (8)                     See networkx.draw_networkx() for a description of optional keywords.
63: (4)                 Examples
64: (4)                 --------
65: (4)                 >>> G = nx.dodecahedral_graph()
66: (4)                 >>> nx.draw(G)
67: (4)                 >>> nx.draw(G, pos=nx.spring_layout(G))  # use spring layout
68: (4)                 See Also
69: (4)                 --------
70: (4)                 draw_networkx
71: (4)                 draw_networkx_nodes
72: (4)                 draw_networkx_edges
73: (4)                 draw_networkx_labels
74: (4)                 draw_networkx_edge_labels
75: (4)                 Notes
76: (4)                 -----
77: (4)                 This function has the same name as pylab.draw and pyplot.draw
78: (4)                 so beware when using `from networkx import *`
79: (4)                 since you might overwrite the pylab.draw function.
80: (4)                 With pyplot use
81: (4)                 >>> import matplotlib.pyplot as plt
82: (4)                 >>> G = nx.dodecahedral_graph()
83: (4)                 >>> nx.draw(G)  # networkx draw()
84: (4)                 >>> plt.draw()  # pyplot draw()
85: (4)                 Also see the NetworkX drawing examples at
86: (4)                 https://networkx.org/documentation/latest/auto_examples/index.html
87: (4)                 """
88: (4)                 import matplotlib.pyplot as plt
89: (4)                 if ax is None:
90: (8)                     cf = plt.gcf()
91: (4)                 else:
```

```
92: (8)                     cf = ax.get_figure()
93: (4)                 cf.set_facecolor("w")
94: (4)                 if ax is None:
95: (8)                     if cf.axes:
96: (12)                        ax = cf.gca()
97: (8)                     else:
98: (12)                        ax = cf.add_axes((0, 0, 1, 1))
99: (4)                 if "with_labels" not in kwds:
100: (8)                    kwds["with_labels"] = "labels" in kwds
101: (4)                draw_networkx(G, pos=pos, ax=ax, **kwds)
102: (4)                ax.set_axis_off()
103: (4)                plt.draw_if_interactive()
104: (4)                return
105: (0)            def draw_networkx(G, pos=None, arrows=None, with_labels=True, **kwds):
106: (4)                r"""Draw the graph G using Matplotlib.
107: (4)                Draw the graph with Matplotlib with options for node positions,
108: (4)                labeling, titles, and many other drawing features.
109: (4)                See draw() for simple drawing without labels or axes.
110: (4)                Parameters
111: (4)                ----------
112: (4)                G : graph
113: (8)                    A networkx graph
114: (4)                pos : dictionary, optional
115: (8)                    A dictionary with nodes as keys and positions as values.
116: (8)                    If not specified a spring layout positioning will be computed.
117: (8)                    See :py:mod:`networkx.drawing.layout` for functions that
118: (8)                    compute node positions.
119: (4)                arrows : bool or None, optional (default=None)
120: (8)                    If `None`, directed graphs draw arrowheads with
121: (8)                    `~matplotlib.patches.FancyArrowPatch`, while undirected graphs draw
edges
122: (8)                    via `~matplotlib.collections.LineCollection` for speed.
123: (8)                    If `True`, draw arrowheads with FancyArrowPatches (bendable and
stylish).
124: (8)                    If `False`, draw edges using LineCollection (linear and fast).
125: (8)                    For directed graphs, if True draw arrowheads.
126: (8)                    Note: Arrows will be the same color as edges.
127: (4)                arrowstyle : str (default='-\|>' for directed graphs)
128: (8)                    For directed graphs, choose the style of the arrowsheads.
129: (8)                    For undirected graphs default to '-'
130: (8)                    See `matplotlib.patches.ArrowStyle` for more options.
131: (4)                arrowsize : int or list (default=10)
132: (8)                    For directed graphs, choose the size of the arrow head's length and
133: (8)                    width. A list of values can be passed in to assign a different size
for arrow head's length and width.
134: (8)                    See `matplotlib.patches.FancyArrowPatch` for attribute
`mutation_scale`
135: (8)                    for more info.
136: (4)                with_labels :  bool (default=True)
137: (8)                    Set to True to draw labels on the nodes.
138: (4)                ax : Matplotlib Axes object, optional
139: (8)                    Draw the graph in the specified Matplotlib axes.
140: (4)                nodelist : list (default=list(G))
141: (8)                    Draw only specified nodes
142: (4)                edgelist : list (default=list(G.edges()))
143: (8)                    Draw only specified edges
144: (4)                node_size : scalar or array (default=300)
145: (8)                    Size of nodes.  If an array is specified it must be the
146: (8)                    same length as nodelist.
147: (4)                node_color : color or array of colors (default='#1f78b4')
148: (8)                    Node color. Can be a single color or a sequence of colors with the
same
149: (8)                    length as nodelist. Color can be string or rgb (or rgba) tuple of
150: (8)                    floats from 0-1. If numeric values are specified they will be
151: (8)                    mapped to colors using the cmap and vmin,vmax parameters. See
152: (8)                    matplotlib.scatter for more details.
153: (4)                node_shape :  string (default='o')
154: (8)                    The shape of the node.  Specification is as matplotlib.scatter
155: (8)                    marker, one of 'so^>v<dph8'.
```

```
    alpha : float or None (default=None)
        The node and edge transparency
    cmap : Matplotlib colormap, optional
        Colormap for mapping intensities of nodes
    vmin,vmax : float, optional
        Minimum and maximum for node colormap scaling
    linewidths : scalar or sequence (default=1.0)
        Line width of symbol border
    width : float or array of floats (default=1.0)
        Line width of edges
    edge_color : color or array of colors (default='k')
        Edge color. Can be a single color or a sequence of colors with the same

        length as edgelist. Color can be string or rgb (or rgba) tuple of
        floats from 0-1. If numeric values are specified they will be
        mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.
    edge_cmap : Matplotlib colormap, optional
        Colormap for mapping intensities of edges
    edge_vmin,edge_vmax : floats, optional
        Minimum and maximum for edge colormap scaling
    style : string (default=solid line)
        Edge line style e.g.: '-', '--', '-.', ':'
        or words like 'solid' or 'dashed'.
        (See `matplotlib.patches.FancyArrowPatch`: `linestyle`)
    labels : dictionary (default=None)
        Node labels in a dictionary of text labels keyed by node
    font_size : int (default=12 for nodes, 10 for edges)
        Font size for text labels
    font_color : color (default='k' black)
        Font color string. Color can be string or rgb (or rgba) tuple of
        floats from 0-1.
    font_weight : string (default='normal')
        Font weight
    font_family : string (default='sans-serif')
        Font family
    label : string, optional
        Label for graph legend
    hide_ticks : bool, optional
        Hide ticks of axes. When `True` (the default), ticks and ticklabels
        are removed from the axes. To set ticks and tick labels to the pyplot default,

        use ``hide_ticks=False``.
    kwds : optional keywords
        See networkx.draw_networkx_nodes(), networkx.draw_networkx_edges(), and

        networkx.draw_networkx_labels() for a description of optional keywords.
    Notes
    -----
    For directed graphs, arrows  are drawn at the head end.  Arrows can be
    turned off with keyword arrows=False.
    Examples
    --------
    >>> G = nx.dodecahedral_graph()
    >>> nx.draw(G)
    >>> nx.draw(G, pos=nx.spring_layout(G))  # use spring layout
    >>> import matplotlib.pyplot as plt
    >>> limits = plt.axis("off")  # turn off axis
    Also see the NetworkX drawing examples at
    https://networkx.org/documentation/latest/auto_examples/index.html
    See Also
    --------
    draw
    draw_networkx_nodes
    draw_networkx_edges
    draw_networkx_labels
    draw_networkx_edge_labels
    """
```

```
220: (4)                from inspect import signature
221: (4)                import matplotlib.pyplot as plt
222: (4)                valid_node_kwds = signature(draw_networkx_nodes).parameters.keys()
223: (4)                valid_edge_kwds = signature(draw_networkx_edges).parameters.keys()
224: (4)                valid_label_kwds = signature(draw_networkx_labels).parameters.keys()
225: (4)                valid_kwds = (valid_node_kwds | valid_edge_kwds | valid_label_kwds) - {
226: (8)                    "G",
227: (8)                    "pos",
228: (8)                    "arrows",
229: (8)                    "with_labels",
230: (4)                }
231: (4)                if any(k not in valid_kwds for k in kwds):
232: (8)                    invalid_args = ", ".join([k for k in kwds if k not in valid_kwds])
233: (8)                    raise ValueError(f"Received invalid argument(s): {invalid_args}")
234: (4)                node_kwds = {k: v for k, v in kwds.items() if k in valid_node_kwds}
235: (4)                edge_kwds = {k: v for k, v in kwds.items() if k in valid_edge_kwds}
236: (4)                label_kwds = {k: v for k, v in kwds.items() if k in valid_label_kwds}
237: (4)                if pos is None:
238: (8)                    pos = nx.drawing.spring_layout(G)  # default to spring layout
239: (4)                draw_networkx_nodes(G, pos, **node_kwds)
240: (4)                draw_networkx_edges(G, pos, arrows=arrows, **edge_kwds)
241: (4)                if with_labels:
242: (8)                    draw_networkx_labels(G, pos, **label_kwds)
243: (4)                plt.draw_if_interactive()
244: (0)            def draw_networkx_nodes(
245: (4)                G,
246: (4)                pos,
247: (4)                nodelist=None,
248: (4)                node_size=300,
249: (4)                node_color="#1f78b4",
250: (4)                node_shape="o",
251: (4)                alpha=None,
252: (4)                cmap=None,
253: (4)                vmin=None,
254: (4)                vmax=None,
255: (4)                ax=None,
256: (4)                linewidths=None,
257: (4)                edgecolors=None,
258: (4)                label=None,
259: (4)                margins=None,
260: (4)                hide_ticks=True,
261: (0)            ):
262: (4)                """Draw the nodes of the graph G.
263: (4)                This draws only the nodes of the graph G.
264: (4)                Parameters
265: (4)                ----------
266: (4)                G : graph
267: (8)                    A networkx graph
268: (4)                pos : dictionary
269: (8)                    A dictionary with nodes as keys and positions as values.
270: (8)                    Positions should be sequences of length 2.
271: (4)                ax : Matplotlib Axes object, optional
272: (8)                    Draw the graph in the specified Matplotlib axes.
273: (4)                nodelist : list (default list(G))
274: (8)                    Draw only specified nodes
275: (4)                node_size : scalar or array (default=300)
276: (8)                    Size of nodes.  If an array it must be the same length as nodelist.
277: (4)                node_color : color or array of colors (default='#1f78b4')
278: (8)                    Node color. Can be a single color or a sequence of colors with the
same
279: (8)                    length as nodelist. Color can be string or rgb (or rgba) tuple of
280: (8)                    floats from 0-1. If numeric values are specified they will be
281: (8)                    mapped to colors using the cmap and vmin,vmax parameters. See
282: (8)                    matplotlib.scatter for more details.
283: (4)                node_shape :  string (default='o')
284: (8)                    The shape of the node.  Specification is as matplotlib.scatter
285: (8)                    marker, one of 'so^>v<dph8'.
286: (4)                alpha : float or array of floats (default=None)
287: (8)                    The node transparency.  This can be a single alpha value,
```

```
288: (8)                    in which case it will be applied to all the nodes of color. Otherwise,
289: (8)                    if it is an array, the elements of alpha will be applied to the colors
290: (8)                    in order (cycling through alpha multiple times if necessary).
291: (4)                cmap : Matplotlib colormap (default=None)
292: (8)                    Colormap for mapping intensities of nodes
293: (4)                vmin,vmax : floats or None (default=None)
294: (8)                    Minimum and maximum for node colormap scaling
295: (4)                linewidths : [None | scalar | sequence] (default=1.0)
296: (8)                    Line width of symbol border
297: (4)                edgecolors : [None | scalar | sequence] (default = node_color)
298: (8)                    Colors of node borders. Can be a single color or a sequence of colors
with the
299: (8)                    same length as nodelist. Color can be string or rgb (or rgba) tuple of
floats
300: (8)                    from 0-1. If numeric values are specified they will be mapped to
colors
301: (8)                    using the cmap and vmin,vmax parameters. See
`~matplotlib.pyplot.scatter` for more details.
302: (4)                label : [None | string]
303: (8)                    Label for legend
304: (4)                margins : float or 2-tuple, optional
305: (8)                    Sets the padding for axis autoscaling. Increase margin to prevent
306: (8)                    clipping for nodes that are near the edges of an image. Values should
307: (8)                    be in the range ``[0, 1]``. See :meth:`matplotlib.axes.Axes.margins`
308: (8)                    for details. The default is `None`, which uses the Matplotlib default.
309: (4)                hide_ticks : bool, optional
310: (8)                    Hide ticks of axes. When `True` (the default), ticks and ticklabels
311: (8)                    are removed from the axes. To set ticks and tick labels to the pyplot
default,
312: (8)                    use ``hide_ticks=False``.
313: (4)                Returns
314: (4)                -------
315: (4)                matplotlib.collections.PathCollection
316: (8)                    `PathCollection` of the nodes.
317: (4)                Examples
318: (4)                --------
319: (4)                >>> G = nx.dodecahedral_graph()
320: (4)                >>> nodes = nx.draw_networkx_nodes(G, pos=nx.spring_layout(G))
321: (4)                Also see the NetworkX drawing examples at
322: (4)                https://networkx.org/documentation/latest/auto_examples/index.html
323: (4)                See Also
324: (4)                --------
325: (4)                draw
326: (4)                draw_networkx
327: (4)                draw_networkx_edges
328: (4)                draw_networkx_labels
329: (4)                draw_networkx_edge_labels
330: (4)                """
331: (4)                from collections.abc import Iterable
332: (4)                import matplotlib as mpl
333: (4)                import matplotlib.collections  # call as mpl.collections
334: (4)                import matplotlib.pyplot as plt
335: (4)                import numpy as np
336: (4)                if ax is None:
337: (8)                    ax = plt.gca()
338: (4)                if nodelist is None:
339: (8)                    nodelist = list(G)
340: (4)                if len(nodelist) == 0:  # empty nodelist, no drawing
341: (8)                    return mpl.collections.PathCollection(None)
342: (4)                try:
343: (8)                    xy = np.asarray([pos[v] for v in nodelist])
344: (4)                except KeyError as err:
345: (8)                    raise nx.NetworkXError(f"Node {err} has no position.") from err
346: (4)                if isinstance(alpha, Iterable):
347: (8)                    node_color = apply_alpha(node_color, alpha, nodelist, cmap, vmin,
vmax)
348: (8)                    alpha = None
349: (4)                node_collection = ax.scatter(
350: (8)                    xy[:, 0],
```

```
(8)                     xy[:, 1],
(8)                     s=node_size,
(8)                     c=node_color,
(8)                     marker=node_shape,
(8)                     cmap=cmap,
(8)                     vmin=vmin,
(8)                     vmax=vmax,
(8)                     alpha=alpha,
(8)                     linewidths=linewidths,
(8)                     edgecolors=edgecolors,
(8)                     label=label,
(4)                 )
(4)                 if hide_ticks:
(8)                     ax.tick_params(
(12)                        axis="both",
(12)                        which="both",
(12)                        bottom=False,
(12)                        left=False,
(12)                        labelbottom=False,
(12)                        labelleft=False,
(8)                     )
(4)                 if margins is not None:
(8)                     if isinstance(margins, Iterable):
(12)                        ax.margins(*margins)
(8)                     else:
(12)                        ax.margins(margins)
(4)                 node_collection.set_zorder(2)
(4)                 return node_collection
(0)             class FancyArrowFactory:
(4)                 """Draw arrows with `matplotlib.patches.FancyarrowPatch`"""
(4)                 class ConnectionStyleFactory:
(8)                     def __init__(self, connectionstyles, selfloop_height, ax=None):
(12)                        import matplotlib as mpl
(12)                        import matplotlib.path  # call as mpl.path
(12)                        import numpy as np
(12)                        self.ax = ax
(12)                        self.mpl = mpl
(12)                        self.np = np
(12)                        self.base_connection_styles = [
(16)                            mpl.patches.ConnectionStyle(cs) for cs in connectionstyles
(12)                        ]
(12)                        self.n = len(self.base_connection_styles)
(12)                        self.selfloop_height = selfloop_height
(8)                     def curved(self, edge_index):
(12)                        return self.base_connection_styles[edge_index % self.n]
(8)                     def self_loop(self, edge_index):
(12)                        def self_loop_connection(posA, posB, *args, **kwargs):
(16)                            if not self.np.all(posA == posB):
(20)                                raise nx.NetworkXError(
(24)                                    "`self_loop` connection style method"
(24)                                    "is only to be used for self-loops"
(20)                                )
(16)                            data_loc = self.ax.transData.inverted().transform(posA)
(16)                            v_shift = 0.1 * self.selfloop_height
(16)                            h_shift = v_shift * 0.5
(16)                            path = self.np.asarray(
(20)                                [
(24)                                    [0, v_shift],
(24)                                    [h_shift, v_shift],
(24)                                    [h_shift, 0],
(24)                                    [0, 0],
(24)                                    [-h_shift, 0],
(24)                                    [-h_shift, v_shift],
(24)                                    [0, v_shift],
(20)                                ]
(16)                            )
(16)                            if edge_index % 4:
(20)                                x, y = path.T
(20)                                for _ in range(edge_index % 4):
```

```
420: (24)                                x, y = y, -x
421: (20)                            path = self.np.array([x, y]).T
422: (16)                        return self.mpl.path.Path(
423: (20)                            self.ax.transData.transform(data_loc + path), [1, 4, 4, 4,
4, 4, 4]
424: (16)                        )
425: (12)                    return self_loop_connection
426: (4)                def __init__(
427: (8)                    self,
428: (8)                    edge_pos,
429: (8)                    edgelist,
430: (8)                    nodelist,
431: (8)                    edge_indices,
432: (8)                    node_size,
433: (8)                    selfloop_height,
434: (8)                    connectionstyle="arc3",
435: (8)                    node_shape="o",
436: (8)                    arrowstyle="-",
437: (8)                    arrowsize=10,
438: (8)                    edge_color="k",
439: (8)                    alpha=None,
440: (8)                    linewidth=1.0,
441: (8)                    style="solid",
442: (8)                    min_source_margin=0,
443: (8)                    min_target_margin=0,
444: (8)                    ax=None,
445: (4)                ):
446: (8)                    import matplotlib as mpl
447: (8)                    import matplotlib.patches  # call as mpl.patches
448: (8)                    import matplotlib.pyplot as plt
449: (8)                    import numpy as np
450: (8)                    if isinstance(connectionstyle, str):
451: (12)                    connectionstyle = [connectionstyle]
452: (8)                    elif np.iterable(connectionstyle):
453: (12)                    connectionstyle = list(connectionstyle)
454: (8)                    else:
455: (12)                    msg = "ConnectionStyleFactory arg `connectionstyle` must be str or
iterable"
456: (12)                    raise nx.NetworkXError(msg)
457: (8)                    self.ax = ax
458: (8)                    self.mpl = mpl
459: (8)                    self.np = np
460: (8)                    self.edge_pos = edge_pos
461: (8)                    self.edgelist = edgelist
462: (8)                    self.nodelist = nodelist
463: (8)                    self.node_shape = node_shape
464: (8)                    self.min_source_margin = min_source_margin
465: (8)                    self.min_target_margin = min_target_margin
466: (8)                    self.edge_indices = edge_indices
467: (8)                    self.node_size = node_size
468: (8)                    self.connectionstyle_factory = self.ConnectionStyleFactory(
469: (12)                    connectionstyle, selfloop_height, ax
470: (8)                    )
471: (8)                    self.arrowstyle = arrowstyle
472: (8)                    self.arrowsize = arrowsize
473: (8)                    self.arrow_colors =
mpl.colors.colorConverter.to_rgba_array(edge_color, alpha)
474: (8)                    self.linewidth = linewidth
475: (8)                    self.style = style
476: (8)                    if isinstance(arrowsize, list) and len(arrowsize) != len(edge_pos):
477: (12)                    raise ValueError("arrowsize should have the same length as
edgelist")
478: (4)                def __call__(self, i):
479: (8)                    (x1, y1), (x2, y2) = self.edge_pos[i]
480: (8)                    shrink_source = 0  # space from source to tail
481: (8)                    shrink_target = 0  # space from  head to target
482: (8)                    if self.np.iterable(self.node_size):  # many node sizes
483: (12)                    source, target = self.edgelist[i][:2]
484: (12)                    source_node_size = self.node_size[self.nodelist.index(source)]
```

```
485: (12)                        target_node_size = self.node_size[self.nodelist.index(target)]
486: (12)                        shrink_source = self.to_marker_edge(source_node_size,
self.node_shape)
487: (12)                        shrink_target = self.to_marker_edge(target_node_size,
self.node_shape)
488: (8)                    else:
489: (12)                        shrink_source = self.to_marker_edge(self.node_size,
self.node_shape)
490: (12)                        shrink_target = shrink_source
491: (8)                    shrink_source = max(shrink_source, self.min_source_margin)
492: (8)                    shrink_target = max(shrink_target, self.min_target_margin)
493: (8)                    if isinstance(self.arrowsize, list):
494: (12)                        mutation_scale = self.arrowsize[i]
495: (8)                    else:
496: (12)                        mutation_scale = self.arrowsize
497: (8)                    if len(self.arrow_colors) > i:
498: (12)                        arrow_color = self.arrow_colors[i]
499: (8)                    elif len(self.arrow_colors) == 1:
500: (12)                        arrow_color = self.arrow_colors[0]
501: (8)                    else:  # Cycle through colors
502: (12)                        arrow_color = self.arrow_colors[i % len(self.arrow_colors)]
503: (8)                    if self.np.iterable(self.linewidth):
504: (12)                        if len(self.linewidth) > i:
505: (16)                            linewidth = self.linewidth[i]
506: (12)                        else:
507: (16)                            linewidth = self.linewidth[i % len(self.linewidth)]
508: (8)                    else:
509: (12)                        linewidth = self.linewidth
510: (8)                    if (
511: (12)                        self.np.iterable(self.style)
512: (12)                        and not isinstance(self.style, str)
513: (12)                        and not isinstance(self.style, tuple)
514: (8)                    ):
515: (12)                        if len(self.style) > i:
516: (16)                            linestyle = self.style[i]
517: (12)                        else:  # Cycle through styles
518: (16)                            linestyle = self.style[i % len(self.style)]
519: (8)                    else:
520: (12)                        linestyle = self.style
521: (8)                    if x1 == x2 and y1 == y2:
522: (12)                        connectionstyle = self.connectionstyle_factory.self_loop(
523: (16)                            self.edge_indices[i]
524: (12)                        )
525: (8)                    else:
526: (12)                        connectionstyle =
self.connectionstyle_factory.curved(self.edge_indices[i])
527: (8)                    return self.mpl.patches.FancyArrowPatch(
528: (12)                        (x1, y1),
529: (12)                        (x2, y2),
530: (12)                        arrowstyle=self.arrowstyle,
531: (12)                        shrinkA=shrink_source,
532: (12)                        shrinkB=shrink_target,
533: (12)                        mutation_scale=mutation_scale,
534: (12)                        color=arrow_color,
535: (12)                        linewidth=linewidth,
536: (12)                        connectionstyle=connectionstyle,
537: (12)                        linestyle=linestyle,
538: (12)                        zorder=1,  # arrows go behind nodes
539: (8)                    )
540: (4)                def to_marker_edge(self, marker_size, marker):
541: (8)                    if marker in "s^>v<d":  # `large` markers need extra space
542: (12)                        return self.np.sqrt(2 * marker_size) / 2
543: (8)                    else:
544: (12)                        return self.np.sqrt(marker_size) / 2
545: (0)            def draw_networkx_edges(
546: (4)                G,
547: (4)                pos,
548: (4)                edgelist=None,
549: (4)                width=1.0,
```

```
    edge_color="k",
    style="solid",
    alpha=None,
    arrowstyle=None,
    arrowsize=10,
    edge_cmap=None,
    edge_vmin=None,
    edge_vmax=None,
    ax=None,
    arrows=None,
    label=None,
    node_size=300,
    nodelist=None,
    node_shape="o",
    connectionstyle="arc3",
    min_source_margin=0,
    min_target_margin=0,
    hide_ticks=True,
):
    r"""Draw the edges of the graph G.
    This draws only the edges of the graph G.
    Parameters
    ----------
    G : graph
        A networkx graph
    pos : dictionary
        A dictionary with nodes as keys and positions as values.
        Positions should be sequences of length 2.
    edgelist : collection of edge tuples (default=G.edges())
        Draw only specified edges
    width : float or array of floats (default=1.0)
        Line width of edges
    edge_color : color or array of colors (default='k')
        Edge color. Can be a single color or a sequence of colors with the same
        length as edgelist. Color can be string or rgb (or rgba) tuple of
        floats from 0-1. If numeric values are specified they will be
        mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.
    style : string or array of strings (default='solid')
        Edge line style e.g.: '-', '--', '-.', ':'
        or words like 'solid' or 'dashed'.
        Can be a single style or a sequence of styles with the same
        length as the edge list.
        If less styles than edges are given the styles will cycle.
        If more styles than edges are given the styles will be used sequentially
        and not be exhausted.
        Also, `(offset, onoffseq)` tuples can be used as style instead of a strings.
        (See `matplotlib.patches.FancyArrowPatch`: `linestyle`)
    alpha : float or array of floats (default=None)
        The edge transparency.  This can be a single alpha value,
        in which case it will be applied to all specified edges. Otherwise,
        if it is an array, the elements of alpha will be applied to the colors
        in order (cycling through alpha multiple times if necessary).
    edge_cmap : Matplotlib colormap, optional
        Colormap for mapping intensities of edges
    edge_vmin,edge_vmax : floats, optional
        Minimum and maximum for edge colormap scaling
    ax : Matplotlib Axes object, optional
        Draw the graph in the specified Matplotlib axes.
    arrows : bool or None, optional (default=None)
        If `None`, directed graphs draw arrowheads with
        `~matplotlib.patches.FancyArrowPatch`, while undirected graphs draw edges
        via `~matplotlib.collections.LineCollection` for speed.
        If `True`, draw arrowheads with FancyArrowPatches (bendable and stylish).
```

```
(8)                 If `False`, draw edges using LineCollection (linear and fast).
(8)                 Note: Arrowheads will be the same color as edges.
(4)             arrowstyle : str (default='-\|>' for directed graphs)
(8)                 For directed graphs and `arrows==True` defaults to '-\|>',
(8)                 For undirected graphs default to '-'.
(8)                 See `matplotlib.patches.ArrowStyle` for more options.
(4)             arrowsize : int (default=10)
(8)                 For directed graphs, choose the size of the arrow head's length and
(8)                 width. See `matplotlib.patches.FancyArrowPatch` for attribute
(8)                 `mutation_scale` for more info.
(4)             connectionstyle : string or iterable of strings (default="arc3")
(8)                 Pass the connectionstyle parameter to create curved arc of rounding
(8)                 radius rad. For example, connectionstyle='arc3,rad=0.2'.
(8)                 See `matplotlib.patches.ConnectionStyle` and
(8)                 `matplotlib.patches.FancyArrowPatch` for more info.
(8)                 If Iterable, index indicates i'th edge key of MultiGraph
(4)             node_size : scalar or array (default=300)
(8)                 Size of nodes. Though the nodes are not drawn with this function, the
(8)                 node size is used in determining edge positioning.
(4)             nodelist : list, optional (default=G.nodes())
(7)                This provides the node order for the `node_size` array (if it is an
array).
(4)             node_shape :  string (default='o')
(8)                 The marker used for nodes, used in determining edge positioning.
(8)                 Specification is as a `matplotlib.markers` marker, e.g. one of
'so^>v<dph8'.
(4)             label : None or string
(8)                 Label for legend
(4)             min_source_margin : int (default=0)
(8)                 The minimum margin (gap) at the beginning of the edge at the source.
(4)             min_target_margin : int (default=0)
(8)                 The minimum margin (gap) at the end of the edge at the target.
(4)             hide_ticks : bool, optional
(8)                 Hide ticks of axes. When `True` (the default), ticks and ticklabels
(8)                 are removed from the axes. To set ticks and tick labels to the pyplot
default,
(8)                 use ``hide_ticks=False``.
(4)             Returns
(4)             -------
(5)              matplotlib.collections.LineCollection or a list of
matplotlib.patches.FancyArrowPatch
(8)                 If ``arrows=True``, a list of FancyArrowPatches is returned.
(8)                 If ``arrows=False``, a LineCollection is returned.
(8)                 If ``arrows=None`` (the default), then a LineCollection is returned if
(8)                 `G` is undirected, otherwise returns a list of FancyArrowPatches.
(4)             Notes
(4)             -----
(4)             For directed graphs, arrows are drawn at the head end.  Arrows can be
(4)             turned off with keyword arrows=False or by passing an arrowstyle without
(4)             an arrow on the end.
(4)             Be sure to include `node_size` as a keyword argument; arrows are
(4)             drawn considering the size of nodes.
(4)             Self-loops are always drawn with `~matplotlib.patches.FancyArrowPatch`
(4)             regardless of the value of `arrows` or whether `G` is directed.
(4)             When ``arrows=False`` or ``arrows=None`` and `G` is undirected, the
(4)             FancyArrowPatches corresponding to the self-loops are not explicitly
(4)             returned. They should instead be accessed via the ``Axes.patches``
(4)             attribute (see examples).
(4)             Examples
(4)             --------
(4)             >>> G = nx.dodecahedral_graph()
(4)             >>> edges = nx.draw_networkx_edges(G, pos=nx.spring_layout(G))
(4)             >>> G = nx.DiGraph()
(4)             >>> G.add_edges_from([(1, 2), (1, 3), (2, 3)])
(4)             >>> arcs = nx.draw_networkx_edges(G, pos=nx.spring_layout(G))
(4)             >>> alphas = [0.3, 0.4, 0.5]
(4)             >>> for i, arc in enumerate(arcs):  # change alpha values of arcs
(4)             ...     arc.set_alpha(alphas[i])
(4)             The FancyArrowPatches corresponding to self-loops are not always
```

```
    returned, but can always be accessed via the ``patches`` attribute of the
    `matplotlib.Axes` object.
    >>> import matplotlib.pyplot as plt
    >>> fig, ax = plt.subplots()
    >>> G = nx.Graph([(0, 1), (0, 0)])  # Self-loop at node 0
    >>> edge_collection = nx.draw_networkx_edges(G, pos=nx.circular_layout(G), ax=ax)
    >>> self_loop_fap = ax.patches[0]
    Also see the NetworkX drawing examples at
    https://networkx.org/documentation/latest/auto_examples/index.html
    See Also
    --------
    draw
    draw_networkx
    draw_networkx_nodes
    draw_networkx_labels
    draw_networkx_edge_labels
    """
    import warnings
    import matplotlib as mpl
    import matplotlib.collections  # call as mpl.collections
    import matplotlib.colors  # call as mpl.colors
    import matplotlib.pyplot as plt
    import numpy as np
    if arrows is None:
        use_linecollection = not (G.is_directed() or G.is_multigraph())
    else:
        if not isinstance(arrows, bool):
            raise TypeError("Argument `arrows` must be of type bool or None")
        use_linecollection = not arrows
    if isinstance(connectionstyle, str):
        connectionstyle = [connectionstyle]
    elif np.iterable(connectionstyle):
        connectionstyle = list(connectionstyle)
    else:
        msg = "draw_networkx_edges arg `connectionstyle` must be str or iterable"
        raise nx.NetworkXError(msg)
    if use_linecollection:
        msg = (
            "\n\nThe {0} keyword argument is not applicable when drawing edges\n"
            "with LineCollection.\n\n"
            "To make this warning go away, either specify `arrows=True` to\n"
            "force FancyArrowPatches or use the default values.\n"
            "Note that using FancyArrowPatches may be slow for large graphs.\n"
        )
        if arrowstyle is not None:
            warnings.warn(msg.format("arrowstyle"), category=UserWarning, stacklevel=2)
        if arrowsize != 10:
            warnings.warn(msg.format("arrowsize"), category=UserWarning, stacklevel=2)
        if min_source_margin != 0:
            warnings.warn(
                msg.format("min_source_margin"), category=UserWarning, stacklevel=2
            )
        if min_target_margin != 0:
            warnings.warn(
                msg.format("min_target_margin"), category=UserWarning, stacklevel=2
            )
        if any(cs != "arc3" for cs in connectionstyle):
            warnings.warn(
                msg.format("connectionstyle"), category=UserWarning, stacklevel=2
            )
```

```
(4)                 if arrowstyle is None:
(8)                     arrowstyle = "-|>" if G.is_directed() else "-"
(4)                 if ax is None:
(8)                     ax = plt.gca()
(4)                 if edgelist is None:
(8)                     edgelist = list(G.edges)  # (u, v, k) for multigraph (u, v) otherwise
(4)                 if len(edgelist):
(8)                     if G.is_multigraph():
(12)                        key_count = collections.defaultdict(lambda: itertools.count(0))
(12)                        edge_indices = [next(key_count[tuple(e[:2])]) for e in edgelist]
(8)                     else:
(12)                        edge_indices = [0] * len(edgelist)
(4)                 else:  # no edges!
(8)                     return []
(4)                 if nodelist is None:
(8)                     nodelist = list(G.nodes())
(4)                 if edge_color is None:
(8)                     edge_color = "k"
(4)                 edge_pos = np.asarray([(pos[e[0]], pos[e[1]]) for e in edgelist])
(4)                 if (
(8)                     np.iterable(edge_color)
(8)                     and (len(edge_color) == len(edge_pos))
(8)                     and np.all([isinstance(c, Number) for c in edge_color])
(4)                 ):
(8)                     if edge_cmap is not None:
(12)                        assert isinstance(edge_cmap, mpl.colors.Colormap)
(8)                     else:
(12)                        edge_cmap = plt.get_cmap()
(8)                     if edge_vmin is None:
(12)                        edge_vmin = min(edge_color)
(8)                     if edge_vmax is None:
(12)                        edge_vmax = max(edge_color)
(8)                     color_normal = mpl.colors.Normalize(vmin=edge_vmin, vmax=edge_vmax)
(8)                     edge_color = [edge_cmap(color_normal(e)) for e in edge_color]
(4)                 minx = np.amin(np.ravel(edge_pos[:, :, 0]))
(4)                 maxx = np.amax(np.ravel(edge_pos[:, :, 0]))
(4)                 miny = np.amin(np.ravel(edge_pos[:, :, 1]))
(4)                 maxy = np.amax(np.ravel(edge_pos[:, :, 1]))
(4)                 w = maxx - minx
(4)                 h = maxy - miny
(4)                 selfloop_height = h if h != 0 else 0.005 * np.array(node_size).max()
(4)                 fancy_arrow_factory = FancyArrowFactory(
(8)                     edge_pos,
(8)                     edgelist,
(8)                     nodelist,
(8)                     edge_indices,
(8)                     node_size,
(8)                     selfloop_height,
(8)                     connectionstyle,
(8)                     node_shape,
(8)                     arrowstyle,
(8)                     arrowsize,
(8)                     edge_color,
(8)                     alpha,
(8)                     width,
(8)                     style,
(8)                     min_source_margin,
(8)                     min_target_margin,
(8)                     ax=ax,
(4)                 )
(4)                 if use_linecollection:
(8)                     edge_collection = mpl.collections.LineCollection(
(12)                        edge_pos,
(12)                        colors=edge_color,
(12)                        linewidths=width,
(12)                        antialiaseds=(1,),
(12)                        linestyle=style,
(12)                        alpha=alpha,
(8)                     )
```

```
807: (8)                    edge_collection.set_cmap(edge_cmap)
808: (8)                    edge_collection.set_clim(edge_vmin, edge_vmax)
809: (8)                    edge_collection.set_zorder(1)  # edges go behind nodes
810: (8)                    edge_collection.set_label(label)
811: (8)                    ax.add_collection(edge_collection)
812: (8)                    edge_viz_obj = edge_collection
813: (8)                    selfloops_to_draw = [loop for loop in nx.selfloop_edges(G) if loop in
edgelist]
814: (8)                    if selfloops_to_draw:
815: (12)                       edgelist_tuple = list(map(tuple, edgelist))
816: (12)                       arrow_collection = []
817: (12)                       for loop in selfloops_to_draw:
818: (16)                           i = edgelist_tuple.index(loop)
819: (16)                           arrow = fancy_arrow_factory(i)
820: (16)                           arrow_collection.append(arrow)
821: (16)                           ax.add_patch(arrow)
822: (4)                else:
823: (8)                    edge_viz_obj = []
824: (8)                    for i in range(len(edgelist)):
825: (12)                       arrow = fancy_arrow_factory(i)
826: (12)                       ax.add_patch(arrow)
827: (12)                       edge_viz_obj.append(arrow)
828: (4)                padx, pady = 0.05 * w, 0.05 * h
829: (4)                corners = (minx - padx, miny - pady), (maxx + padx, maxy + pady)
830: (4)                ax.update_datalim(corners)
831: (4)                ax.autoscale_view()
832: (4)                if hide_ticks:
833: (8)                    ax.tick_params(
834: (12)                       axis="both",
835: (12)                       which="both",
836: (12)                       bottom=False,
837: (12)                       left=False,
838: (12)                       labelbottom=False,
839: (12)                       labelleft=False,
840: (8)                    )
841: (4)                return edge_viz_obj
842: (0)            def draw_networkx_labels(
843: (4)                G,
844: (4)                pos,
845: (4)                labels=None,
846: (4)                font_size=12,
847: (4)                font_color="k",
848: (4)                font_family="sans-serif",
849: (4)                font_weight="normal",
850: (4)                alpha=None,
851: (4)                bbox=None,
852: (4)                horizontalalignment="center",
853: (4)                verticalalignment="center",
854: (4)                ax=None,
855: (4)                clip_on=True,
856: (4)                hide_ticks=True,
857: (0)            ):
858: (4)                """Draw node labels on the graph G.
859: (4)                Parameters
860: (4)                ----------
861: (4)                G : graph
862: (8)                    A networkx graph
863: (4)                pos : dictionary
864: (8)                    A dictionary with nodes as keys and positions as values.
865: (8)                    Positions should be sequences of length 2.
866: (4)                labels : dictionary (default={n: n for n in G})
867: (8)                    Node labels in a dictionary of text labels keyed by node.
868: (8)                    Node-keys in labels should appear as keys in `pos`.
869: (8)                    If needed use: `{n:lab for n,lab in labels.items() if n in pos}`
870: (4)                font_size : int (default=12)
871: (8)                    Font size for text labels
872: (4)                font_color : color (default='k' black)
873: (8)                    Font color string. Color can be string or rgb (or rgba) tuple of
874: (8)                    floats from 0-1.
```

```
    font_weight : string (default='normal')
        Font weight
    font_family : string (default='sans-serif')
        Font family
    alpha : float or None (default=None)
        The text transparency
    bbox : Matplotlib bbox, (default is Matplotlib's ax.text default)
        Specify text box properties (e.g. shape, color etc.) for node labels.
    horizontalalignment : string (default='center')
        Horizontal alignment {'center', 'right', 'left'}
    verticalalignment : string (default='center')
        Vertical alignment {'center', 'top', 'bottom', 'baseline',
'center_baseline'}
    ax : Matplotlib Axes object, optional
        Draw the graph in the specified Matplotlib axes.
    clip_on : bool (default=True)
        Turn on clipping of node labels at axis boundaries
    hide_ticks : bool, optional
        Hide ticks of axes. When `True` (the default), ticks and ticklabels
        are removed from the axes. To set ticks and tick labels to the pyplot
default,
        use ``hide_ticks=False``.
    Returns
    -------
    dict
        `dict` of labels keyed on the nodes
    Examples
    --------
    >>> G = nx.dodecahedral_graph()
    >>> labels = nx.draw_networkx_labels(G, pos=nx.spring_layout(G))
    Also see the NetworkX drawing examples at
    https://networkx.org/documentation/latest/auto_examples/index.html
    See Also
    --------
    draw
    draw_networkx
    draw_networkx_nodes
    draw_networkx_edges
    draw_networkx_edge_labels
    """
    import matplotlib.pyplot as plt
    if ax is None:
        ax = plt.gca()
    if labels is None:
        labels = {n: n for n in G.nodes()}
    text_items = {}  # there is no text collection so we'll fake one
    for n, label in labels.items():
        (x, y) = pos[n]
        if not isinstance(label, str):
            label = str(label)  # this makes "1" and 1 labeled the same
        t = ax.text(
            x,
            y,
            label,
            size=font_size,
            color=font_color,
            family=font_family,
            weight=font_weight,
            alpha=alpha,
            horizontalalignment=horizontalalignment,
            verticalalignment=verticalalignment,
            transform=ax.transData,
            bbox=bbox,
            clip_on=clip_on,
        )
        text_items[n] = t
    if hide_ticks:
        ax.tick_params(
            axis="both",
```

```
(12)                        which="both",
(12)                        bottom=False,
(12)                        left=False,
(12)                        labelbottom=False,
(12)                        labelleft=False,
(8)                     )
(4)                 return text_items
(0)             def draw_networkx_edge_labels(
(4)                 G,
(4)                 pos,
(4)                 edge_labels=None,
(4)                 label_pos=0.5,
(4)                 font_size=10,
(4)                 font_color="k",
(4)                 font_family="sans-serif",
(4)                 font_weight="normal",
(4)                 alpha=None,
(4)                 bbox=None,
(4)                 horizontalalignment="center",
(4)                 verticalalignment="center",
(4)                 ax=None,
(4)                 rotate=True,
(4)                 clip_on=True,
(4)                 node_size=300,
(4)                 nodelist=None,
(4)                 connectionstyle="arc3",
(4)                 hide_ticks=True,
(0)             ):
(4)                 """Draw edge labels.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     A networkx graph
(4)                 pos : dictionary
(8)                     A dictionary with nodes as keys and positions as values.
(8)                     Positions should be sequences of length 2.
(4)                 edge_labels : dictionary (default=None)
(8)                     Edge labels in a dictionary of labels keyed by edge two-tuple.
(8)                     Only labels for the keys in the dictionary are drawn.
(4)                 label_pos : float (default=0.5)
(8)                     Position of edge label along edge (0=head, 0.5=center, 1=tail)
(4)                 font_size : int (default=10)
(8)                     Font size for text labels
(4)                 font_color : color (default='k' black)
(8)                     Font color string. Color can be string or rgb (or rgba) tuple of
(8)                     floats from 0-1.
(4)                 font_weight : string (default='normal')
(8)                     Font weight
(4)                 font_family : string (default='sans-serif')
(8)                     Font family
(4)                 alpha : float or None (default=None)
(8)                     The text transparency
(4)                 bbox : Matplotlib bbox, optional
(8)                     Specify text box properties (e.g. shape, color etc.) for edge labels.
(8)                     Default is {boxstyle='round', ec=(1.0, 1.0, 1.0), fc=(1.0, 1.0, 1.0)}.
(4)                 horizontalalignment : string (default='center')
(8)                     Horizontal alignment {'center', 'right', 'left'}
(4)                 verticalalignment : string (default='center')
(8)                     Vertical alignment {'center', 'top', 'bottom', 'baseline',
'center_baseline'}
(4)                 ax : Matplotlib Axes object, optional
(8)                     Draw the graph in the specified Matplotlib axes.
(4)                 rotate : bool (default=True)
(8)                     Rotate edge labels to lie parallel to edges
(4)                 clip_on : bool (default=True)
(8)                     Turn on clipping of edge labels at axis boundaries
(4)                 node_size : scalar or array (default=300)
(8)                     Size of nodes.  If an array it must be the same length as nodelist.
(4)                 nodelist : list, optional (default=G.nodes())
```

```
(7)                 This provides the node order for the `node_size` array (if it is an array).
(4)             connectionstyle : string or iterable of strings (default="arc3")
(8)                 Pass the connectionstyle parameter to create curved arc of rounding
(8)                 radius rad. For example, connectionstyle='arc3,rad=0.2'.
(8)                 See `matplotlib.patches.ConnectionStyle` and
(8)                 `matplotlib.patches.FancyArrowPatch` for more info.
(8)                 If Iterable, index indicates i'th edge key of MultiGraph
(4)             hide_ticks : bool, optional
(8)                 Hide ticks of axes. When `True` (the default), ticks and ticklabels
(8)                 are removed from the axes. To set ticks and tick labels to the pyplot default,
(8)                 use ``hide_ticks=False``.
(4)             Returns
(4)             -------
(4)             dict
(8)                 `dict` of labels keyed by edge
(4)             Examples
(4)             --------
(4)             >>> G = nx.dodecahedral_graph()
(4)             >>> edge_labels = nx.draw_networkx_edge_labels(G, pos=nx.spring_layout(G))
(4)             Also see the NetworkX drawing examples at
(4)             https://networkx.org/documentation/latest/auto_examples/index.html
(4)             See Also
(4)             --------
(4)             draw
(4)             draw_networkx
(4)             draw_networkx_nodes
(4)             draw_networkx_edges
(4)             draw_networkx_labels
(4)             """
(4)             import matplotlib as mpl
(4)             import matplotlib.pyplot as plt
(4)             import numpy as np
(4)             class CurvedArrowText(mpl.text.Text):
(8)                 def __init__(
(12)                    self,
(12)                    arrow,
(12)                    *args,
(12)                    label_pos=0.5,
(12)                    labels_horizontal=False,
(12)                    ax=None,
(12)                    **kwargs,
(8)                 ):
(12)                    self.arrow = arrow
(12)                    self.label_pos = label_pos
(12)                    self.labels_horizontal = labels_horizontal
(12)                    if ax is None:
(16)                        ax = plt.gca()
(12)                    self.ax = ax
(12)                    self.x, self.y, self.angle = self._update_text_pos_angle(arrow)
(12)                    super().__init__(self.x, self.y, *args, rotation=self.angle, **kwargs)
(12)                    self.ax.add_artist(self)
(8)                 def _get_arrow_path_disp(self, arrow):
(12)                    """
(12)                    This is part of FancyArrowPatch._get_path_in_displaycoord
(12)                    It omits the second part of the method where path is converted
(16)                        to polygon based on width
(12)                    The transform is taken from ax, not the object, as the object
(16)                        has not been added yet, and doesn't have transform
(12)                    """
(12)                    dpi_cor = arrow._dpi_cor
(12)                    trans_data = self.ax.transData
(12)                    if arrow._posA_posB is not None:
(16)                        posA = arrow._convert_xy_units(arrow._posA_posB[0])
(16)                        posB = arrow._convert_xy_units(arrow._posA_posB[1])
(16)                        (posA, posB) = trans_data.transform((posA, posB))
(16)                        _path = arrow.get_connectionstyle()(
```

```
1076: (20)                                posA,
1077: (20)                                posB,
1078: (20)                                patchA=arrow.patchA,
1079: (20)                                patchB=arrow.patchB,
1080: (20)                                shrinkA=arrow.shrinkA * dpi_cor,
1081: (20)                                shrinkB=arrow.shrinkB * dpi_cor,
1082: (16)                            )
1083: (12)                        else:
1084: (16)                            _path = trans_data.transform_path(arrow._path_original)
1085: (12)                        return _path
1086: (8)                    def _update_text_pos_angle(self, arrow):
1087: (12)                        path_disp = self._get_arrow_path_disp(arrow)
1088: (12)                        (x1, y1), (cx, cy), (x2, y2) = path_disp.vertices
1089: (12)                        t = self.label_pos
1090: (12)                        tt = 1 - t
1091: (12)                        x = tt**2 * x1 + 2 * t * tt * cx + t**2 * x2
1092: (12)                        y = tt**2 * y1 + 2 * t * tt * cy + t**2 * y2
1093: (12)                        if self.labels_horizontal:
1094: (16)                            angle = 0
1095: (12)                        else:
1096: (16)                            change_x = 2 * tt * (cx - x1) + 2 * t * (x2 - cx)
1097: (16)                            change_y = 2 * tt * (cy - y1) + 2 * t * (y2 - cy)
1098: (16)                            angle = (np.arctan2(change_y, change_x) / (2 * np.pi)) * 360
1099: (16)                            if angle > 90:
1100: (20)                                angle -= 180
1101: (16)                            if angle < -90:
1102: (20)                                angle += 180
1103: (12)                        (x, y) = self.ax.transData.inverted().transform((x, y))
1104: (12)                        return x, y, angle
1105: (8)                    def draw(self, renderer):
1106: (12)                        self.x, self.y, self.angle =
self._update_text_pos_angle(self.arrow)
1107: (12)                        self.set_position((self.x, self.y))
1108: (12)                        self.set_rotation(self.angle)
1109: (12)                        super().draw(renderer)
1110: (4)                if bbox is None:
1111: (8)                    bbox = {"boxstyle": "round", "ec": (1.0, 1.0, 1.0), "fc": (1.0, 1.0,
1.0)}
1112: (4)                if isinstance(connectionstyle, str):
1113: (8)                    connectionstyle = [connectionstyle]
1114: (4)                elif np.iterable(connectionstyle):
1115: (8)                    connectionstyle = list(connectionstyle)
1116: (4)                else:
1117: (8)                    raise nx.NetworkXError(
1118: (12)                        "draw_networkx_edges arg `connectionstyle` must be"
1119: (12)                        "string or iterable of strings"
1120: (8)                    )
1121: (4)                if ax is None:
1122: (8)                    ax = plt.gca()
1123: (4)                if edge_labels is None:
1124: (8)                    kwds = {"keys": True} if G.is_multigraph() else {}
1125: (8)                    edge_labels = {tuple(edge): d for *edge, d in G.edges(data=True,
**kwds)}
1126: (4)                if not edge_labels:
1127: (8)                    return {}
1128: (4)                edgelist, labels = zip(*edge_labels.items())
1129: (4)                if nodelist is None:
1130: (8)                    nodelist = list(G.nodes())
1131: (4)                edge_pos = np.asarray([(pos[e[0]], pos[e[1]]) for e in edgelist])
1132: (4)                if G.is_multigraph():
1133: (8)                    key_count = collections.defaultdict(lambda: itertools.count(0))
1134: (8)                    edge_indices = [next(key_count[tuple(e[:2])]) for e in edgelist]
1135: (4)                else:
1136: (8)                    edge_indices = [0] * len(edgelist)
1137: (4)                h = 0
1138: (4)                if edge_labels:
1139: (8)                    miny = np.amin(np.ravel(edge_pos[:, :, 1]))
1140: (8)                    maxy = np.amax(np.ravel(edge_pos[:, :, 1]))
1141: (8)                    h = maxy - miny
```

```
(4)                 selfloop_height = h if h != 0 else 0.005 * np.array(node_size).max()
(4)                 fancy_arrow_factory = FancyArrowFactory(
(8)                     edge_pos,
(8)                     edgelist,
(8)                     nodelist,
(8)                     edge_indices,
(8)                     node_size,
(8)                     selfloop_height,
(8)                     connectionstyle,
(8)                     ax=ax,
(4)                 )
(4)                 text_items = {}
(4)                 for i, (edge, label) in enumerate(zip(edgelist, labels)):
(8)                     if not isinstance(label, str):
(12)                        label = str(label)  # this makes "1" and 1 labeled the same
(8)                     n1, n2 = edge[:2]
(8)                     arrow = fancy_arrow_factory(i)
(8)                     if n1 == n2:
(12)                        connectionstyle_obj = arrow.get_connectionstyle()
(12)                        posA = ax.transData.transform(pos[n1])
(12)                        path_disp = connectionstyle_obj(posA, posA)
(12)                        path_data = ax.transData.inverted().transform_path(path_disp)
(12)                        x, y = path_data.vertices[0]
(12)                        text_items[edge] = ax.text(
(16)                            x,
(16)                            y,
(16)                            label,
(16)                            size=font_size,
(16)                            color=font_color,
(16)                            family=font_family,
(16)                            weight=font_weight,
(16)                            alpha=alpha,
(16)                            horizontalalignment=horizontalalignment,
(16)                            verticalalignment=verticalalignment,
(16)                            rotation=0,
(16)                            transform=ax.transData,
(16)                            bbox=bbox,
(16)                            zorder=1,
(16)                            clip_on=clip_on,
(12)                        )
(8)                     else:
(12)                        text_items[edge] = CurvedArrowText(
(16)                            arrow,
(16)                            label,
(16)                            size=font_size,
(16)                            color=font_color,
(16)                            family=font_family,
(16)                            weight=font_weight,
(16)                            alpha=alpha,
(16)                            horizontalalignment=horizontalalignment,
(16)                            verticalalignment=verticalalignment,
(16)                            transform=ax.transData,
(16)                            bbox=bbox,
(16)                            zorder=1,
(16)                            clip_on=clip_on,
(16)                            label_pos=label_pos,
(16)                            labels_horizontal=not rotate,
(16)                            ax=ax,
(12)                        )
(4)                 if hide_ticks:
(8)                     ax.tick_params(
(12)                        axis="both",
(12)                        which="both",
(12)                        bottom=False,
(12)                        left=False,
(12)                        labelbottom=False,
(12)                        labelleft=False,
(8)                     )
(4)                 return text_items
```

```
(0)         def draw_circular(G, **kwargs):
(4)             """Draw the graph `G` with a circular layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.circular_layout(G), **kwargs)
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A networkx graph
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
(4)             Notes
(4)             -----
(4)             The layout is computed each time this function is called. For
(4)             repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.circular_layout` directly and reuse the result::
(8)                 >>> G = nx.complete_graph(5)
(8)                 >>> pos = nx.circular_layout(G)
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
(8)                 >>> # Draw a subgraph, reusing the same node positions
(8)                 >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> nx.draw_circular(G)
(4)             See Also
(4)             --------
(4)             :func:`~networkx.drawing.layout.circular_layout`
(4)             """
(4)             draw(G, circular_layout(G), **kwargs)
(0)         def draw_kamada_kawai(G, **kwargs):
(4)             """Draw the graph `G` with a Kamada-Kawai force-directed layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.kamada_kawai_layout(G), **kwargs)
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A networkx graph
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
(4)             Notes
(4)             -----
(4)             The layout is computed each time this function is called.
(4)             For repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.kamada_kawai_layout` directly and reuse the
(4)             result::
(8)                 >>> G = nx.complete_graph(5)
(8)                 >>> pos = nx.kamada_kawai_layout(G)
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
(8)                 >>> # Draw a subgraph, reusing the same node positions
(8)                 >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> nx.draw_kamada_kawai(G)
(4)             See Also
(4)             --------
(4)             :func:`~networkx.drawing.layout.kamada_kawai_layout`
(4)             """
(4)             draw(G, kamada_kawai_layout(G), **kwargs)
(0)         def draw_random(G, **kwargs):
(4)             """Draw the graph `G` with a random layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.random_layout(G), **kwargs)
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A networkx graph
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
```

```
(4)             Notes
(4)             -----
(4)             The layout is computed each time this function is called.
(4)             For repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.random_layout` directly and reuse the result::
(8)                 >>> G = nx.complete_graph(5)
(8)                 >>> pos = nx.random_layout(G)
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
(8)                 >>> # Draw a subgraph, reusing the same node positions
(8)                 >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
(4)             Examples
(4)             --------
(4)             >>> G = nx.lollipop_graph(4, 3)
(4)             >>> nx.draw_random(G)
(4)             See Also
(4)             --------
(4)             :func:`~networkx.drawing.layout.random_layout`
(4)             """
(4)             draw(G, random_layout(G), **kwargs)
(0)         def draw_spectral(G, **kwargs):
(4)             """Draw the graph `G` with a spectral 2D layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.spectral_layout(G), **kwargs)
(4)             For more information about how node positions are determined, see
(4)             `~networkx.drawing.layout.spectral_layout`.
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A networkx graph
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
(4)             Notes
(4)             -----
(4)             The layout is computed each time this function is called.
(4)             For repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.spectral_layout` directly and reuse the result::
(8)                 >>> G = nx.complete_graph(5)
(8)                 >>> pos = nx.spectral_layout(G)
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
(8)                 >>> # Draw a subgraph, reusing the same node positions
(8)                 >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(5)
(4)             >>> nx.draw_spectral(G)
(4)             See Also
(4)             --------
(4)             :func:`~networkx.drawing.layout.spectral_layout`
(4)             """
(4)             draw(G, spectral_layout(G), **kwargs)
(0)         def draw_spring(G, **kwargs):
(4)             """Draw the graph `G` with a spring layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.spring_layout(G), **kwargs)
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A networkx graph
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
(4)             Notes
(4)             -----
(4)             `~networkx.drawing.layout.spring_layout` is also the default layout for
(4)             `draw`, so this function is equivalent to `draw`.
(4)             The layout is computed each time this function is called.
(4)             For repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.spring_layout` directly and reuse the result::
(8)                 >>> G = nx.complete_graph(5)
(8)                 >>> pos = nx.spring_layout(G)
```

```
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
(8)                 >>> # Draw a subgraph, reusing the same node positions
(8)                 >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(20)
(4)             >>> nx.draw_spring(G)
(4)             See Also
(4)             --------
(4)             draw
(4)             :func:`~networkx.drawing.layout.spring_layout`
(4)             """
(4)             draw(G, spring_layout(G), **kwargs)
(0)         def draw_shell(G, nlist=None, **kwargs):
(4)             """Draw networkx graph `G` with shell layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.shell_layout(G, nlist=nlist), **kwargs)
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A networkx graph
(4)             nlist : list of list of nodes, optional
(8)                 A list containing lists of nodes representing the shells.
(8)                 Default is `None`, meaning all nodes are in a single shell.
(8)                 See `~networkx.drawing.layout.shell_layout` for details.
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
(4)             Notes
(4)             -----
(4)             The layout is computed each time this function is called.
(4)             For repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.shell_layout` directly and reuse the result::
(8)                 >>> G = nx.complete_graph(5)
(8)                 >>> pos = nx.shell_layout(G)
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
(8)                 >>> # Draw a subgraph, reusing the same node positions
(8)                 >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
(4)             Examples
(4)             --------
(4)             >>> G = nx.path_graph(4)
(4)             >>> shells = [[0], [1, 2, 3]]
(4)             >>> nx.draw_shell(G, nlist=shells)
(4)             See Also
(4)             --------
(4)             :func:`~networkx.drawing.layout.shell_layout`
(4)             """
(4)             draw(G, shell_layout(G, nlist=nlist), **kwargs)
(0)         def draw_planar(G, **kwargs):
(4)             """Draw a planar networkx graph `G` with planar layout.
(4)             This is a convenience function equivalent to::
(8)                 nx.draw(G, pos=nx.planar_layout(G), **kwargs)
(4)             Parameters
(4)             ----------
(4)             G : graph
(8)                 A planar networkx graph
(4)             kwargs : optional keywords
(8)                 See `draw_networkx` for a description of optional keywords.
(4)             Raises
(4)             ------
(4)             NetworkXException
(8)                 When `G` is not planar
(4)             Notes
(4)             -----
(4)             The layout is computed each time this function is called.
(4)             For repeated drawing it is much more efficient to call
(4)             `~networkx.drawing.layout.planar_layout` directly and reuse the result::
(8)                 >>> G = nx.path_graph(5)
(8)                 >>> pos = nx.planar_layout(G)
(8)                 >>> nx.draw(G, pos=pos)  # Draw the original graph
```

```
1418: (8)                   >>> # Draw a subgraph, reusing the same node positions
1419: (8)                   >>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
1420: (4)               Examples
1421: (4)               --------
1422: (4)               >>> G = nx.path_graph(4)
1423: (4)               >>> nx.draw_planar(G)
1424: (4)               See Also
1425: (4)               --------
1426: (4)               :func:`~networkx.drawing.layout.planar_layout`
1427: (4)               """
1428: (4)               draw(G, planar_layout(G), **kwargs)
1429: (0)           def apply_alpha(colors, alpha, elem_list, cmap=None, vmin=None, vmax=None):
1430: (4)               """Apply an alpha (or list of alphas) to the colors provided.
1431: (4)               Parameters
1432: (4)               ----------
1433: (4)               colors : color string or array of floats (default='r')
1434: (8)                   Color of element. Can be a single color format string,
1435: (8)                   or a sequence of colors with the same length as nodelist.
1436: (8)                   If numeric values are specified they will be mapped to
1437: (8)                   colors using the cmap and vmin,vmax parameters.  See
1438: (8)                   matplotlib.scatter for more details.
1439: (4)               alpha : float or array of floats
1440: (8)                   Alpha values for elements. This can be a single alpha value, in
1441: (8)                   which case it will be applied to all the elements of color. Otherwise,
1442: (8)                   if it is an array, the elements of alpha will be applied to the colors
1443: (8)                   in order (cycling through alpha multiple times if necessary).
1444: (4)               elem_list : array of networkx objects
1445: (8)                   The list of elements which are being colored. These could be nodes,
1446: (8)                   edges or labels.
1447: (4)               cmap : matplotlib colormap
1448: (8)                   Color map for use if colors is a list of floats corresponding to
points
1449: (8)                   on a color mapping.
1450: (4)               vmin, vmax : float
1451: (8)                   Minimum and maximum values for normalizing colors if a colormap is
used
1452: (4)               Returns
1453: (4)               -------
1454: (4)               rgba_colors : numpy ndarray
1455: (8)                   Array containing RGBA format values for each of the node colours.
1456: (4)               """
1457: (4)               from itertools import cycle, islice
1458: (4)               import matplotlib as mpl
1459: (4)               import matplotlib.cm  # call as mpl.cm
1460: (4)               import matplotlib.colors  # call as mpl.colors
1461: (4)               import numpy as np
1462: (4)               if len(colors) == len(elem_list) and isinstance(colors[0], Number):
1463: (8)                   mapper = mpl.cm.ScalarMappable(cmap=cmap)
1464: (8)                   mapper.set_clim(vmin, vmax)
1465: (8)                   rgba_colors = mapper.to_rgba(colors)
1466: (4)               else:
1467: (8)                   try:
1468: (12)                      rgba_colors =
np.array([mpl.colors.colorConverter.to_rgba(colors)])
1469: (8)                   except ValueError:
1470: (12)                      rgba_colors = np.array(
1471: (16)                          [mpl.colors.colorConverter.to_rgba(color) for color in colors]
1472: (12)                      )
1473: (4)               try:
1474: (8)                   if len(alpha) > len(rgba_colors) or rgba_colors.size ==
len(elem_list):
1475: (12)                      rgba_colors = np.resize(rgba_colors, (len(elem_list), 4))
1476: (12)                      rgba_colors[1:, 0] = rgba_colors[0, 0]
1477: (12)                      rgba_colors[1:, 1] = rgba_colors[0, 1]
1478: (12)                      rgba_colors[1:, 2] = rgba_colors[0, 2]
1479: (8)                   rgba_colors[:, 3] = list(islice(cycle(alpha), len(rgba_colors)))
1480: (4)               except TypeError:
1481: (8)                   rgba_colors[:, -1] = alpha
1482: (4)               return rgba_colors
```

```
----------------------------------------

File 420 - __init__.py:

1: (0)

----------------------------------------

File 421 - test_agraph.py:

1: (0)              """Unit tests for PyGraphviz interface."""
2: (0)              import warnings
3: (0)              import pytest
4: (0)              pygraphviz = pytest.importorskip("pygraphviz")
5: (0)              import networkx as nx
6: (0)              from networkx.utils import edges_equal, graphs_equal, nodes_equal
7: (0)              class TestAGraph:
8: (4)                  def build_graph(self, G):
9: (8)                      edges = [("A", "B"), ("A", "C"), ("A", "C"), ("B", "C"), ("A", "D")]
10: (8)                     G.add_edges_from(edges)
11: (8)                     G.add_node("E")
12: (8)                     G.graph["metal"] = "bronze"
13: (8)                     return G
14: (4)                 def assert_equal(self, G1, G2):
15: (8)                     assert nodes_equal(G1.nodes(), G2.nodes())
16: (8)                     assert edges_equal(G1.edges(), G2.edges())
17: (8)                     assert G1.graph["metal"] == G2.graph["metal"]
18: (4)                 @pytest.mark.parametrize(
19: (8)                     "G", (nx.Graph(), nx.DiGraph(), nx.MultiGraph(), nx.MultiDiGraph())
20: (4)                 )
21: (4)                 def test_agraph_roundtripping(self, G, tmp_path):
22: (8)                     G = self.build_graph(G)
23: (8)                     A = nx.nx_agraph.to_agraph(G)
24: (8)                     H = nx.nx_agraph.from_agraph(A)
25: (8)                     self.assert_equal(G, H)
26: (8)                     fname = tmp_path / "test.dot"
27: (8)                     nx.drawing.nx_agraph.write_dot(H, fname)
28: (8)                     Hin = nx.nx_agraph.read_dot(fname)
29: (8)                     self.assert_equal(H, Hin)
30: (8)                     fname = tmp_path / "fh_test.dot"
31: (8)                     with open(fname, "w") as fh:
32: (12)                        nx.drawing.nx_agraph.write_dot(H, fh)
33: (8)                     with open(fname) as fh:
34: (12)                        Hin = nx.nx_agraph.read_dot(fh)
35: (8)                     self.assert_equal(H, Hin)
36: (4)                 def test_from_agraph_name(self):
37: (8)                     G = nx.Graph(name="test")
38: (8)                     A = nx.nx_agraph.to_agraph(G)
39: (8)                     H = nx.nx_agraph.from_agraph(A)
40: (8)                     assert G.name == "test"
41: (4)                 @pytest.mark.parametrize(
42: (8)                     "graph_class", (nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph)
43: (4)                 )
44: (4)                 def test_from_agraph_create_using(self, graph_class):
45: (8)                     G = nx.path_graph(3)
46: (8)                     A = nx.nx_agraph.to_agraph(G)
47: (8)                     H = nx.nx_agraph.from_agraph(A, create_using=graph_class)
48: (8)                     assert isinstance(H, graph_class)
49: (4)                 def test_from_agraph_named_edges(self):
50: (8)                     G = nx.Graph()
51: (8)                     G.add_nodes_from([0, 1])
52: (8)                     A = nx.nx_agraph.to_agraph(G)
53: (8)                     A.add_edge(0, 1, key="foo")
54: (8)                     H = nx.nx_agraph.from_agraph(A)
55: (8)                     assert isinstance(H, nx.Graph)
56: (8)                     assert ("0", "1", {"key": "foo"}) in H.edges(data=True)
57: (4)                 def test_to_agraph_with_nodedata(self):
58: (8)                     G = nx.Graph()
```

```
59: (8)                     G.add_node(1, color="red")
60: (8)                     A = nx.nx_agraph.to_agraph(G)
61: (8)                     assert dict(A.nodes()[0].attr) == {"color": "red"}
62: (4)                 @pytest.mark.parametrize("graph_class", (nx.Graph, nx.MultiGraph))
63: (4)                 def test_to_agraph_with_edgedata(self, graph_class):
64: (8)                     G = graph_class()
65: (8)                     G.add_nodes_from([0, 1])
66: (8)                     G.add_edge(0, 1, color="yellow")
67: (8)                     A = nx.nx_agraph.to_agraph(G)
68: (8)                     assert dict(A.edges()[0].attr) == {"color": "yellow"}
69: (4)                 def test_view_pygraphviz_path(self, tmp_path):
70: (8)                     G = nx.complete_graph(3)
71: (8)                     input_path = str(tmp_path / "graph.png")
72: (8)                     out_path, A = nx.nx_agraph.view_pygraphviz(G, path=input_path,
show=False)
73: (8)                     assert out_path == input_path
74: (8)                     with open(input_path, "rb") as fh:
75: (12)                        data = fh.read()
76: (8)                     assert len(data) > 0
77: (4)                 def test_view_pygraphviz_file_suffix(self, tmp_path):
78: (8)                     G = nx.complete_graph(3)
79: (8)                     path, A = nx.nx_agraph.view_pygraphviz(G, suffix=1, show=False)
80: (8)                     assert path[-6:] == "_1.png"
81: (4)                 def test_view_pygraphviz(self):
82: (8)                     G = nx.Graph()  # "An empty graph cannot be drawn."
83: (8)                     pytest.raises(nx.NetworkXException, nx.nx_agraph.view_pygraphviz, G)
84: (8)                     G = nx.barbell_graph(4, 6)
85: (8)                     nx.nx_agraph.view_pygraphviz(G, show=False)
86: (4)                 def test_view_pygraphviz_edgelabel(self):
87: (8)                     G = nx.Graph()
88: (8)                     G.add_edge(1, 2, weight=7)
89: (8)                     G.add_edge(2, 3, weight=8)
90: (8)                     path, A = nx.nx_agraph.view_pygraphviz(G, edgelabel="weight",
show=False)
91: (8)                     for edge in A.edges():
92: (12)                        assert edge.attr["weight"] in ("7", "8")
93: (4)                 def test_view_pygraphviz_callable_edgelabel(self):
94: (8)                     G = nx.complete_graph(3)
95: (8)                     def foo_label(data):
96: (12)                        return "foo"
97: (8)                     path, A = nx.nx_agraph.view_pygraphviz(G, edgelabel=foo_label,
show=False)
98: (8)                     for edge in A.edges():
99: (12)                        assert edge.attr["label"] == "foo"
100: (4)                def test_view_pygraphviz_multigraph_edgelabels(self):
101: (8)                    G = nx.MultiGraph()
102: (8)                    G.add_edge(0, 1, key=0, name="left_fork")
103: (8)                    G.add_edge(0, 1, key=1, name="right_fork")
104: (8)                    path, A = nx.nx_agraph.view_pygraphviz(G, edgelabel="name",
show=False)
105: (8)                    edges = A.edges()
106: (8)                    assert len(edges) == 2
107: (8)                    for edge in edges:
108: (12)                       assert edge.attr["label"].strip() in ("left_fork", "right_fork")
109: (4)                def test_graph_with_reserved_keywords(self):
110: (8)                    G = nx.Graph()
111: (8)                    G = self.build_graph(G)
112: (8)                    G.nodes["E"]["n"] = "keyword"
113: (8)                    G.edges[("A", "B")]["u"] = "keyword"
114: (8)                    G.edges[("A", "B")]["v"] = "keyword"
115: (8)                    A = nx.nx_agraph.to_agraph(G)
116: (4)                def test_view_pygraphviz_no_added_attrs_to_input(self):
117: (8)                    G = nx.complete_graph(2)
118: (8)                    path, A = nx.nx_agraph.view_pygraphviz(G, show=False)
119: (8)                    assert G.graph == {}
120: (4)                @pytest.mark.xfail(reason="known bug in clean_attrs")
121: (4)                def test_view_pygraphviz_leaves_input_graph_unmodified(self):
122: (8)                    G = nx.complete_graph(2)
123: (8)                    G.graph["node"] = {"width": "0.80"}
```

```
124: (8)                    G.graph["edge"] = {"fontsize": "14"}
125: (8)                    path, A = nx.nx_agraph.view_pygraphviz(G, show=False)
126: (8)                    assert G.graph == {"node": {"width": "0.80"}, "edge": {"fontsize":
"14"}}
127: (4)                def test_graph_with_AGraph_attrs(self):
128: (8)                    G = nx.complete_graph(2)
129: (8)                    G.graph["node"] = {"width": "0.80"}
130: (8)                    G.graph["edge"] = {"fontsize": "14"}
131: (8)                    path, A = nx.nx_agraph.view_pygraphviz(G, show=False)
132: (8)                    assert dict(A.node_attr)["width"] == "0.80"
133: (8)                    assert dict(A.edge_attr)["fontsize"] == "14"
134: (4)                def test_round_trip_empty_graph(self):
135: (8)                    G = nx.Graph()
136: (8)                    A = nx.nx_agraph.to_agraph(G)
137: (8)                    H = nx.nx_agraph.from_agraph(A)
138: (8)                    AA = nx.nx_agraph.to_agraph(H)
139: (8)                    HH = nx.nx_agraph.from_agraph(AA)
140: (8)                    assert graphs_equal(H, HH)
141: (8)                    G.graph["graph"] = {}
142: (8)                    G.graph["node"] = {}
143: (8)                    G.graph["edge"] = {}
144: (8)                    assert graphs_equal(G, HH)
145: (4)                @pytest.mark.xfail(reason="integer->string node conversion in round trip")
146: (4)                def test_round_trip_integer_nodes(self):
147: (8)                    G = nx.complete_graph(3)
148: (8)                    A = nx.nx_agraph.to_agraph(G)
149: (8)                    H = nx.nx_agraph.from_agraph(A)
150: (8)                    assert graphs_equal(G, H)
151: (4)                def test_graphviz_alias(self):
152: (8)                    G = self.build_graph(nx.Graph())
153: (8)                    pos_graphviz = nx.nx_agraph.graphviz_layout(G)
154: (8)                    pos_pygraphviz = nx.nx_agraph.pygraphviz_layout(G)
155: (8)                    assert pos_graphviz == pos_pygraphviz
156: (4)                @pytest.mark.parametrize("root", range(5))
157: (4)                def test_pygraphviz_layout_root(self, root):
158: (8)                    G = nx.complete_graph(5)
159: (8)                    A = nx.nx_agraph.to_agraph(G)
160: (8)                    pygv_layout = nx.nx_agraph.pygraphviz_layout(G, prog="circo",
root=root)
161: (8)                    A.layout(args=f"-Groot={root}", prog="circo")
162: (8)                    a1_pos = tuple(float(v) for v in dict(A.get_node("1").attr)
["pos"].split(","))
163: (8)                    assert pygv_layout[1] == a1_pos
164: (4)                def test_2d_layout(self):
165: (8)                    G = nx.Graph()
166: (8)                    G = self.build_graph(G)
167: (8)                    G.graph["dimen"] = 2
168: (8)                    pos = nx.nx_agraph.pygraphviz_layout(G, prog="neato")
169: (8)                    pos = list(pos.values())
170: (8)                    assert len(pos) == 5
171: (8)                    assert len(pos[0]) == 2
172: (4)                def test_3d_layout(self):
173: (8)                    G = nx.Graph()
174: (8)                    G = self.build_graph(G)
175: (8)                    G.graph["dimen"] = 3
176: (8)                    pos = nx.nx_agraph.pygraphviz_layout(G, prog="neato")
177: (8)                    pos = list(pos.values())
178: (8)                    assert len(pos) == 5
179: (8)                    assert len(pos[0]) == 3
180: (4)                def test_no_warnings_raised(self):
181: (8)                    G = nx.Graph()
182: (8)                    G.add_node(0, pos=(0, 0))
183: (8)                    G.add_node(1, pos=(1, 1))
184: (8)                    A = nx.nx_agraph.to_agraph(G)
185: (8)                    with warnings.catch_warnings(record=True) as record:
186: (12)                       A.layout()
187: (8)                    assert len(record) == 0

----------------------------------------
```

File 422 - test_latex.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_tikz_attributes():
4: (4)                  G = nx.path_graph(4, create_using=nx.DiGraph)
5: (4)                  pos = {n: (n, n) for n in G}
6: (4)                  G.add_edge(0, 0)
7: (4)                  G.edges[(0, 0)]["label"] = "Loop"
8: (4)                  G.edges[(0, 0)]["label_options"] = "midway"
9: (4)                  G.nodes[0]["style"] = "blue"
10: (4)                 G.nodes[1]["style"] = "line width=3,draw"
11: (4)                 G.nodes[2]["style"] = "circle,draw,blue!50"
12: (4)                 G.nodes[3]["label"] = "Stop"
13: (4)                 G.edges[(0, 1)]["label"] = "1st Step"
14: (4)                 G.edges[(0, 1)]["label_options"] = "near end"
15: (4)                 G.edges[(2, 3)]["label"] = "3rd Step"
16: (4)                 G.edges[(2, 3)]["label_options"] = "near start"
17: (4)                 G.edges[(2, 3)]["style"] = "bend left,green"
18: (4)                 G.edges[(1, 2)]["label"] = "2nd"
19: (4)                 G.edges[(1, 2)]["label_options"] = "pos=0.5"
20: (4)                 G.edges[(1, 2)]["style"] = ">->,bend right,line width=3,green!90"
21: (4)                 output_tex = nx.to_latex(
22: (8)                     G,
23: (8)                     pos=pos,
24: (8)                     as_document=False,
25: (8)                     tikz_options="[scale=3]",
26: (8)                     node_options="style",
27: (8)                     edge_options="style",
28: (8)                     node_label="label",
29: (8)                     edge_label="label",
30: (8)                     edge_label_options="label_options",
31: (4)                 )
32: (4)                 expected_tex = r"""\begin{figure}
33: (2)               \begin{tikzpicture}[scale=3]
34: (6)                   \draw
35: (8)                     (0, 0) node[blue] (0){0}
36: (8)                     (1, 1) node[line width=3,draw] (1){1}
37: (8)                     (2, 2) node[circle,draw,blue!50] (2){2}
38: (8)                     (3, 3) node (3){Stop};
39: (6)                   \begin{scope}[->]
40: (8)                     \draw (0) to node[near end] {1st Step} (1);
41: (8)                     \draw[loop,] (0) to node[midway] {Loop} (0);
42: (8)                     \draw[>->,bend right,line width=3,green!90] (1) to node[pos=0.5] {2nd}
(2);
43: (8)                     \draw[bend left,green] (2) to node[near start] {3rd Step} (3);
44: (6)                   \end{scope}
45: (4)                 \end{tikzpicture}
46: (0)             \end{figure}"""
47: (4)                 assert output_tex == expected_tex
48: (0)             def test_basic_multiple_graphs():
49: (4)                 H1 = nx.path_graph(4)
50: (4)                 H2 = nx.complete_graph(4)
51: (4)                 H3 = nx.path_graph(8)
52: (4)                 H4 = nx.complete_graph(8)
53: (4)                 captions = [
54: (8)                     "Path on 4 nodes",
55: (8)                     "Complete graph on 4 nodes",
56: (8)                     "Path on 8 nodes",
57: (8)                     "Complete graph on 8 nodes",
58: (4)                 ]
59: (4)                 labels = ["fig2a", "fig2b", "fig2c", "fig2d"]
60: (4)                 latex_code = nx.to_latex(
61: (8)                     [H1, H2, H3, H4],
62: (8)                     n_rows=2,
63: (8)                     sub_captions=captions,
64: (8)                     sub_labels=labels,
65: (4)                 )
```

```
(4)                 assert "begin{document}" in latex_code
(4)                 assert "begin{figure}" in latex_code
(4)                 assert latex_code.count("begin{subfigure}") == 4
(4)                 assert latex_code.count("tikzpicture") == 8
(4)                 assert latex_code.count("[-]") == 4
(0)             def test_basic_tikz():
(4)                 expected_tex = r"""\documentclass{report}
(0)             \usepackage{tikz}
(0)             \usepackage{subcaption}
(0)             \begin{document}
(0)             \begin{figure}
(2)               \begin{subfigure}{0.5\textwidth}
(2)               \begin{tikzpicture}[scale=2]
(6)                   \draw[gray!90]
(8)                     (0.749, 0.702) node[red!90] (0){0}
(8)                     (1.0, -0.014) node[red!90] (1){1}
(8)                     (-0.777, -0.705) node (2){2}
(8)                     (-0.984, 0.042) node (3){3}
(8)                     (-0.028, 0.375) node[cyan!90] (4){4}
(8)                     (-0.412, 0.888) node (5){5}
(8)                     (0.448, -0.856) node (6){6}
(8)                     (0.003, -0.431) node[cyan!90] (7){7};
(6)                   \begin{scope}[->,gray!90]
(8)                     \draw (0) to (4);
(8)                     \draw (0) to (5);
(8)                     \draw (0) to (6);
(8)                     \draw (0) to (7);
(8)                     \draw (1) to (4);
(8)                     \draw (1) to (5);
(8)                     \draw (1) to (6);
(8)                     \draw (1) to (7);
(8)                     \draw (2) to (4);
(8)                     \draw (2) to (5);
(8)                     \draw (2) to (6);
(8)                     \draw (2) to (7);
(8)                     \draw (3) to (4);
(8)                     \draw (3) to (5);
(8)                     \draw (3) to (6);
(8)                     \draw (3) to (7);
(6)                   \end{scope}
(4)                 \end{tikzpicture}
(4)                 \caption{My tikz number 1 of 2}\label{tikz_1_2}
(2)               \end{subfigure}
(2)               \begin{subfigure}{0.5\textwidth}
(2)               \begin{tikzpicture}[scale=2]
(6)                   \draw[gray!90]
(8)                     (0.749, 0.702) node[green!90] (0){0}
(8)                     (1.0, -0.014) node[green!90] (1){1}
(8)                     (-0.777, -0.705) node (2){2}
(8)                     (-0.984, 0.042) node (3){3}
(8)                     (-0.028, 0.375) node[purple!90] (4){4}
(8)                     (-0.412, 0.888) node (5){5}
(8)                     (0.448, -0.856) node (6){6}
(8)                     (0.003, -0.431) node[purple!90] (7){7};
(6)                   \begin{scope}[->,gray!90]
(8)                     \draw (0) to (4);
(8)                     \draw (0) to (5);
(8)                     \draw (0) to (6);
(8)                     \draw (0) to (7);
(8)                     \draw (1) to (4);
(8)                     \draw (1) to (5);
(8)                     \draw (1) to (6);
(8)                     \draw (1) to (7);
(8)                     \draw (2) to (4);
(8)                     \draw (2) to (5);
(8)                     \draw (2) to (6);
(8)                     \draw (2) to (7);
(8)                     \draw (3) to (4);
(8)                     \draw (3) to (5);
```

```
(8)                 \draw (3) to (6);
(8)                 \draw (3) to (7);
(6)               \end{scope}
(4)             \end{tikzpicture}
(4)             \caption{My tikz number 2 of 2}\label{tikz_2_2}
(2)           \end{subfigure}
(2)           \caption{A graph generated with python and latex.}
(0)         \end{figure}
(0)         \end{document}"""
(4)             edges = [
(8)                 (0, 4),
(8)                 (0, 5),
(8)                 (0, 6),
(8)                 (0, 7),
(8)                 (1, 4),
(8)                 (1, 5),
(8)                 (1, 6),
(8)                 (1, 7),
(8)                 (2, 4),
(8)                 (2, 5),
(8)                 (2, 6),
(8)                 (2, 7),
(8)                 (3, 4),
(8)                 (3, 5),
(8)                 (3, 6),
(8)                 (3, 7),
(4)             ]
(4)             G = nx.DiGraph()
(4)             G.add_nodes_from(range(8))
(4)             G.add_edges_from(edges)
(4)             pos = {
(8)                 0: (0.7490296171687696, 0.702353520257394),
(8)                 1: (1.0, -0.014221357723796535),
(8)                 2: (-0.7765783344161441, -0.7054170966808919),
(8)                 3: (-0.9842690223417624, 0.04177547602465483),
(8)                 4: (-0.02768523817180917, 0.3745724439551441),
(8)                 5: (-0.41154855146767433, 0.8880106515525136),
(8)                 6: (0.44780153389148264, -0.8561492709269164),
(8)                 7: (0.0032499953371383505, -0.43092436645809945),
(4)             }
(4)             rc_node_color = {0: "red!90", 1: "red!90", 4: "cyan!90", 7: "cyan!90"}
(4)             gp_node_color = {0: "green!90", 1: "green!90", 4: "purple!90", 7:
"purple!90"}
(4)             H = G.copy()
(4)             nx.set_node_attributes(G, rc_node_color, "color")
(4)             nx.set_node_attributes(H, gp_node_color, "color")
(4)             sub_captions = ["My tikz number 1 of 2", "My tikz number 2 of 2"]
(4)             sub_labels = ["tikz_1_2", "tikz_2_2"]
(4)             output_tex = nx.to_latex(
(8)                 [G, H],
(8)                 [pos, pos],
(8)                 tikz_options="[scale=2]",
(8)                 default_node_options="gray!90",
(8)                 default_edge_options="gray!90",
(8)                 node_options="color",
(8)                 sub_captions=sub_captions,
(8)                 sub_labels=sub_labels,
(8)                 caption="A graph generated with python and latex.",
(8)                 n_rows=2,
(8)                 as_document=True,
(4)             )
(4)             assert output_tex == expected_tex
(0)         def test_exception_pos_single_graph(to_latex=nx.to_latex):
(4)             G = nx.path_graph(4)
(4)             to_latex(G, pos="pos")
(4)             pos = {0: (1, 2), 1: (0, 1), 2: (2, 1)}
(4)             with pytest.raises(nx.NetworkXError):
(8)                 to_latex(G, pos)
(4)             pos[3] = (1, 2, 3)
```

```
203: (4)                with pytest.raises(nx.NetworkXError):
204: (8)                    to_latex(G, pos)
205: (4)                pos[3] = 2
206: (4)                with pytest.raises(nx.NetworkXError):
207: (8)                    to_latex(G, pos)
208: (4)                pos[3] = (3, 2)
209: (4)                to_latex(G, pos)
210: (0)            def test_exception_multiple_graphs(to_latex=nx.to_latex):
211: (4)                G = nx.path_graph(3)
212: (4)                pos_bad = {0: (1, 2), 1: (0, 1)}
213: (4)                pos_OK = {0: (1, 2), 1: (0, 1), 2: (2, 1)}
214: (4)                fourG = [G, G, G, G]
215: (4)                fourpos = [pos_OK, pos_OK, pos_OK, pos_OK]
216: (4)                to_latex(fourG, pos_OK)
217: (4)                with pytest.raises(nx.NetworkXError):
218: (8)                    to_latex(fourG, pos_bad)
219: (4)                to_latex(fourG, fourpos)
220: (4)                with pytest.raises(nx.NetworkXError):
221: (8)                    to_latex(fourG, [pos_bad, pos_bad, pos_bad, pos_bad])
222: (4)                with pytest.raises(nx.NetworkXError):
223: (8)                    to_latex(fourG, [pos_OK, pos_OK, pos_bad, pos_OK])
224: (4)                with pytest.raises(nx.NetworkXError):
225: (8)                    to_latex(fourG, fourpos, sub_captions=["hi", "hi"])
226: (4)                with pytest.raises(nx.NetworkXError):
227: (8)                    to_latex(fourG, fourpos, sub_labels=["hi", "hi"])
228: (4)                to_latex(fourG, fourpos, sub_captions=["hi"] * 4, sub_labels=["lbl"] * 4)
229: (0)            def test_exception_multigraph():
230: (4)                G = nx.path_graph(4, create_using=nx.MultiGraph)
231: (4)                G.add_edge(1, 2)
232: (4)                with pytest.raises(nx.NetworkXNotImplemented):
233: (8)                    nx.to_latex(G)
```

File 423 - test_layout.py:

```
1: (0)              """Unit tests for layout functions."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              np = pytest.importorskip("numpy")
5: (0)              pytest.importorskip("scipy")
6: (0)              class TestLayout:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      cls.Gi = nx.grid_2d_graph(5, 5)
10: (8)                     cls.Gs = nx.Graph()
11: (8)                     nx.add_path(cls.Gs, "abcdef")
12: (8)                     cls.bigG = nx.grid_2d_graph(25, 25)  # > 500 nodes for sparse
13: (4)                 def test_spring_fixed_without_pos(self):
14: (8)                     G = nx.path_graph(4)
15: (8)                     pytest.raises(ValueError, nx.spring_layout, G, fixed=[0])
16: (8)                     pos = {0: (1, 1), 2: (0, 0)}
17: (8)                     pytest.raises(ValueError, nx.spring_layout, G, fixed=[0, 1], pos=pos)
18: (8)                     nx.spring_layout(G, fixed=[0, 2], pos=pos)  # No ValueError
19: (4)                 def test_spring_init_pos(self):
20: (8)                     import math
21: (8)                     G = nx.Graph()
22: (8)                     G.add_edges_from([(0, 1), (1, 2), (2, 0), (2, 3)])
23: (8)                     init_pos = {0: (0.0, 0.0)}
24: (8)                     fixed_pos = [0]
25: (8)                     pos = nx.fruchterman_reingold_layout(G, pos=init_pos, fixed=fixed_pos)
26: (8)                     has_nan = any(math.isnan(c) for coords in pos.values() for c in
coords)
27: (8)                     assert not has_nan, "values should not be nan"
28: (4)                 def test_smoke_empty_graph(self):
29: (8)                     G = []
30: (8)                     nx.random_layout(G)
31: (8)                     nx.circular_layout(G)
32: (8)                     nx.planar_layout(G)
```

```
33: (8)                     nx.spring_layout(G)
34: (8)                     nx.fruchterman_reingold_layout(G)
35: (8)                     nx.spectral_layout(G)
36: (8)                     nx.shell_layout(G)
37: (8)                     nx.bipartite_layout(G, G)
38: (8)                     nx.spiral_layout(G)
39: (8)                     nx.multipartite_layout(G)
40: (8)                     nx.kamada_kawai_layout(G)
41: (4)                 def test_smoke_int(self):
42: (8)                     G = self.Gi
43: (8)                     nx.random_layout(G)
44: (8)                     nx.circular_layout(G)
45: (8)                     nx.planar_layout(G)
46: (8)                     nx.spring_layout(G)
47: (8)                     nx.fruchterman_reingold_layout(G)
48: (8)                     nx.fruchterman_reingold_layout(self.bigG)
49: (8)                     nx.spectral_layout(G)
50: (8)                     nx.spectral_layout(G.to_directed())
51: (8)                     nx.spectral_layout(self.bigG)
52: (8)                     nx.spectral_layout(self.bigG.to_directed())
53: (8)                     nx.shell_layout(G)
54: (8)                     nx.spiral_layout(G)
55: (8)                     nx.kamada_kawai_layout(G)
56: (8)                     nx.kamada_kawai_layout(G, dim=1)
57: (8)                     nx.kamada_kawai_layout(G, dim=3)
58: (8)                     nx.arf_layout(G)
59: (4)                 def test_smoke_string(self):
60: (8)                     G = self.Gs
61: (8)                     nx.random_layout(G)
62: (8)                     nx.circular_layout(G)
63: (8)                     nx.planar_layout(G)
64: (8)                     nx.spring_layout(G)
65: (8)                     nx.fruchterman_reingold_layout(G)
66: (8)                     nx.spectral_layout(G)
67: (8)                     nx.shell_layout(G)
68: (8)                     nx.spiral_layout(G)
69: (8)                     nx.kamada_kawai_layout(G)
70: (8)                     nx.kamada_kawai_layout(G, dim=1)
71: (8)                     nx.kamada_kawai_layout(G, dim=3)
72: (8)                     nx.arf_layout(G)
73: (4)                 def check_scale_and_center(self, pos, scale, center):
74: (8)                     center = np.array(center)
75: (8)                     low = center - scale
76: (8)                     hi = center + scale
77: (8)                     vpos = np.array(list(pos.values()))
78: (8)                     length = vpos.max(0) - vpos.min(0)
79: (8)                     assert (length <= 2 * scale).all()
80: (8)                     assert (vpos >= low).all()
81: (8)                     assert (vpos <= hi).all()
82: (4)                 def test_scale_and_center_arg(self):
83: (8)                     sc = self.check_scale_and_center
84: (8)                     c = (4, 5)
85: (8)                     G = nx.complete_graph(9)
86: (8)                     G.add_node(9)
87: (8)                     sc(nx.random_layout(G, center=c), scale=0.5, center=(4.5, 5.5))
88: (8)                     sc(nx.spring_layout(G, scale=2, center=c), scale=2, center=c)
89: (8)                     sc(nx.spectral_layout(G, scale=2, center=c), scale=2, center=c)
90: (8)                     sc(nx.circular_layout(G, scale=2, center=c), scale=2, center=c)
91: (8)                     sc(nx.shell_layout(G, scale=2, center=c), scale=2, center=c)
92: (8)                     sc(nx.spiral_layout(G, scale=2, center=c), scale=2, center=c)
93: (8)                     sc(nx.kamada_kawai_layout(G, scale=2, center=c), scale=2, center=c)
94: (8)                     c = (2, 3, 5)
95: (8)                     sc(nx.kamada_kawai_layout(G, dim=3, scale=2, center=c), scale=2,
center=c)
96: (4)                 def test_planar_layout_non_planar_input(self):
97: (8)                     G = nx.complete_graph(9)
98: (8)                     pytest.raises(nx.NetworkXException, nx.planar_layout, G)
99: (4)                 def test_smoke_planar_layout_embedding_input(self):
100: (8)                    embedding = nx.PlanarEmbedding()
```

```
101: (8)                    embedding.set_data({0: [1, 2], 1: [0, 2], 2: [0, 1]})
102: (8)                    nx.planar_layout(embedding)
103: (4)                def test_default_scale_and_center(self):
104: (8)                    sc = self.check_scale_and_center
105: (8)                    c = (0, 0)
106: (8)                    G = nx.complete_graph(9)
107: (8)                    G.add_node(9)
108: (8)                    sc(nx.random_layout(G), scale=0.5, center=(0.5, 0.5))
109: (8)                    sc(nx.spring_layout(G), scale=1, center=c)
110: (8)                    sc(nx.spectral_layout(G), scale=1, center=c)
111: (8)                    sc(nx.circular_layout(G), scale=1, center=c)
112: (8)                    sc(nx.shell_layout(G), scale=1, center=c)
113: (8)                    sc(nx.spiral_layout(G), scale=1, center=c)
114: (8)                    sc(nx.kamada_kawai_layout(G), scale=1, center=c)
115: (8)                    c = (0, 0, 0)
116: (8)                    sc(nx.kamada_kawai_layout(G, dim=3), scale=1, center=c)
117: (4)                def test_circular_planar_and_shell_dim_error(self):
118: (8)                    G = nx.path_graph(4)
119: (8)                    pytest.raises(ValueError, nx.circular_layout, G, dim=1)
120: (8)                    pytest.raises(ValueError, nx.shell_layout, G, dim=1)
121: (8)                    pytest.raises(ValueError, nx.shell_layout, G, dim=3)
122: (8)                    pytest.raises(ValueError, nx.planar_layout, G, dim=1)
123: (8)                    pytest.raises(ValueError, nx.planar_layout, G, dim=3)
124: (4)                def test_adjacency_interface_numpy(self):
125: (8)                    A = nx.to_numpy_array(self.Gs)
126: (8)                    pos = nx.drawing.layout._fruchterman_reingold(A)
127: (8)                    assert pos.shape == (6, 2)
128: (8)                    pos = nx.drawing.layout._fruchterman_reingold(A, dim=3)
129: (8)                    assert pos.shape == (6, 3)
130: (8)                    pos = nx.drawing.layout._sparse_fruchterman_reingold(A)
131: (8)                    assert pos.shape == (6, 2)
132: (4)                def test_adjacency_interface_scipy(self):
133: (8)                    A = nx.to_scipy_sparse_array(self.Gs, dtype="d")
134: (8)                    pos = nx.drawing.layout._sparse_fruchterman_reingold(A)
135: (8)                    assert pos.shape == (6, 2)
136: (8)                    pos = nx.drawing.layout._sparse_spectral(A)
137: (8)                    assert pos.shape == (6, 2)
138: (8)                    pos = nx.drawing.layout._sparse_fruchterman_reingold(A, dim=3)
139: (8)                    assert pos.shape == (6, 3)
140: (4)                def test_single_nodes(self):
141: (8)                    G = nx.path_graph(1)
142: (8)                    vpos = nx.shell_layout(G)
143: (8)                    assert not vpos[0].any()
144: (8)                    G = nx.path_graph(4)
145: (8)                    vpos = nx.shell_layout(G, [[0], [1, 2], [3]])
146: (8)                    assert not vpos[0].any()
147: (8)                    assert vpos[3].any()  # ensure node 3 not at origin (#3188)
148: (8)                    assert np.linalg.norm(vpos[3]) <= 1  # ensure node 3 fits (#3753)
149: (8)                    vpos = nx.shell_layout(G, [[0], [1, 2], [3]], rotate=0)
150: (8)                    assert np.linalg.norm(vpos[3]) <= 1  # ensure node 3 fits (#3753)
151: (4)                def test_smoke_initial_pos_fruchterman_reingold(self):
152: (8)                    pos = nx.circular_layout(self.Gi)
153: (8)                    npos = nx.fruchterman_reingold_layout(self.Gi, pos=pos)
154: (4)                def test_smoke_initial_pos_arf(self):
155: (8)                    pos = nx.circular_layout(self.Gi)
156: (8)                    npos = nx.arf_layout(self.Gi, pos=pos)
157: (4)                def test_fixed_node_fruchterman_reingold(self):
158: (8)                    pos = nx.circular_layout(self.Gi)
159: (8)                    npos = nx.spring_layout(self.Gi, pos=pos, fixed=[(0, 0)])
160: (8)                    assert tuple(pos[(0, 0)]) == tuple(npos[(0, 0)])
161: (8)                    pos = nx.circular_layout(self.bigG)
162: (8)                    npos = nx.spring_layout(self.bigG, pos=pos, fixed=[(0, 0)])
163: (8)                    for axis in range(2):
164: (12)                       assert pos[(0, 0)][axis] == pytest.approx(npos[(0, 0)][axis],
abs=1e-7)
165: (4)                def test_center_parameter(self):
166: (8)                    G = nx.path_graph(1)
167: (8)                    nx.random_layout(G, center=(1, 1))
168: (8)                    vpos = nx.circular_layout(G, center=(1, 1))
```

```
(8)                 assert tuple(vpos[0]) == (1, 1)
(8)                 vpos = nx.planar_layout(G, center=(1, 1))
(8)                 assert tuple(vpos[0]) == (1, 1)
(8)                 vpos = nx.spring_layout(G, center=(1, 1))
(8)                 assert tuple(vpos[0]) == (1, 1)
(8)                 vpos = nx.fruchterman_reingold_layout(G, center=(1, 1))
(8)                 assert tuple(vpos[0]) == (1, 1)
(8)                 vpos = nx.spectral_layout(G, center=(1, 1))
(8)                 assert tuple(vpos[0]) == (1, 1)
(8)                 vpos = nx.shell_layout(G, center=(1, 1))
(8)                 assert tuple(vpos[0]) == (1, 1)
(8)                 vpos = nx.spiral_layout(G, center=(1, 1))
(8)                 assert tuple(vpos[0]) == (1, 1)
(4)             def test_center_wrong_dimensions(self):
(8)                 G = nx.path_graph(1)
(8)                 assert id(nx.spring_layout) == id(nx.fruchterman_reingold_layout)
(8)                 pytest.raises(ValueError, nx.random_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.circular_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.planar_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.spring_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.spring_layout, G, dim=3, center=(1, 1))
(8)                 pytest.raises(ValueError, nx.spectral_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.spectral_layout, G, dim=3, center=(1, 1))
(8)                 pytest.raises(ValueError, nx.shell_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.spiral_layout, G, center=(1, 1, 1))
(8)                 pytest.raises(ValueError, nx.kamada_kawai_layout, G, center=(1, 1, 1))
(4)             def test_empty_graph(self):
(8)                 G = nx.empty_graph()
(8)                 vpos = nx.random_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.circular_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.planar_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.bipartite_layout(G, G)
(8)                 assert vpos == {}
(8)                 vpos = nx.spring_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.fruchterman_reingold_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.spectral_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.shell_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.spiral_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.multipartite_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.kamada_kawai_layout(G, center=(1, 1))
(8)                 assert vpos == {}
(8)                 vpos = nx.arf_layout(G)
(8)                 assert vpos == {}
(4)             def test_bipartite_layout(self):
(8)                 G = nx.complete_bipartite_graph(3, 5)
(8)                 top, bottom = nx.bipartite.sets(G)
(8)                 vpos = nx.bipartite_layout(G, top)
(8)                 assert len(vpos) == len(G)
(8)                 top_x = vpos[list(top)[0]][0]
(8)                 bottom_x = vpos[list(bottom)[0]][0]
(8)                 for node in top:
(12)                    assert vpos[node][0] == top_x
(8)                 for node in bottom:
(12)                    assert vpos[node][0] == bottom_x
(8)                 vpos = nx.bipartite_layout(
(12)                    G, top, align="horizontal", center=(2, 2), scale=2, aspect_ratio=1
(8)                 )
(8)                 assert len(vpos) == len(G)
(8)                 top_y = vpos[list(top)[0]][1]
(8)                 bottom_y = vpos[list(bottom)[0]][1]
```

```
238: (8)                    for node in top:
239: (12)                       assert vpos[node][1] == top_y
240: (8)                    for node in bottom:
241: (12)                       assert vpos[node][1] == bottom_y
242: (8)                    pytest.raises(ValueError, nx.bipartite_layout, G, top, align="foo")
243: (4)                def test_multipartite_layout(self):
244: (8)                    sizes = (0, 5, 7, 2, 8)
245: (8)                    G = nx.complete_multipartite_graph(*sizes)
246: (8)                    vpos = nx.multipartite_layout(G)
247: (8)                    assert len(vpos) == len(G)
248: (8)                    start = 0
249: (8)                    for n in sizes:
250: (12)                       end = start + n
251: (12)                       assert all(vpos[start][0] == vpos[i][0] for i in range(start + 1,
end))
252: (12)                       start += n
253: (8)                    vpos = nx.multipartite_layout(G, align="horizontal", scale=2, center=
(2, 2))
254: (8)                    assert len(vpos) == len(G)
255: (8)                    start = 0
256: (8)                    for n in sizes:
257: (12)                       end = start + n
258: (12)                       assert all(vpos[start][1] == vpos[i][1] for i in range(start + 1,
end))
259: (12)                       start += n
260: (8)                    pytest.raises(ValueError, nx.multipartite_layout, G, align="foo")
261: (4)                def test_kamada_kawai_costfn_1d(self):
262: (8)                    costfn = nx.drawing.layout._kamada_kawai_costfn
263: (8)                    pos = np.array([4.0, 7.0])
264: (8)                    invdist = 1 / np.array([[0.1, 2.0], [2.0, 0.3]])
265: (8)                    cost, grad = costfn(pos, np, invdist, meanweight=0, dim=1)
266: (8)                    assert cost == pytest.approx(((3 / 2.0 - 1) ** 2), abs=1e-7)
267: (8)                    assert grad[0] == pytest.approx((-0.5), abs=1e-7)
268: (8)                    assert grad[1] == pytest.approx(0.5, abs=1e-7)
269: (4)                def check_kamada_kawai_costfn(self, pos, invdist, meanwt, dim):
270: (8)                    costfn = nx.drawing.layout._kamada_kawai_costfn
271: (8)                    cost, grad = costfn(pos.ravel(), np, invdist, meanweight=meanwt,
dim=dim)
272: (8)                    expected_cost = 0.5 * meanwt * np.sum(np.sum(pos, axis=0) ** 2)
273: (8)                    for i in range(pos.shape[0]):
274: (12)                       for j in range(i + 1, pos.shape[0]):
275: (16)                           diff = np.linalg.norm(pos[i] - pos[j])
276: (16)                           expected_cost += (diff * invdist[i][j] - 1.0) ** 2
277: (8)                    assert cost == pytest.approx(expected_cost, abs=1e-7)
278: (8)                    dx = 1e-4
279: (8)                    for nd in range(pos.shape[0]):
280: (12)                       for dm in range(pos.shape[1]):
281: (16)                           idx = nd * pos.shape[1] + dm
282: (16)                           ps = pos.flatten()
283: (16)                           ps[idx] += dx
284: (16)                           cplus = costfn(ps, np, invdist, meanweight=meanwt,
dim=pos.shape[1])[0]
285: (16)                           ps[idx] -= 2 * dx
286: (16)                           cminus = costfn(ps, np, invdist, meanweight=meanwt,
dim=pos.shape[1])[0]
287: (16)                           assert grad[idx] == pytest.approx((cplus - cminus) / (2 * dx),
abs=1e-5)
288: (4)                def test_kamada_kawai_costfn(self):
289: (8)                    invdist = 1 / np.array([[0.1, 2.1, 1.7], [2.1, 0.2, 0.6], [1.7, 0.6,
0.3]])
290: (8)                    meanwt = 0.3
291: (8)                    pos = np.array([[1.3, -3.2], [2.7, -0.3], [5.1, 2.5]])
292: (8)                    self.check_kamada_kawai_costfn(pos, invdist, meanwt, 2)
293: (8)                    pos = np.array([[0.9, 8.6, -8.7], [-10, -0.5, -7.1], [9.1, -8.1,
1.6]])
294: (8)                    self.check_kamada_kawai_costfn(pos, invdist, meanwt, 3)
295: (4)                def test_spiral_layout(self):
296: (8)                    G = self.Gs
297: (8)                    pos_standard = np.array(list(nx.spiral_layout(G,
```

```
resolution=0.35).values()))
298: (8)                    pos_tighter = np.array(list(nx.spiral_layout(G,
resolution=0.34).values()))
299: (8)                    distances = np.linalg.norm(pos_standard[:-1] - pos_standard[1:],
axis=1)
300: (8)                    distances_tighter = np.linalg.norm(pos_tighter[:-1] - pos_tighter[1:],
axis=1)
301: (8)                    assert sum(distances) > sum(distances_tighter)
302: (8)                    pos_equidistant = np.array(list(nx.spiral_layout(G,
equidistant=True).values()))
303: (8)                    distances_equidistant = np.linalg.norm(
304: (12)                       pos_equidistant[:-1] - pos_equidistant[1:], axis=1
305: (8)                    )
306: (8)                    assert np.allclose(
307: (12)                       distances_equidistant[1:], distances_equidistant[-1], atol=0.01
308: (8)                    )
309: (4)                def test_spiral_layout_equidistant(self):
310: (8)                    G = nx.path_graph(10)
311: (8)                    pos = nx.spiral_layout(G, equidistant=True)
312: (8)                    p = np.array(list(pos.values()))
313: (8)                    dist = np.linalg.norm(p[1:] - p[:-1], axis=1)
314: (8)                    assert np.allclose(np.diff(dist), 0, atol=1e-3)
315: (4)                def test_rescale_layout_dict(self):
316: (8)                    G = nx.empty_graph()
317: (8)                    vpos = nx.random_layout(G, center=(1, 1))
318: (8)                    assert nx.rescale_layout_dict(vpos) == {}
319: (8)                    G = nx.empty_graph(2)
320: (8)                    vpos = {0: (0.0, 0.0), 1: (1.0, 1.0)}
321: (8)                    s_vpos = nx.rescale_layout_dict(vpos)
322: (8)                    assert np.linalg.norm([sum(x) for x in zip(*s_vpos.values())]) < 1e-6
323: (8)                    G = nx.empty_graph(3)
324: (8)                    vpos = {0: (0, 0), 1: (1, 1), 2: (0.5, 0.5)}
325: (8)                    s_vpos = nx.rescale_layout_dict(vpos)
326: (8)                    expectation = {
327: (12)                       0: np.array((-1, -1)),
328: (12)                       1: np.array((1, 1)),
329: (12)                       2: np.array((0, 0)),
330: (8)                    }
331: (8)                    for k, v in expectation.items():
332: (12)                       assert (s_vpos[k] == v).all()
333: (8)                    s_vpos = nx.rescale_layout_dict(vpos, scale=2)
334: (8)                    expectation = {
335: (12)                       0: np.array((-2, -2)),
336: (12)                       1: np.array((2, 2)),
337: (12)                       2: np.array((0, 0)),
338: (8)                    }
339: (8)                    for k, v in expectation.items():
340: (12)                       assert (s_vpos[k] == v).all()
341: (4)                def test_arf_layout_partial_input_test(self):
342: (8)                    """
343: (8)                    Checks whether partial pos input still returns a proper position.
344: (8)                    """
345: (8)                    G = self.Gs
346: (8)                    node = nx.utils.arbitrary_element(G)
347: (8)                    pos = nx.circular_layout(G)
348: (8)                    del pos[node]
349: (8)                    pos = nx.arf_layout(G, pos=pos)
350: (8)                    assert len(pos) == len(G)
351: (4)                def test_arf_layout_negative_a_check(self):
352: (8)                    """
353: (8)                    Checks input parameters correctly raises errors. For example,  `a`
should be larger than 1
354: (8)                    """
355: (8)                    G = self.Gs
356: (8)                    pytest.raises(ValueError, nx.arf_layout, G=G, a=-1)
357: (0)            def test_multipartite_layout_nonnumeric_partition_labels():
358: (4)                """See gh-5123."""
359: (4)                G = nx.Graph()
360: (4)                G.add_node(0, subset="s0")
```

```
361: (4)                G.add_node(1, subset="s0")
362: (4)                G.add_node(2, subset="s1")
363: (4)                G.add_node(3, subset="s1")
364: (4)                G.add_edges_from([(0, 2), (0, 3), (1, 2)])
365: (4)                pos = nx.multipartite_layout(G)
366: (4)                assert len(pos) == len(G)
367: (0)            def test_multipartite_layout_layer_order():
368: (4)                """Return the layers in sorted order if the layers of the multipartite
369: (4)                graph are sortable. See gh-5691"""
370: (4)                G = nx.Graph()
371: (4)                node_group = dict(zip(("a", "b", "c", "d", "e"), (2, 3, 1, 2, 4)))
372: (4)                for node, layer in node_group.items():
373: (8)                    G.add_node(node, subset=layer)
374: (4)                pos = nx.multipartite_layout(G, align="horizontal")
375: (4)                layers = nx.utils.groups(node_group)
376: (4)                pos_from_layers = nx.multipartite_layout(G, align="horizontal",
subset_key=layers)
377: (4)                for (n1, p1), (n2, p2) in zip(pos.items(), pos_from_layers.items()):
378: (8)                    assert n1 == n2 and (p1 == p2).all()
379: (4)                assert pos["a"][-1] == pos["d"][-1]
380: (4)                assert pos["c"][-1] < pos["a"][-1] < pos["b"][-1] < pos["e"][-1]
381: (4)                G.nodes["a"]["subset"] = "layer_0"  # Can't sort mixed strs/ints
382: (4)                pos_nosort = nx.multipartite_layout(G)  # smoke test: this should not
raise
383: (4)                assert pos_nosort.keys() == pos.keys()
384: (0)            def _num_nodes_per_bfs_layer(pos):
385: (4)                """Helper function to extract the number of nodes in each layer of
bfs_layout"""
386: (4)                x = np.array(list(pos.values()))[:, 0]  # node positions in layered
dimension
387: (4)                _, layer_count = np.unique(x, return_counts=True)
388: (4)                return layer_count
389: (0)            @pytest.mark.parametrize("n", range(2, 7))
390: (0)            def test_bfs_layout_complete_graph(n):
391: (4)                """The complete graph should result in two layers: the starting node and
392: (4)                a second layer containing all neighbors."""
393: (4)                G = nx.complete_graph(n)
394: (4)                pos = nx.bfs_layout(G, start=0)
395: (4)                assert np.array_equal(_num_nodes_per_bfs_layer(pos), [1, n - 1])
396: (0)            def test_bfs_layout_barbell():
397: (4)                G = nx.barbell_graph(5, 3)
398: (4)                pos = nx.bfs_layout(G, start=0)
399: (4)                expected_nodes_per_layer = [1, 4, 1, 1, 1, 1, 4]
400: (4)                assert np.array_equal(_num_nodes_per_bfs_layer(pos),
expected_nodes_per_layer)
401: (4)                pos = nx.bfs_layout(G, start=12)
402: (4)                assert np.array_equal(_num_nodes_per_bfs_layer(pos),
expected_nodes_per_layer)
403: (4)                pos = nx.bfs_layout(G, start=6)
404: (4)                expected_nodes_per_layer = [1, 2, 2, 8]
405: (4)                assert np.array_equal(_num_nodes_per_bfs_layer(pos),
expected_nodes_per_layer)
406: (0)            def test_bfs_layout_disconnected():
407: (4)                G = nx.complete_graph(5)
408: (4)                G.add_edges_from([(10, 11), (11, 12)])
409: (4)                with pytest.raises(nx.NetworkXError, match="bfs_layout didn't include all
nodes"):
410: (8)                    nx.bfs_layout(G, start=0)

----------------------------------------

File 424 - test_pydot.py:

1: (0)              """Unit tests for pydot drawing functions."""
2: (0)              from io import StringIO
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.utils import graphs_equal
6: (0)              pydot = pytest.importorskip("pydot")
```

```
7: (0)              class TestPydot:
8: (4)                  @pytest.mark.parametrize("G", (nx.Graph(), nx.DiGraph()))
9: (4)                  @pytest.mark.parametrize("prog", ("neato", "dot"))
10: (4)                 def test_pydot(self, G, prog, tmp_path):
11: (8)                     """
12: (8)                     Validate :mod:`pydot`-based usage of the passed NetworkX graph with
the
13: (8)                     passed basename of an external GraphViz command (e.g., `dot`,
`neato`).
14: (8)                     """
15: (8)                     G.graph["name"] = "G"
16: (8)                     G.add_edges_from([("A", "B"), ("A", "C"), ("B", "C"), ("A", "D")])
17: (8)                     G.add_node("E")
18: (8)                     graph_layout = nx.nx_pydot.pydot_layout(G, prog=prog)
19: (8)                     assert isinstance(graph_layout, dict)
20: (8)                     P = nx.nx_pydot.to_pydot(G)
21: (8)                     G2 = G.__class__(nx.nx_pydot.from_pydot(P))
22: (8)                     assert graphs_equal(G, G2)
23: (8)                     fname = tmp_path / "out.dot"
24: (8)                     P.write_raw(fname)
25: (8)                     Pin_list = pydot.graph_from_dot_file(path=fname, encoding="utf-8")
26: (8)                     assert len(Pin_list) == 1
27: (8)                     Pin = Pin_list[0]
28: (8)                     n1 = sorted(p.get_name() for p in P.get_node_list())
29: (8)                     n2 = sorted(p.get_name() for p in Pin.get_node_list())
30: (8)                     assert n1 == n2
31: (8)                     e1 = sorted((e.get_source(), e.get_destination()) for e in
P.get_edge_list())
32: (8)                     e2 = sorted((e.get_source(), e.get_destination()) for e in
Pin.get_edge_list())
33: (8)                     assert e1 == e2
34: (8)                     Hin = nx.nx_pydot.read_dot(fname)
35: (8)                     Hin = G.__class__(Hin)
36: (8)                     assert graphs_equal(G, Hin)
37: (4)                 def test_read_write(self):
38: (8)                     G = nx.MultiGraph()
39: (8)                     G.graph["name"] = "G"
40: (8)                     G.add_edge("1", "2", key="0")  # read assumes strings
41: (8)                     fh = StringIO()
42: (8)                     nx.nx_pydot.write_dot(G, fh)
43: (8)                     fh.seek(0)
44: (8)                     H = nx.nx_pydot.read_dot(fh)
45: (8)                     assert graphs_equal(G, H)
46: (0)             def test_pydot_issue_258():
47: (4)                 G = nx.Graph([("Example:A", 1)])
48: (4)                 with pytest.raises(ValueError):
49: (8)                     nx.nx_pydot.to_pydot(G)
50: (4)                 with pytest.raises(ValueError):
51: (8)                     nx.nx_pydot.pydot_layout(G)
52: (4)                 G = nx.Graph()
53: (4)                 G.add_node("1.2", style="filled", fillcolor="red:yellow")
54: (4)                 with pytest.raises(ValueError):
55: (8)                     nx.nx_pydot.to_pydot(G)
56: (4)                 G.remove_node("1.2")
57: (4)                 G.add_node("1.2", style="filled", fillcolor='"red:yellow"')
58: (4)                 assert (
59: (8)                     G.nodes.data() ==
nx.nx_pydot.from_pydot(nx.nx_pydot.to_pydot(G)).nodes.data()
60: (4)                 )
61: (4)                 G = nx.DiGraph()
62: (4)                 G.add_edge("1", "2", foo="bar:1")
63: (4)                 with pytest.raises(ValueError):
64: (8)                     nx.nx_pydot.to_pydot(G)
65: (4)                 G = nx.DiGraph()
66: (4)                 G.add_edge("1", "2", foo='"bar:1"')
67: (4)                 assert G["1"]["2"] == nx.nx_pydot.from_pydot(nx.nx_pydot.to_pydot(G))["1"]
["2"]
68: (4)                 G = nx.MultiGraph()
69: (4)                 G.add_edge("1", "2", foo="b:1")
```

```
70: (4)                 G.add_edge("1", "2", bar="foo:foo")
71: (4)                 with pytest.raises(ValueError):
72: (8)                     nx.nx_pydot.to_pydot(G)
73: (4)                 G = nx.MultiGraph()
74: (4)                 G.add_edge("1", "2", foo='"b:1"')
75: (4)                 G.add_edge("1", "2", bar='"foo:foo"')
76: (4)                 assert [attr for _, _, attr in G.edges.data()] == [
77: (8)                     attr
78: (8)                     for _, _, attr in
nx.nx_pydot.from_pydot(nx.nx_pydot.to_pydot(G)).edges.data()
79: (4)                 ]
80: (4)                 G = nx.Graph()
81: (4)                 G.add_edge("1", "2")
82: (4)                 G["1"]["2"]["f:oo"] = "bar"
83: (4)                 with pytest.raises(ValueError):
84: (8)                     nx.nx_pydot.to_pydot(G)
85: (4)                 G = nx.Graph()
86: (4)                 G.add_edge("1", "2")
87: (4)                 G["1"]["2"]['"f:oo"'] = "bar"
88: (4)                 assert G["1"]["2"] == nx.nx_pydot.from_pydot(nx.nx_pydot.to_pydot(G))["1"]
["2"]
89: (4)                 G = nx.Graph([('"Example:A"', 1)])
90: (4)                 layout = nx.nx_pydot.pydot_layout(G)
91: (4)                 assert isinstance(layout, dict)
92: (0)             @pytest.mark.parametrize(
93: (4)                 "graph_type", [nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph]
94: (0)             )
95: (0)             def test_hashable_pydot(graph_type):
96: (4)                 G = graph_type()
97: (4)                 G.add_edge("5", frozenset([1]), t='"Example:A"', l=False)
98: (4)                 G.add_edge("1", 2, w=True, t=("node1",), l=frozenset(["node1"]))
99: (4)                 G.add_edge("node", (3, 3), w="string")
100: (4)                assert [
101: (8)                    {"t": '"Example:A"', "l": "False"},
102: (8)                    {"w": "True", "t": "('node1',)", "l": "frozenset({'node1'})"},
103: (8)                    {"w": "string"},
104: (4)                ] == [
105: (8)                    attr
106: (8)                    for _, _, attr in
nx.nx_pydot.from_pydot(nx.nx_pydot.to_pydot(G)).edges.data()
107: (4)                ]
108: (4)                assert {str(i) for i in G.nodes()} == set(
109: (8)                    nx.nx_pydot.from_pydot(nx.nx_pydot.to_pydot(G)).nodes
110: (4)                )
111: (0)            def test_pydot_numerical_name():
112: (4)                G = nx.Graph()
113: (4)                G.add_edges_from([("A", "B"), (0, 1)])
114: (4)                graph_layout = nx.nx_pydot.pydot_layout(G, prog="dot")
115: (4)                assert isinstance(graph_layout, dict)
116: (4)                assert "0" not in graph_layout
117: (4)                assert 0 in graph_layout
118: (4)                assert "1" not in graph_layout
119: (4)                assert 1 in graph_layout
120: (4)                assert "A" in graph_layout
121: (4)                assert "B" in graph_layout
```

File 425 - test_pylab.py:

```
1: (0)              """Unit tests for matplotlib drawing functions."""
2: (0)              import itertools
3: (0)              import os
4: (0)              import warnings
5: (0)              import pytest
6: (0)              mpl = pytest.importorskip("matplotlib")
7: (0)              np = pytest.importorskip("numpy")
8: (0)              mpl.use("PS")
9: (0)              plt = pytest.importorskip("matplotlib.pyplot")
```

```
(0)             plt.rcParams["text.usetex"] = False
(0)             import networkx as nx
(0)             barbell = nx.barbell_graph(4, 6)
(0)             def test_draw():
(4)                 try:
(8)                     functions = [
(12)                        nx.draw_circular,
(12)                        nx.draw_kamada_kawai,
(12)                        nx.draw_planar,
(12)                        nx.draw_random,
(12)                        nx.draw_spectral,
(12)                        nx.draw_spring,
(12)                        nx.draw_shell,
(8)                     ]
(8)                     options = [{"node_color": "black", "node_size": 100, "width": 3}]
(8)                     for function, option in itertools.product(functions, options):
(12)                        function(barbell, **option)
(12)                        plt.savefig("test.ps")
(4)                 except ModuleNotFoundError:  # draw_kamada_kawai requires scipy
(8)                     pass
(4)                 finally:
(8)                     try:
(12)                        os.unlink("test.ps")
(8)                     except OSError:
(12)                        pass
(0)             def test_draw_shell_nlist():
(4)                 try:
(8)                     nlist = [list(range(4)), list(range(4, 10)), list(range(10, 14))]
(8)                     nx.draw_shell(barbell, nlist=nlist)
(8)                     plt.savefig("test.ps")
(4)                 finally:
(8)                     try:
(12)                        os.unlink("test.ps")
(8)                     except OSError:
(12)                        pass
(0)             def test_edge_colormap():
(4)                 colors = range(barbell.number_of_edges())
(4)                 nx.draw_spring(
(8)                     barbell, edge_color=colors, width=4, edge_cmap=plt.cm.Blues,
with_labels=True
(4)                 )
(0)             def test_arrows():
(4)                 nx.draw_spring(barbell.to_directed())
(0)             @pytest.mark.parametrize(
(4)                 ("edge_color", "expected"),
(4)                 (
(8)                     (None, "black"),  # Default
(8)                     ("r", "red"),  # Non-default color string
(8)                     (["r"], "red"),  # Single non-default color in a list
(8)                     ((1.0, 1.0, 0.0), "yellow"),  # single color as rgb tuple
(8)                     ([(1.0, 1.0, 0.0)], "yellow"),  # single color as rgb tuple in list
(8)                     ((0, 1, 0, 1), "lime"),  # single color as rgba tuple
(8)                     ([(0, 1, 0, 1)], "lime"),  # single color as rgba tuple in list
(8)                     ("#0000ff", "blue"),  # single color hex code
(8)                     (["#0000ff"], "blue"),  # hex code in list
(4)                 ),
(0)             )
(0)             @pytest.mark.parametrize("edgelist", (None, [(0, 1)]))
(0)             def test_single_edge_color_undirected(edge_color, expected, edgelist):
(4)                 """Tests ways of specifying all edges have a single color for edges
(4)                 drawn with a LineCollection"""
(4)                 G = nx.path_graph(3)
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos=nx.random_layout(G), edgelist=edgelist, edge_color=edge_color
(4)                 )
(4)                 assert mpl.colors.same_color(drawn_edges.get_color(), expected)
(0)             @pytest.mark.parametrize(
(4)                 ("edge_color", "expected"),
(4)                 (
```

```
(8)                     (None, "black"),  # Default
(8)                     ("r", "red"),  # Non-default color string
(8)                     (["r"], "red"),  # Single non-default color in a list
(8)                     ((1.0, 1.0, 0.0), "yellow"),  # single color as rgb tuple
(8)                     ([(1.0, 1.0, 0.0)], "yellow"),  # single color as rgb tuple in list
(8)                     ((0, 1, 0, 1), "lime"),  # single color as rgba tuple
(8)                     ([(0, 1, 0, 1)], "lime"),  # single color as rgba tuple in list
(8)                     ("#0000ff", "blue"),  # single color hex code
(8)                     (["#0000ff"], "blue"),  # hex code in list
(4)                 ),
(0)             )
(0)             @pytest.mark.parametrize("edgelist", (None, [(0, 1)]))
(0)             def test_single_edge_color_directed(edge_color, expected, edgelist):
(4)                 """Tests ways of specifying all edges have a single color for edges drawn
(4)                 with FancyArrowPatches"""
(4)                 G = nx.path_graph(3, create_using=nx.DiGraph)
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos=nx.random_layout(G), edgelist=edgelist, edge_color=edge_color
(4)                 )
(4)                 for fap in drawn_edges:
(8)                     assert mpl.colors.same_color(fap.get_edgecolor(), expected)
(0)             def test_edge_color_tuple_interpretation():
(4)                 """If edge_color is a sequence with the same length as edgelist, then each
(4)                 value in edge_color is mapped onto each edge via colormap."""
(4)                 G = nx.path_graph(6, create_using=nx.DiGraph)
(4)                 pos = {n: (n, n) for n in range(len(G))}
(4)                 for ec in ((0, 0, 1), (0, 0, 1, 1)):
(8)                     drawn_edges = nx.draw_networkx_edges(G, pos, edge_color=ec)
(8)                     for fap in drawn_edges:
(12)                        assert mpl.colors.same_color(fap.get_edgecolor(), ec)
(8)                     drawn_edges = nx.draw_networkx_edges(
(12)                        G, pos, edgelist=[(0, 1), (1, 2)], edge_color=ec
(8)                     )
(8)                     for fap in drawn_edges:
(12)                        assert mpl.colors.same_color(fap.get_edgecolor(), ec)
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos, edgelist=[(0, 1), (1, 2), (2, 3)], edge_color=(0, 0, 1, 1)
(4)                 )
(4)                 for fap in drawn_edges:
(8)                     assert mpl.colors.same_color(fap.get_edgecolor(), "blue")
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos, edgelist=[(0, 1), (1, 2), (2, 3), (3, 4)], edge_color=(0, 0,
1)
(4)                 )
(4)                 for fap in drawn_edges:
(8)                     assert mpl.colors.same_color(fap.get_edgecolor(), "blue")
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos, edgelist=[(0, 1), (1, 2), (2, 3)], edge_color=(0, 0, 1)
(4)                 )
(4)                 assert mpl.colors.same_color(
(8)                     drawn_edges[0].get_edgecolor(), drawn_edges[1].get_edgecolor()
(4)                 )
(4)                 for fap in drawn_edges:
(8)                     assert not mpl.colors.same_color(fap.get_edgecolor(), "blue")
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos, edgelist=[(0, 1), (1, 2), (2, 3), (3, 4)], edge_color=(0, 0,
1, 1)
(4)                 )
(4)                 assert mpl.colors.same_color(
(8)                     drawn_edges[0].get_edgecolor(), drawn_edges[1].get_edgecolor()
(4)                 )
(4)                 assert mpl.colors.same_color(
(8)                     drawn_edges[2].get_edgecolor(), drawn_edges[3].get_edgecolor()
(4)                 )
(4)                 for fap in drawn_edges:
(8)                     assert not mpl.colors.same_color(fap.get_edgecolor(), "blue")
(0)             def test_fewer_edge_colors_than_num_edges_directed():
(4)                 """Test that the edge colors are cycled when there are fewer specified
(4)                 colors than edges."""
```

```
(4)                 G = barbell.to_directed()
(4)                 pos = nx.random_layout(barbell)
(4)                 edgecolors = ("r", "g", "b")
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, edge_color=edgecolors)
(4)                 for fap, expected in zip(drawn_edges, itertools.cycle(edgecolors)):
(8)                     assert mpl.colors.same_color(fap.get_edgecolor(), expected)
(0)             def test_more_edge_colors_than_num_edges_directed():
(4)                 """Test that extra edge colors are ignored when there are more specified
(4)                 colors than edges."""
(4)                 G = nx.path_graph(4, create_using=nx.DiGraph)  # 3 edges
(4)                 pos = nx.random_layout(barbell)
(4)                 edgecolors = ("r", "g", "b", "c")  # 4 edge colors
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, edge_color=edgecolors)
(4)                 for fap, expected in zip(drawn_edges, edgecolors[:-1]):
(8)                     assert mpl.colors.same_color(fap.get_edgecolor(), expected)
(0)             def test_edge_color_string_with_global_alpha_undirected():
(4)                 edge_collection = nx.draw_networkx_edges(
(8)                     barbell,
(8)                     pos=nx.random_layout(barbell),
(8)                     edgelist=[(0, 1), (1, 2)],
(8)                     edge_color="purple",
(8)                     alpha=0.2,
(4)                 )
(4)                 ec = edge_collection.get_color().squeeze()  # as rgba tuple
(4)                 assert len(edge_collection.get_paths()) == 2
(4)                 assert mpl.colors.same_color(ec[:-1], "purple")
(4)                 assert ec[-1] == 0.2
(0)             def test_edge_color_string_with_global_alpha_directed():
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     barbell.to_directed(),
(8)                     pos=nx.random_layout(barbell),
(8)                     edgelist=[(0, 1), (1, 2)],
(8)                     edge_color="purple",
(8)                     alpha=0.2,
(4)                 )
(4)                 assert len(drawn_edges) == 2
(4)                 for fap in drawn_edges:
(8)                     ec = fap.get_edgecolor()  # As rgba tuple
(8)                     assert mpl.colors.same_color(ec[:-1], "purple")
(8)                     assert ec[-1] == 0.2
(0)             @pytest.mark.parametrize("graph_type", (nx.Graph, nx.DiGraph))
(0)             def test_edge_width_default_value(graph_type):
(4)                 """Test the default linewidth for edges drawn either via LineCollection or
(4)                 FancyArrowPatches."""
(4)                 G = nx.path_graph(2, create_using=graph_type)
(4)                 pos = {n: (n, n) for n in range(len(G))}
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos)
(4)                 if isinstance(drawn_edges, list):  # directed case: list of
FancyArrowPatch
(8)                     drawn_edges = drawn_edges[0]
(4)                 assert drawn_edges.get_linewidth() == 1
(0)             @pytest.mark.parametrize(
(4)                 ("edgewidth", "expected"),
(4)                 (
(8)                     (3, 3),  # single-value, non-default
(8)                     ([3], 3),  # Single value as a list
(4)                 ),
(0)             )
(0)             def test_edge_width_single_value_undirected(edgewidth, expected):
(4)                 G = nx.path_graph(4)
(4)                 pos = {n: (n, n) for n in range(len(G))}
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, width=edgewidth)
(4)                 assert len(drawn_edges.get_paths()) == 3
(4)                 assert drawn_edges.get_linewidth() == expected
(0)             @pytest.mark.parametrize(
(4)                 ("edgewidth", "expected"),
(4)                 (
(8)                     (3, 3),  # single-value, non-default
(8)                     ([3], 3),  # Single value as a list
```

```
(4)                 ),
(0)             )
(0)             def test_edge_width_single_value_directed(edgewidth, expected):
(4)                 G = nx.path_graph(4, create_using=nx.DiGraph)
(4)                 pos = {n: (n, n) for n in range(len(G))}
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, width=edgewidth)
(4)                 assert len(drawn_edges) == 3
(4)                 for fap in drawn_edges:
(8)                     assert fap.get_linewidth() == expected
(0)             @pytest.mark.parametrize(
(4)                 "edgelist",
(4)                 (
(8)                     [(0, 1), (1, 2), (2, 3)],  # one width specification per edge
(8)                     None,  #  fewer widths than edges - widths cycle
(8)                     [(0, 1), (1, 2)],  # More widths than edges - unused widths ignored
(4)                 ),
(0)             )
(0)             def test_edge_width_sequence(edgelist):
(4)                 G = barbell.to_directed()
(4)                 pos = nx.random_layout(G)
(4)                 widths = (0.5, 2.0, 12.0)
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, edgelist=edgelist,
width=widths)
(4)                 for fap, expected_width in zip(drawn_edges, itertools.cycle(widths)):
(8)                     assert fap.get_linewidth() == expected_width
(0)             def test_edge_color_with_edge_vmin_vmax():
(4)                 """Test that edge_vmin and edge_vmax properly set the dynamic range of the
(4)                 color map when num edges == len(edge_colors)."""
(4)                 G = nx.path_graph(3, create_using=nx.DiGraph)
(4)                 pos = nx.random_layout(G)
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, edge_color=[0, 1.0])
(4)                 orig_colors = [e.get_edgecolor() for e in drawn_edges]
(4)                 drawn_edges = nx.draw_networkx_edges(
(8)                     G, pos, edge_color=[0.2, 0.8], edge_vmin=0.2, edge_vmax=0.8
(4)                 )
(4)                 scaled_colors = [e.get_edgecolor() for e in drawn_edges]
(4)                 assert mpl.colors.same_color(orig_colors, scaled_colors)
(0)             def test_directed_edges_linestyle_default():
(4)                 """Test default linestyle for edges drawn with FancyArrowPatches."""
(4)                 G = nx.path_graph(4, create_using=nx.DiGraph)  # Graph with 3 edges
(4)                 pos = {n: (n, n) for n in range(len(G))}
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos)
(4)                 assert len(drawn_edges) == 3
(4)                 for fap in drawn_edges:
(8)                     assert fap.get_linestyle() == "solid"
(0)             @pytest.mark.parametrize(
(4)                 "style",
(4)                 (
(8)                     "dashed",  # edge with string style
(8)                     "--",  # edge with simplified string style
(8)                     (1, (1, 1)),  # edge with (offset, onoffseq) style
(4)                 ),
(0)             )
(0)             def test_directed_edges_linestyle_single_value(style):
(4)                 """Tests support for specifying linestyles with a single value to be
applied to
(4)                 all edges in ``draw_networkx_edges`` for FancyArrowPatch outputs
(4)                 (e.g. directed edges)."""
(4)                 G = nx.path_graph(4, create_using=nx.DiGraph)  # Graph with 3 edges
(4)                 pos = {n: (n, n) for n in range(len(G))}
(4)                 drawn_edges = nx.draw_networkx_edges(G, pos, style=style)
(4)                 assert len(drawn_edges) == 3
(4)                 for fap in drawn_edges:
(8)                     assert fap.get_linestyle() == style
(0)             @pytest.mark.parametrize(
(4)                 "style_seq",
(4)                 (
(8)                     ["dashed"],  # edge with string style in list
(8)                     ["--"],  # edge with simplified string style in list
```

```
280: (8)                    [(1, (1, 1))],  # edge with (offset, onoffseq) style in list
281: (8)                    ["--", "-", ":"],  # edges with styles for each edge
282: (8)                    ["--", "-"],  # edges with fewer styles than edges (styles cycle)
283: (8)                    ["--", "-", ":", "-."],  # edges with more styles than edges (extra
unused)
284: (4)                ),
285: (0)            )
286: (0)            def test_directed_edges_linestyle_sequence(style_seq):
287: (4)                """Tests support for specifying linestyles with sequences in
288: (4)                ``draw_networkx_edges`` for FancyArrowPatch outputs (e.g. directed
edges)."""
289: (4)                G = nx.path_graph(4, create_using=nx.DiGraph)  # Graph with 3 edges
290: (4)                pos = {n: (n, n) for n in range(len(G))}
291: (4)                drawn_edges = nx.draw_networkx_edges(G, pos, style=style_seq)
292: (4)                assert len(drawn_edges) == 3
293: (4)                for fap, style in zip(drawn_edges, itertools.cycle(style_seq)):
294: (8)                    assert fap.get_linestyle() == style
295: (0)            def test_labels_and_colors():
296: (4)                G = nx.cubical_graph()
297: (4)                pos = nx.spring_layout(G)  # positions for all nodes
298: (4)                nx.draw_networkx_nodes(
299: (8)                    G, pos, nodelist=[0, 1, 2, 3], node_color="r", node_size=500,
alpha=0.75
300: (4)                )
301: (4)                nx.draw_networkx_nodes(
302: (8)                    G,
303: (8)                    pos,
304: (8)                    nodelist=[4, 5, 6, 7],
305: (8)                    node_color="b",
306: (8)                    node_size=500,
307: (8)                    alpha=[0.25, 0.5, 0.75, 1.0],
308: (4)                )
309: (4)                nx.draw_networkx_edges(G, pos, width=1.0, alpha=0.5)
310: (4)                nx.draw_networkx_edges(
311: (8)                    G,
312: (8)                    pos,
313: (8)                    edgelist=[(0, 1), (1, 2), (2, 3), (3, 0)],
314: (8)                    width=8,
315: (8)                    alpha=0.5,
316: (8)                    edge_color="r",
317: (4)                )
318: (4)                nx.draw_networkx_edges(
319: (8)                    G,
320: (8)                    pos,
321: (8)                    edgelist=[(4, 5), (5, 6), (6, 7), (7, 4)],
322: (8)                    width=8,
323: (8)                    alpha=0.5,
324: (8)                    edge_color="b",
325: (4)                )
326: (4)                nx.draw_networkx_edges(
327: (8)                    G,
328: (8)                    pos,
329: (8)                    edgelist=[(4, 5), (5, 6), (6, 7), (7, 4)],
330: (8)                    arrows=True,
331: (8)                    min_source_margin=0.5,
332: (8)                    min_target_margin=0.75,
333: (8)                    width=8,
334: (8)                    edge_color="b",
335: (4)                )
336: (4)                labels = {}
337: (4)                labels[0] = r"$a$"
338: (4)                labels[1] = r"$b$"
339: (4)                labels[2] = r"$c$"
340: (4)                labels[3] = r"$d$"
341: (4)                labels[4] = r"$\alpha$"
342: (4)                labels[5] = r"$\beta$"
343: (4)                labels[6] = r"$\gamma$"
344: (4)                labels[7] = r"$\delta$"
345: (4)                nx.draw_networkx_labels(G, pos, labels, font_size=16)
```

```
346: (4)                nx.draw_networkx_edge_labels(G, pos, edge_labels=None, rotate=False)
347: (4)                nx.draw_networkx_edge_labels(G, pos, edge_labels={(4, 5): "4-5"})
348: (0)            @pytest.mark.mpl_image_compare
349: (0)            def test_house_with_colors():
350: (4)                G = nx.house_graph()
351: (4)                fig, ax = plt.subplots()
352: (4)                pos = {0: (0, 0), 1: (1, 0), 2: (0, 1), 3: (1, 1), 4: (0.5, 2.0)}
353: (4)                nx.draw_networkx_nodes(
354: (8)                    G,
355: (8)                    pos,
356: (8)                    node_size=3000,
357: (8)                    nodelist=[0, 1, 2, 3],
358: (8)                    node_color="tab:blue",
359: (4)                )
360: (4)                nx.draw_networkx_nodes(
361: (8)                    G, pos, node_size=2000, nodelist=[4], node_color="tab:orange"
362: (4)                )
363: (4)                nx.draw_networkx_edges(G, pos, alpha=0.5, width=6)
364: (4)                ax.margins(0.11)
365: (4)                plt.tight_layout()
366: (4)                plt.axis("off")
367: (4)                return fig
368: (0)            def test_axes():
369: (4)                fig, ax = plt.subplots()
370: (4)                nx.draw(barbell, ax=ax)
371: (4)                nx.draw_networkx_edge_labels(barbell, nx.circular_layout(barbell), ax=ax)
372: (0)            def test_empty_graph():
373: (4)                G = nx.Graph()
374: (4)                nx.draw(G)
375: (0)            def test_draw_empty_nodes_return_values():
376: (4)                import matplotlib.collections  # call as mpl.collections
377: (4)                G = nx.Graph([(1, 2), (2, 3)])
378: (4)                DG = nx.DiGraph([(1, 2), (2, 3)])
379: (4)                pos = nx.circular_layout(G)
380: (4)                assert isinstance(
381: (8)                    nx.draw_networkx_nodes(G, pos, nodelist=[]),
mpl.collections.PathCollection
382: (4)                )
383: (4)                assert isinstance(
384: (8)                    nx.draw_networkx_nodes(DG, pos, nodelist=[]),
mpl.collections.PathCollection
385: (4)                )
386: (4)                assert nx.draw_networkx_edges(G, pos, edgelist=[], arrows=True) == []
387: (4)                assert nx.draw_networkx_edges(G, pos, edgelist=[], arrows=False) == []
388: (4)                assert nx.draw_networkx_edges(DG, pos, edgelist=[], arrows=False) == []
389: (4)                assert nx.draw_networkx_edges(DG, pos, edgelist=[], arrows=True) == []
390: (0)            def test_multigraph_edgelist_tuples():
391: (4)                G = nx.path_graph(3, create_using=nx.MultiDiGraph)
392: (4)                nx.draw_networkx(G, edgelist=[(0, 1, 0)])
393: (4)                nx.draw_networkx(G, edgelist=[(0, 1, 0)], node_size=[10, 20, 0])
394: (0)            def test_alpha_iter():
395: (4)                pos = nx.random_layout(barbell)
396: (4)                fig = plt.figure()
397: (4)                fig.add_subplot(131)  # Each test in a new axis object
398: (4)                nx.draw_networkx_nodes(barbell, pos, alpha=[0.1, 0.2])
399: (4)                num_nodes = len(barbell.nodes)
400: (4)                alpha = [x / num_nodes for x in range(num_nodes)]
401: (4)                colors = range(num_nodes)
402: (4)                fig.add_subplot(132)
403: (4)                nx.draw_networkx_nodes(barbell, pos, node_color=colors, alpha=alpha)
404: (4)                alpha.append(1)
405: (4)                fig.add_subplot(133)
406: (4)                nx.draw_networkx_nodes(barbell, pos, alpha=alpha)
407: (0)            def test_error_invalid_kwds():
408: (4)                with pytest.raises(ValueError, match="Received invalid argument"):
409: (8)                    nx.draw(barbell, foo="bar")
410: (0)            def test_draw_networkx_arrowsize_incorrect_size():
411: (4)                G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 3)])
412: (4)                arrowsize = [1, 2, 3]
```

```
(4)                 with pytest.raises(
(8)                     ValueError, match="arrowsize should have the same length as edgelist"
(4)                 ):
(8)                     nx.draw(G, arrowsize=arrowsize)
(0)             @pytest.mark.parametrize("arrowsize", (30, [10, 20, 30]))
(0)             def test_draw_edges_arrowsize(arrowsize):
(4)                 G = nx.DiGraph([(0, 1), (0, 2), (1, 2)])
(4)                 pos = {0: (0, 0), 1: (0, 1), 2: (1, 0)}
(4)                 edges = nx.draw_networkx_edges(G, pos=pos, arrowsize=arrowsize)
(4)                 arrowsize = itertools.repeat(arrowsize) if isinstance(arrowsize, int) else
arrowsize
(4)                 for fap, expected in zip(edges, arrowsize):
(8)                     assert isinstance(fap, mpl.patches.FancyArrowPatch)
(8)                     assert fap.get_mutation_scale() == expected
(0)             def test_np_edgelist():
(4)                 nx.draw_networkx(barbell, edgelist=np.array([(0, 2), (0, 3)]))
(0)             def test_draw_nodes_missing_node_from_position():
(4)                 G = nx.path_graph(3)
(4)                 pos = {0: (0, 0), 1: (1, 1)}  # No position for node 2
(4)                 with pytest.raises(nx.NetworkXError, match="has no position"):
(8)                     nx.draw_networkx_nodes(G, pos)
(0)             @pytest.mark.parametrize("node_shape", ("o", "s"))
(0)             def test_draw_edges_min_source_target_margins(node_shape):
(4)                 """Test that there is a wider gap between the node and the start of an
(4)                 incident edge when min_source_margin is specified.
(4)                 This test checks that the use of min_{source/target}_margin kwargs result
(4)                 in shorter (more padding) between the edges and source and target nodes.
(4)                 As a crude visual example, let 's' and 't' represent source and target
(4)                 nodes, respectively:
(7)                    Default:
(7)                    s-----------------------------t
(7)                    With margins:
(7)                    s   ------------------------   t
(4)                 """
(4)                 fig, ax = plt.subplots()
(4)                 G = nx.DiGraph([(0, 1)])
(4)                 pos = {0: (0, 0), 1: (1, 0)}  # horizontal layout
(4)                 default_patch = nx.draw_networkx_edges(G, pos, ax=ax,
node_shape=node_shape)[0]
(4)                 default_extent = default_patch.get_extents().corners()[::2, 0]
(4)                 padded_patch = nx.draw_networkx_edges(
(8)                     G,
(8)                     pos,
(8)                     ax=ax,
(8)                     node_shape=node_shape,
(8)                     min_source_margin=100,
(8)                     min_target_margin=100,
(4)                 )[0]
(4)                 padded_extent = padded_patch.get_extents().corners()[::2, 0]
(4)                 assert padded_extent[0] > default_extent[0]
(4)                 assert padded_extent[1] < default_extent[1]
(0)             def test_nonzero_selfloop_with_single_node():
(4)                 """Ensure that selfloop extent is non-zero when there is only one node."""
(4)                 fig, ax = plt.subplots()
(4)                 G = nx.DiGraph()
(4)                 G.add_node(0)
(4)                 G.add_edge(0, 0)
(4)                 patch = nx.draw_networkx_edges(G, {0: (0, 0)})[0]
(4)                 bbox = patch.get_extents()
(4)                 assert bbox.width > 0 and bbox.height > 0
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             def test_nonzero_selfloop_with_single_edge_in_edgelist():
(4)                 """Ensure that selfloop extent is non-zero when only a single edge is
(4)                 specified in the edgelist.
(4)                 """
(4)                 fig, ax = plt.subplots()
(4)                 G = nx.path_graph(2, create_using=nx.DiGraph)
(4)                 G.add_edge(1, 1)
```

```
(4)                 pos = {n: (n, n) for n in G.nodes}
(4)                 patch = nx.draw_networkx_edges(G, pos, edgelist=[(1, 1)])[0]
(4)                 bbox = patch.get_extents()
(4)                 assert bbox.width > 0 and bbox.height > 0
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             def test_apply_alpha():
(4)                 """Test apply_alpha when there is a mismatch between the number of
(4)                 supplied colors and elements.
(4)                 """
(4)                 nodelist = [0, 1, 2]
(4)                 colorlist = ["r", "g", "b"]
(4)                 alpha = 0.5
(4)                 rgba_colors = nx.drawing.nx_pylab.apply_alpha(colorlist, alpha, nodelist)
(4)                 assert all(rgba_colors[:, -1] == alpha)
(0)             def test_draw_edges_toggling_with_arrows_kwarg():
(4)                 """
(4)                 The `arrows` keyword argument is used as a 3-way switch to select which
(4)                 type of object to use for drawing edges:
(6)                   - ``arrows=None`` -> default (FancyArrowPatches for directed, else
LineCollection)
(6)                   - ``arrows=True`` -> FancyArrowPatches
(6)                   - ``arrows=False`` -> LineCollection
(4)                 """
(4)                 import matplotlib.collections
(4)                 import matplotlib.patches
(4)                 UG = nx.path_graph(3)
(4)                 DG = nx.path_graph(3, create_using=nx.DiGraph)
(4)                 pos = {n: (n, n) for n in UG}
(4)                 for G in (UG, DG):
(8)                     edges = nx.draw_networkx_edges(G, pos, arrows=True)
(8)                     assert len(edges) == len(G.edges)
(8)                     assert isinstance(edges[0], mpl.patches.FancyArrowPatch)
(4)                 for G in (UG, DG):
(8)                     edges = nx.draw_networkx_edges(G, pos, arrows=False)
(8)                     assert isinstance(edges, mpl.collections.LineCollection)
(4)                 edges = nx.draw_networkx_edges(UG, pos)
(4)                 assert isinstance(edges, mpl.collections.LineCollection)
(4)                 edges = nx.draw_networkx_edges(DG, pos)
(4)                 assert len(edges) == len(G.edges)
(4)                 assert isinstance(edges[0], mpl.patches.FancyArrowPatch)
(0)             @pytest.mark.parametrize("drawing_func", (nx.draw, nx.draw_networkx))
(0)             def test_draw_networkx_arrows_default_undirected(drawing_func):
(4)                 import matplotlib.collections
(4)                 G = nx.path_graph(3)
(4)                 fig, ax = plt.subplots()
(4)                 drawing_func(G, ax=ax)
(4)                 assert any(isinstance(c, mpl.collections.LineCollection) for c in
ax.collections)
(4)                 assert not ax.patches
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             @pytest.mark.parametrize("drawing_func", (nx.draw, nx.draw_networkx))
(0)             def test_draw_networkx_arrows_default_directed(drawing_func):
(4)                 import matplotlib.collections
(4)                 G = nx.path_graph(3, create_using=nx.DiGraph)
(4)                 fig, ax = plt.subplots()
(4)                 drawing_func(G, ax=ax)
(4)                 assert not any(
(8)                     isinstance(c, mpl.collections.LineCollection) for c in ax.collections
(4)                 )
(4)                 assert ax.patches
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             def test_edgelist_kwarg_not_ignored():
(4)                 G = nx.path_graph(3)
(4)                 G.add_edge(0, 0)
(4)                 fig, ax = plt.subplots()
(4)                 nx.draw(G, edgelist=[(0, 1), (1, 2)], ax=ax)  # Exclude self-loop from
```

```
edgelist
547: (4)                assert not ax.patches
548: (4)                plt.delaxes(ax)
549: (4)                plt.close()
550: (0)            @pytest.mark.parametrize(
551: (4)                ("G", "expected_n_edges"),
552: (4)                ([nx.DiGraph(), 2], [nx.MultiGraph(), 4], [nx.MultiDiGraph(), 4]),
553: (0)            )
554: (0)            def test_draw_networkx_edges_multiedge_connectionstyle(G, expected_n_edges):
555: (4)                """Draws edges correctly for 3 types of graphs and checks for valid
length"""
556: (4)                for i, (u, v) in enumerate([(0, 1), (0, 1), (0, 1), (0, 2)]):
557: (8)                    G.add_edge(u, v, weight=round(i / 3, 2))
558: (4)                pos = {n: (n, n) for n in G}
559: (4)                for conn_style in [
560: (8)                    "arc3,rad=0.1",
561: (8)                    ["arc3,rad=0.1", "arc3,rad=0.1"],
562: (8)                    ["arc3,rad=0.1", "arc3,rad=0.1", "arc3,rad=0.2"],
563: (4)                ]:
564: (8)                    nx.draw_networkx_edges(G, pos, connectionstyle=conn_style)
565: (8)                    arrows = nx.draw_networkx_edges(G, pos, connectionstyle=conn_style)
566: (8)                    assert len(arrows) == expected_n_edges
567: (0)            @pytest.mark.parametrize(
568: (4)                ("G", "expected_n_edges"),
569: (4)                ([nx.DiGraph(), 2], [nx.MultiGraph(), 4], [nx.MultiDiGraph(), 4]),
570: (0)            )
571: (0)            def test_draw_networkx_edge_labels_multiedge_connectionstyle(G,
expected_n_edges):
572: (4)                """Draws labels correctly for 3 types of graphs and checks for valid
length and class names"""
573: (4)                for i, (u, v) in enumerate([(0, 1), (0, 1), (0, 1), (0, 2)]):
574: (8)                    G.add_edge(u, v, weight=round(i / 3, 2))
575: (4)                pos = {n: (n, n) for n in G}
576: (4)                arrows = nx.draw_networkx_edges(
577: (8)                    G, pos, connectionstyle=["arc3,rad=0.1", "arc3,rad=0.1",
"arc3,rad=0.1"]
578: (4)                )
579: (4)                for conn_style in [
580: (8)                    "arc3,rad=0.1",
581: (8)                    ["arc3,rad=0.1", "arc3,rad=0.2"],
582: (8)                    ["arc3,rad=0.1", "arc3,rad=0.1", "arc3,rad=0.1"],
583: (4)                ]:
584: (8)                    text_items = nx.draw_networkx_edge_labels(G, pos,
connectionstyle=conn_style)
585: (8)                    assert len(text_items) == expected_n_edges
586: (8)                    for ti in text_items.values():
587: (12)                       assert ti.__class__.__name__ == "CurvedArrowText"
588: (0)            def test_draw_networkx_edge_label_multiedge():
589: (4)                G = nx.MultiGraph()
590: (4)                G.add_edge(0, 1, weight=10)
591: (4)                G.add_edge(0, 1, weight=20)
592: (4)                edge_labels = nx.get_edge_attributes(G, "weight")  # Includes edge keys
593: (4)                pos = {n: (n, n) for n in G}
594: (4)                text_items = nx.draw_networkx_edge_labels(
595: (8)                    G,
596: (8)                    pos,
597: (8)                    edge_labels=edge_labels,
598: (8)                    connectionstyle=["arc3,rad=0.1", "arc3,rad=0.2"],
599: (4)                )
600: (4)                assert len(text_items) == 2
601: (0)            def test_draw_networkx_edge_label_empty_dict():
602: (4)                """Regression test for draw_networkx_edge_labels with empty dict. See
603: (4)                gh-5372."""
604: (4)                G = nx.path_graph(3)
605: (4)                pos = {n: (n, n) for n in G.nodes}
606: (4)                assert nx.draw_networkx_edge_labels(G, pos, edge_labels={}) == {}
607: (0)            def test_draw_networkx_edges_undirected_selfloop_colors():
608: (4)                """When an edgelist is supplied along with a sequence of colors, check
that
```

```
(4)                 the self-loops have the correct colors."""
(4)                 fig, ax = plt.subplots()
(4)                 edgelist = [(1, 3), (1, 2), (2, 3), (1, 1), (3, 3), (2, 2)]
(4)                 edge_colors = ["pink", "cyan", "black", "red", "blue", "green"]
(4)                 G = nx.Graph(edgelist)
(4)                 pos = {n: (n, n) for n in G.nodes}
(4)                 nx.draw_networkx_edges(G, pos, ax=ax, edgelist=edgelist,
edge_color=edge_colors)
(4)                 assert len(ax.patches) == 3
(4)                 sl_points = np.array(edgelist[-3:]) + np.array([0, 0.1])
(4)                 for fap, clr, slp in zip(ax.patches, edge_colors[-3:], sl_points):
(8)                     assert fap.get_path().contains_point(slp)
(8)                     assert mpl.colors.same_color(fap.get_edgecolor(), clr)
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             @pytest.mark.parametrize(
(4)                 "fap_only_kwarg",  # Non-default values for kwargs that only apply to FAPs
(4)                 (
(8)                     {"arrowstyle": "-"},
(8)                     {"arrowsize": 20},
(8)                     {"connectionstyle": "arc3,rad=0.2"},
(8)                     {"min_source_margin": 10},
(8)                     {"min_target_margin": 10},
(4)                 ),
(0)             )
(0)             def test_user_warnings_for_unused_edge_drawing_kwargs(fap_only_kwarg):
(4)                 """Users should get a warning when they specify a non-default value for
(4)                 one of the kwargs that applies only to edges drawn with FancyArrowPatches,
(4)                 but FancyArrowPatches aren't being used under the hood."""
(4)                 G = nx.path_graph(3)
(4)                 pos = {n: (n, n) for n in G}
(4)                 fig, ax = plt.subplots()
(4)                 kwarg_name = list(fap_only_kwarg.keys())[0]
(4)                 with pytest.warns(
(8)                     UserWarning, match=f"\n\nThe {kwarg_name} keyword argument is not
applicable"
(4)                 ):
(8)                     nx.draw_networkx_edges(G, pos, ax=ax, **fap_only_kwarg)
(4)                 with warnings.catch_warnings():
(8)                     warnings.simplefilter("error")
(8)                     nx.draw_networkx_edges(G, pos, ax=ax, arrows=True, **fap_only_kwarg)
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             @pytest.mark.parametrize("draw_fn", (nx.draw, nx.draw_circular))
(0)             def test_no_warning_on_default_draw_arrowstyle(draw_fn):
(4)                 fig, ax = plt.subplots()
(4)                 G = nx.cycle_graph(5)
(4)                 with warnings.catch_warnings(record=True) as w:
(8)                     draw_fn(G, ax=ax)
(4)                 assert len(w) == 0
(4)                 plt.delaxes(ax)
(4)                 plt.close()
(0)             @pytest.mark.parametrize("hide_ticks", [False, True])
(0)             @pytest.mark.parametrize(
(4)                 "method",
(4)                 [
(8)                     nx.draw_networkx,
(8)                     nx.draw_networkx_edge_labels,
(8)                     nx.draw_networkx_edges,
(8)                     nx.draw_networkx_labels,
(8)                     nx.draw_networkx_nodes,
(4)                 ],
(0)             )
(0)             def test_hide_ticks(method, hide_ticks):
(4)                 G = nx.path_graph(3)
(4)                 pos = {n: (n, n) for n in G.nodes}
(4)                 _, ax = plt.subplots()
(4)                 method(G, pos=pos, ax=ax, hide_ticks=hide_ticks)
(4)                 for axis in [ax.xaxis, ax.yaxis]:
```

```
676: (8)                    assert bool(axis.get_ticklabels()) != hide_ticks
677: (4)                plt.delaxes(ax)
678: (4)                plt.close()
```

File 426 - __init__.py:

```
1: (0)              """
2: (0)              A package for generating various graphs in networkx.
3: (0)              """
4: (0)              from networkx.generators.atlas import *
5: (0)              from networkx.generators.classic import *
6: (0)              from networkx.generators.cographs import *
7: (0)              from networkx.generators.community import *
8: (0)              from networkx.generators.degree_seq import *
9: (0)              from networkx.generators.directed import *
10: (0)             from networkx.generators.duplication import *
11: (0)             from networkx.generators.ego import *
12: (0)             from networkx.generators.expanders import *
13: (0)             from networkx.generators.geometric import *
14: (0)             from networkx.generators.harary_graph import *
15: (0)             from networkx.generators.internet_as_graphs import *
16: (0)             from networkx.generators.intersection import *
17: (0)             from networkx.generators.interval_graph import *
18: (0)             from networkx.generators.joint_degree_seq import *
19: (0)             from networkx.generators.lattice import *
20: (0)             from networkx.generators.line import *
21: (0)             from networkx.generators.mycielski import *
22: (0)             from networkx.generators.nonisomorphic_trees import *
23: (0)             from networkx.generators.random_clustered import *
24: (0)             from networkx.generators.random_graphs import *
25: (0)             from networkx.generators.small import *
26: (0)             from networkx.generators.social import *
27: (0)             from networkx.generators.spectral_graph_forge import *
28: (0)             from networkx.generators.stochastic import *
29: (0)             from networkx.generators.sudoku import *
30: (0)             from networkx.generators.time_series import *
31: (0)             from networkx.generators.trees import *
32: (0)             from networkx.generators.triads import *
```

File 427 - atlas.py:

```
1: (0)              """
2: (0)              Generators for the small graph atlas.
3: (0)              """
4: (0)              import gzip
5: (0)              import importlib.resources
6: (0)              import os
7: (0)              import os.path
8: (0)              from itertools import islice
9: (0)              import networkx as nx
10: (0)             __all__ = ["graph_atlas", "graph_atlas_g"]
11: (0)             NUM_GRAPHS = 1253
12: (0)             ATLAS_FILE = importlib.resources.files("networkx.generators") / "atlas.dat.gz"
13: (0)             def _generate_graphs():
14: (4)                 """Sequentially read the file containing the edge list data for the
15: (4)                 graphs in the atlas and generate the graphs one at a time.
16: (4)                 This function reads the file given in :data:`.ATLAS_FILE`.
17: (4)                 """
18: (4)                 with gzip.open(ATLAS_FILE, "rb") as f:
19: (8)                     line = f.readline()
20: (8)                     while line and line.startswith(b"GRAPH"):
21: (12)                        graph_index = int(line[6:].rstrip())
22: (12)                        line = f.readline()
23: (12)                        num_nodes = int(line[6:].rstrip())
24: (12)                        edgelist = []
```

```
 25: (12)                        line = f.readline()
 26: (12)                        while line and not line.startswith(b"GRAPH"):
 27: (16)                            edgelist.append(line.rstrip())
 28: (16)                            line = f.readline()
 29: (12)                        G = nx.Graph()
 30: (12)                        G.name = f"G{graph_index}"
 31: (12)                        G.add_nodes_from(range(num_nodes))
 32: (12)                        G.add_edges_from(tuple(map(int, e.split())) for e in edgelist)
 33: (12)                        yield G
 34: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 35: (0)             def graph_atlas(i):
 36: (4)                 """Returns graph number `i` from the Graph Atlas.
 37: (4)                 For more information, see :func:`.graph_atlas_g`.
 38: (4)                 Parameters
 39: (4)                 ----------
 40: (4)                 i : int
 41: (8)                     The index of the graph from the atlas to get. The graph at index
 42: (8)                     0 is assumed to be the null graph.
 43: (4)                 Returns
 44: (4)                 -------
 45: (4)                 list
 46: (8)                     A list of :class:`~networkx.Graph` objects, the one at index *i*
 47: (8)                     corresponding to the graph *i* in the Graph Atlas.
 48: (4)                 See also
 49: (4)                 --------
 50: (4)                 graph_atlas_g
 51: (4)                 Notes
 52: (4)                 -----
 53: (4)                 The time required by this function increases linearly with the
 54: (4)                 argument `i`, since it reads a large file sequentially in order to
 55: (4)                 generate the graph [1]_.
 56: (4)                 References
 57: (4)                 ----------
 58: (4)                 .. [1] Ronald C. Read and Robin J. Wilson, *An Atlas of Graphs*.
 59: (11)                       Oxford University Press, 1998.
 60: (4)                 """
 61: (4)                 if not (0 <= i < NUM_GRAPHS):
 62: (8)                     raise ValueError(f"index must be between 0 and {NUM_GRAPHS}")
 63: (4)                 return next(islice(_generate_graphs(), i, None))
 64: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 65: (0)             def graph_atlas_g():
 66: (4)                 """Returns the list of all graphs with up to seven nodes named in the
 67: (4)                 Graph Atlas.
 68: (4)                 The graphs are listed in increasing order by
 69: (4)                 1. number of nodes,
 70: (4)                 2. number of edges,
 71: (4)                 3. degree sequence (for example 111223 < 112222),
 72: (4)                 4. number of automorphisms,
 73: (4)                 in that order, with three exceptions as described in the *Notes*
 74: (4)                 section below. This causes the list to correspond with the index of
 75: (4)                 the graphs in the Graph Atlas [atlas]_, with the first graph,
 76: (4)                 ``G[0]``, being the null graph.
 77: (4)                 Returns
 78: (4)                 -------
 79: (4)                 list
 80: (8)                     A list of :class:`~networkx.Graph` objects, the one at index *i*
 81: (8)                     corresponding to the graph *i* in the Graph Atlas.
 82: (4)                 See also
 83: (4)                 --------
 84: (4)                 graph_atlas
 85: (4)                 Notes
 86: (4)                 -----
 87: (4)                 This function may be expensive in both time and space, since it
 88: (4)                 reads a large file sequentially in order to populate the list.
 89: (4)                 Although the NetworkX atlas functions match the order of graphs
 90: (4)                 given in the "Atlas of Graphs" book, there are (at least) three
 91: (4)                 errors in the ordering described in the book. The following three
 92: (4)                 pairs of nodes violate the lexicographically nondecreasing sorted
 93: (4)                 degree sequence rule:
```

```
94: (4)                 - graphs 55 and 56 with degree sequences 001111 and 000112,
95: (4)                 - graphs 1007 and 1008 with degree sequences 3333444 and 3333336,
96: (4)                 - graphs 1012 and 1213 with degree sequences 1244555 and 1244456.
97: (4)                 References
98: (4)                 ----------
99: (4)                 .. [atlas] Ronald C. Read and Robin J. Wilson,
100: (15)                           *An Atlas of Graphs*.
101: (15)                           Oxford University Press, 1998.
102: (4)                """
103: (4)                return list(_generate_graphs())

----------------------------------------

File 428 - classic.py:

1: (0)              """Generators for some classic graphs.
2: (0)              The typical graph builder function is called as follows:
3: (0)              >>> G = nx.complete_graph(100)
4: (0)              returning the complete graph on n nodes labeled 0, .., 99
5: (0)              as a simple graph. Except for `empty_graph`, all the functions
6: (0)              in this module return a Graph class (i.e. a simple, undirected graph).
7: (0)              """
8: (0)              import itertools
9: (0)              import numbers
10: (0)             import networkx as nx
11: (0)             from networkx.classes import Graph
12: (0)             from networkx.exception import NetworkXError
13: (0)             from networkx.utils import nodes_or_number, pairwise
14: (0)             __all__ = [
15: (4)                 "balanced_tree",
16: (4)                 "barbell_graph",
17: (4)                 "binomial_tree",
18: (4)                 "complete_graph",
19: (4)                 "complete_multipartite_graph",
20: (4)                 "circular_ladder_graph",
21: (4)                 "circulant_graph",
22: (4)                 "cycle_graph",
23: (4)                 "dorogovtsev_goltsev_mendes_graph",
24: (4)                 "empty_graph",
25: (4)                 "full_rary_tree",
26: (4)                 "kneser_graph",
27: (4)                 "ladder_graph",
28: (4)                 "lollipop_graph",
29: (4)                 "null_graph",
30: (4)                 "path_graph",
31: (4)                 "star_graph",
32: (4)                 "tadpole_graph",
33: (4)                 "trivial_graph",
34: (4)                 "turan_graph",
35: (4)                 "wheel_graph",
36: (0)             ]
37: (0)             def _tree_edges(n, r):
38: (4)                 if n == 0:
39: (8)                     return
40: (4)                 nodes = iter(range(n))
41: (4)                 parents = [next(nodes)]  # stack of max length r
42: (4)                 while parents:
43: (8)                     source = parents.pop(0)
44: (8)                     for i in range(r):
45: (12)                        try:
46: (16)                            target = next(nodes)
47: (16)                            parents.append(target)
48: (16)                            yield source, target
49: (12)                        except StopIteration:
50: (16)                            break
51: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
52: (0)             def full_rary_tree(r, n, create_using=None):
53: (4)                 """Creates a full r-ary tree of `n` nodes.
54: (4)                 Sometimes called a k-ary, n-ary, or m-ary tree.
```

```
    "... all non-leaf nodes have exactly r children and all levels
    are full except for some rightmost position of the bottom level
    (if a leaf at the bottom level is missing, then so are all of the
    leaves to its right." [1]_
    .. plot::
        >>> nx.draw(nx.full_rary_tree(2, 10))
    Parameters
    ----------
    r : int
        branching factor of the tree
    n : int
        Number of nodes in the tree
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : networkx Graph
        An r-ary tree with n nodes
    References
    ----------
    .. [1] An introduction to data structures and algorithms,
           James Andrew Storer,  Birkhauser Boston 2001, (page 225).
    """
    G = empty_graph(n, create_using)
    G.add_edges_from(_tree_edges(n, r))
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def kneser_graph(n, k):
    """Returns the Kneser Graph with parameters `n` and `k`.
    The Kneser Graph has nodes that are k-tuples (subsets) of the integers
    between 0 and ``n-1``. Nodes are adjacent if their corresponding sets are disjoint.
    Parameters
    ----------
    n: int
        Number of integers from which to make node subsets.
        Subsets are drawn from ``set(range(n))``.
    k: int
        Size of the subsets.
    Returns
    -------
    G : NetworkX Graph
    Examples
    --------
    >>> G = nx.kneser_graph(5, 2)
    >>> G.number_of_nodes()
    10
    >>> G.number_of_edges()
    15
    >>> nx.is_isomorphic(G, nx.petersen_graph())
    True
    """
    if n <= 0:
        raise NetworkXError("n should be greater than zero")
    if k <= 0 or k > n:
        raise NetworkXError("k should be greater than zero and smaller than n")
    G = nx.Graph()
    subsets = list(itertools.combinations(range(n), k))
    if 2 * k > n:
        G.add_nodes_from(subsets)
    universe = set(range(n))
    comb = itertools.combinations  # only to make it all fit on one line
    G.add_edges_from((s, t) for s in subsets for t in comb(universe - set(s), k))
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def balanced_tree(r, h, create_using=None):
    """Returns the perfectly balanced `r`-ary tree of height `h`.
```

```
(4)                 .. plot::
(8)                     >>> nx.draw(nx.balanced_tree(2, 3))
(4)                 Parameters
(4)                 ----------
(4)                 r : int
(8)                     Branching factor of the tree; each node will have `r`
(8)                     children.
(4)                 h : int
(8)                     Height of the tree.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX graph
(8)                     A balanced `r`-ary tree of height `h`.
(4)                 Notes
(4)                 -----
(4)                 This is the rooted tree where all leaves are at distance `h` from
(4)                 the root. The root has degree `r` and all other internal nodes
(4)                 have degree `r + 1`.
(4)                 Node labels are integers, starting from zero.
(4)                 A balanced tree is also known as a *complete r-ary tree*.
(4)                 """
(4)                 if r == 1:
(8)                     n = h + 1
(4)                 else:
(8)                     n = (1 - r ** (h + 1)) // (1 - r)
(4)                 return full_rary_tree(r, n, create_using=create_using)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def barbell_graph(m1, m2, create_using=None):
(4)                 """Returns the Barbell Graph: two complete graphs connected by a path.
(4)                 .. plot::
(8)                     >>> nx.draw(nx.barbell_graph(4, 2))
(4)                 Parameters
(4)                 ----------
(4)                 m1 : int
(8)                     Size of the left and right barbells, must be greater than 2.
(4)                 m2 : int
(8)                     Length of the path connecting the barbells.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(7)                    Only undirected Graphs are supported.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX graph
(8)                     A barbell graph.
(4)                 Notes
(4)                 -----
(4)                 Two identical complete graphs $K_{m1}$ form the left and right bells,
(4)                 and are connected by a path $P_{m2}$.
(4)                 The `2*m1+m2`  nodes are numbered
(8)                     `0, ..., m1-1` for the left barbell,
(8)                     `m1, ..., m1+m2-1` for the path,
(8)                     and `m1+m2, ..., 2*m1+m2-1` for the right barbell.
(4)                 The 3 subgraphs are joined via the edges `(m1-1, m1)` and
(4)                 `(m1+m2-1, m1+m2)`. If `m2=0`, this is merely two complete
(4)                 graphs joined together.
(4)                 This graph is an extremal example in David Aldous
(4)                 and Jim Fill's e-text on Random Walks on Graphs.
(4)                 """
(4)                 if m1 < 2:
(8)                     raise NetworkXError("Invalid graph description, m1 should be >=2")
(4)                 if m2 < 0:
(8)                     raise NetworkXError("Invalid graph description, m2 should be >=0")
(4)                 G = complete_graph(m1, create_using)
(4)                 if G.is_directed():
(8)                     raise NetworkXError("Directed Graph not supported")
(4)                 G.add_nodes_from(range(m1, m1 + m2 - 1))
(4)                 if m2 > 1:
```

```
(8)                 G.add_edges_from(pairwise(range(m1, m1 + m2)))
(4)             G.add_edges_from(
(8)                 (u, v) for u in range(m1 + m2, 2 * m1 + m2) for v in range(u + 1, 2 * m1 + m2)
(4)             )
(4)             G.add_edge(m1 - 1, m1)
(4)             if m2 > 0:
(8)                 G.add_edge(m1 + m2 - 1, m1 + m2)
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def binomial_tree(n, create_using=None):
(4)             """Returns the Binomial Tree of order n.
(4)             The binomial tree of order 0 consists of a single node. A binomial tree of order k
(4)             is defined recursively by linking two binomial trees of order k-1: the root of one is
(4)             the leftmost child of the root of the other.
(4)             .. plot::
(8)                 >>> nx.draw(nx.binomial_tree(3))
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 Order of the binomial tree.
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : NetworkX graph
(8)                 A binomial tree of $2^n$ nodes and $2^n - 1$ edges.
(4)             """
(4)             G = nx.empty_graph(1, create_using)
(4)             N = 1
(4)             for i in range(n):
(8)                 edges = [(u + N, v + N) for (u, v) in G.edges()]
(8)                 G.add_edges_from(edges)
(8)                 G.add_edge(0, N)
(8)                 N *= 2
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         @nodes_or_number(0)
(0)         def complete_graph(n, create_using=None):
(4)             """Return the complete graph `K_n` with n nodes.
(4)             A complete graph on `n` nodes means that all pairs
(4)             of distinct nodes have an edge connecting them.
(4)             .. plot::
(8)                 >>> nx.draw(nx.complete_graph(5))
(4)             Parameters
(4)             ----------
(4)             n : int or iterable container of nodes
(8)                 If n is an integer, nodes are from range(n).
(8)                 If n is a container of nodes, those nodes appear in the graph.
(8)                 Warning: n is not checked for duplicates and if present the
(8)                 resulting graph may not be as desired. Make sure you have no duplicates.
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                Graph type to create. If graph instance, then cleared before populated.
(4)             Examples
(4)             --------
(4)             >>> G = nx.complete_graph(9)
(4)             >>> len(G)
(4)             9
(4)             >>> G.size()
(4)             36
(4)             >>> G = nx.complete_graph(range(11, 14))
(4)             >>> list(G.nodes())
(4)             [11, 12, 13]
(4)             >>> G = nx.complete_graph(4, nx.DiGraph())
(4)             >>> G.is_directed()
(4)             True
```

```
(4)                 """
(4)                 _, nodes = n
(4)                 G = empty_graph(nodes, create_using)
(4)                 if len(nodes) > 1:
(8)                     if G.is_directed():
(12)                        edges = itertools.permutations(nodes, 2)
(8)                     else:
(12)                        edges = itertools.combinations(nodes, 2)
(8)                     G.add_edges_from(edges)
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def circular_ladder_graph(n, create_using=None):
(4)                 """Returns the circular ladder graph $CL_n$ of length n.
(4)                 $CL_n$ consists of two concentric n-cycles in which
(4)                 each of the n pairs of concentric nodes are joined by an edge.
(4)                 Node labels are the integers 0 to n-1
(4)                 .. plot::
(8)                     >>> nx.draw(nx.circular_ladder_graph(5))
(4)                 """
(4)                 G = ladder_graph(n, create_using)
(4)                 G.add_edge(0, n - 1)
(4)                 G.add_edge(n, 2 * n - 1)
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def circulant_graph(n, offsets, create_using=None):
(4)                 r"""Returns the circulant graph $Ci_n(x_1, x_2, ..., x_m)$ with $n$ nodes.
(4)                 The circulant graph $Ci_n(x_1, ..., x_m)$ consists of $n$ nodes $0, ...,
n-1$
(4)                 such that node $i$ is connected to nodes $(i + x) \mod n$ and $(i - x)
\mod n$
(4)                 for all $x$ in $x_1, ..., x_m$. Thus $Ci_n(1)$ is a cycle graph.
(4)                 .. plot::
(8)                     >>> nx.draw(nx.circulant_graph(10, [1]))
(4)                 Parameters
(4)                 ----------
(4)                 n : integer
(8)                     The number of nodes in the graph.
(4)                 offsets : list of integers
(8)                     A list of node offsets, $x_1$ up to $x_m$, as described above.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 NetworkX Graph of type create_using
(4)                 Examples
(4)                 --------
(4)                 Many well-known graph families are subfamilies of the circulant graphs;
(4)                 for example, to create the cycle graph on n points, we connect every
(4)                 node to nodes on either side (with offset plus or minus one). For n = 10,
(4)                 >>> G = nx.circulant_graph(10, [1])
(4)                 >>> edges = [
(4)                 ...     (0, 9),
(4)                 ...     (0, 1),
(4)                 ...     (1, 2),
(4)                 ...     (2, 3),
(4)                 ...     (3, 4),
(4)                 ...     (4, 5),
(4)                 ...     (5, 6),
(4)                 ...     (6, 7),
(4)                 ...     (7, 8),
(4)                 ...     (8, 9),
(4)                 ... ]
(4)                 >>> sorted(edges) == sorted(G.edges())
(4)                 True
(4)                 Similarly, we can create the complete graph
(4)                 on 5 points with the set of offsets [1, 2]:
(4)                 >>> G = nx.circulant_graph(5, [1, 2])
(4)                 >>> edges = [
(4)                 ...     (0, 1),
```

```
(4)                 ...     (0, 2),
(4)                 ...     (0, 3),
(4)                 ...     (0, 4),
(4)                 ...     (1, 2),
(4)                 ...     (1, 3),
(4)                 ...     (1, 4),
(4)                 ...     (2, 3),
(4)                 ...     (2, 4),
(4)                 ...     (3, 4),
(4)                 ... ]
(4)                 >>> sorted(edges) == sorted(G.edges())
(4)                 True
(4)                 """
(4)                 G = empty_graph(n, create_using)
(4)                 for i in range(n):
(8)                     for j in offsets:
(12)                        G.add_edge(i, (i - j) % n)
(12)                        G.add_edge(i, (i + j) % n)
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             @nodes_or_number(0)
(0)             def cycle_graph(n, create_using=None):
(4)                 """Returns the cycle graph $C_n$ of cyclically connected nodes.
(4)                 $C_n$ is a path with its two end-nodes connected.
(4)                 .. plot::
(8)                     >>> nx.draw(nx.cycle_graph(5))
(4)                 Parameters
(4)                 ----------
(4)                 n : int or iterable container of nodes
(8)                     If n is an integer, nodes are from `range(n)`.
(8)                     If n is a container of nodes, those nodes appear in the graph.
(8)                     Warning: n is not checked for duplicates and if present the
(8)                     resulting graph may not be as desired. Make sure you have no
duplicates.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Notes
(4)                 -----
(4)                 If create_using is directed, the direction is in increasing order.
(4)                 """
(4)                 _, nodes = n
(4)                 G = empty_graph(nodes, create_using)
(4)                 G.add_edges_from(pairwise(nodes, cyclic=True))
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def dorogovtsev_goltsev_mendes_graph(n, create_using=None):
(4)                 """Returns the hierarchically constructed Dorogovtsev-Goltsev-Mendes
graph.
(4)                 The Dorogovtsev-Goltsev-Mendes [1]_ procedure produces a scale-free graph
(4)                 deterministically with the following properties for a given `n`:
(4)                 - Total number of nodes = ``3 * (3**n + 1) / 2``
(4)                 - Total number of edges = ``3 ** (n + 1)``
(4)                 .. plot::
(8)                     >>> nx.draw(nx.dorogovtsev_goltsev_mendes_graph(3))
(4)                 Parameters
(4)                 ----------
(4)                 n : integer
(7)                    The generation number.
(4)                 create_using : NetworkX Graph, optional
(7)                    Graph type to be returned. Directed graphs and multi graphs are not
(7)                    supported.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX Graph
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.dorogovtsev_goltsev_mendes_graph(3)
(4)                 >>> G.number_of_nodes()
(4)                 15
```

```
(4)                 >>> G.number_of_edges()
(4)                 27
(4)                 >>> nx.is_planar(G)
(4)                 True
(4)                 References
(4)                 ----------
(4)                 .. [1] S. N. Dorogovtsev, A. V. Goltsev and J. F. F. Mendes,
(8)                     "Pseudofractal scale-free web", Physical Review E 65, 066122, 2002.
(8)                     https://arxiv.org/pdf/cond-mat/0112143.pdf
(4)                 """
(4)                 G = empty_graph(0, create_using)
(4)                 if G.is_directed():
(8)                     raise NetworkXError("Directed Graph not supported")
(4)                 if G.is_multigraph():
(8)                     raise NetworkXError("Multigraph not supported")
(4)                 G.add_edge(0, 1)
(4)                 if n == 0:
(8)                     return G
(4)                 new_node = 2  # next node to be added
(4)                 for i in range(1, n + 1):  # iterate over number of generations.
(8)                     last_generation_edges = list(G.edges())
(8)                     number_of_edges_in_last_generation = len(last_generation_edges)
(8)                     for j in range(number_of_edges_in_last_generation):
(12)                        G.add_edge(new_node, last_generation_edges[j][0])
(12)                        G.add_edge(new_node, last_generation_edges[j][1])
(12)                        new_node += 1
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             @nodes_or_number(0)
(0)             def empty_graph(n=0, create_using=None, default=Graph):
(4)                 """Returns the empty graph with n nodes and zero edges.
(4)                 .. plot::
(8)                     >>> nx.draw(nx.empty_graph(5))
(4)                 Parameters
(4)                 ----------
(4)                 n : int or iterable container of nodes (default = 0)
(8)                     If n is an integer, nodes are from `range(n)`.
(8)                     If n is a container of nodes, those nodes appear in the graph.
(4)                 create_using : Graph Instance, Constructor or None
(8)                     Indicator of type of graph to return.
(8)                     If a Graph-type instance, then clear and use it.
(8)                     If None, use the `default` constructor.
(8)                     If a constructor, call it to create an empty graph.
(4)                 default : Graph constructor (optional, default = nx.Graph)
(8)                     The constructor to use if create_using is None.
(8)                     If None, then nx.Graph is used.
(8)                     This is used when passing an unknown `create_using` value
(8)                     through your home-grown function to `empty_graph` and
(8)                     you want a default constructor other than nx.Graph.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.empty_graph(10)
(4)                 >>> G.number_of_nodes()
(4)                 10
(4)                 >>> G.number_of_edges()
(4)                 0
(4)                 >>> G = nx.empty_graph("ABC")
(4)                 >>> G.number_of_nodes()
(4)                 3
(4)                 >>> sorted(G)
(4)                 ['A', 'B', 'C']
(4)                 Notes
(4)                 -----
(4)                 The variable create_using should be a Graph Constructor or a
(4)                 "graph"-like object. Constructors, e.g. `nx.Graph` or `nx.MultiGraph`
(4)                 will be used to create the returned graph. "graph"-like objects
(4)                 will be cleared (nodes and edges will be removed) and refitted as
(4)                 an empty "graph" with nodes specified in n. This capability
(4)                 is useful for specifying the class-nature of the resulting empty
```

```
458: (4)                "graph" (i.e. Graph, DiGraph, MyWeirdGraphClass, etc.).
459: (4)                The variable create_using has three main uses:
460: (4)                Firstly, the variable create_using can be used to create an
461: (4)                empty digraph, multigraph, etc.  For example,
462: (4)                >>> n = 10
463: (4)                >>> G = nx.empty_graph(n, create_using=nx.DiGraph)
464: (4)                will create an empty digraph on n nodes.
465: (4)                Secondly, one can pass an existing graph (digraph, multigraph,
466: (4)                etc.) via create_using. For example, if G is an existing graph
467: (4)                (resp. digraph, multigraph, etc.), then empty_graph(n, create_using=G)
468: (4)                will empty G (i.e. delete all nodes and edges using G.clear())
469: (4)                and then add n nodes and zero edges, and return the modified graph.
470: (4)                Thirdly, when constructing your home-grown graph creation function
471: (4)                you can use empty_graph to construct the graph by passing a user
472: (4)                defined create_using to empty_graph. In this case, if you want the
473: (4)                default constructor to be other than nx.Graph, specify `default`.
474: (4)                >>> def mygraph(n, create_using=None):
475: (4)                ...     G = nx.empty_graph(n, create_using, nx.MultiGraph)
476: (4)                ...     G.add_edges_from([(0, 1), (0, 1)])
477: (4)                ...     return G
478: (4)                >>> G = mygraph(3)
479: (4)                >>> G.is_multigraph()
480: (4)                True
481: (4)                >>> G = mygraph(3, nx.Graph)
482: (4)                >>> G.is_multigraph()
483: (4)                False
484: (4)                See also create_empty_copy(G).
485: (4)                """
486: (4)                if create_using is None:
487: (8)                    G = default()
488: (4)                elif isinstance(create_using, type):
489: (8)                    G = create_using()
490: (4)                elif not hasattr(create_using, "adj"):
491: (8)                    raise TypeError("create_using is not a valid NetworkX graph type or
instance")
492: (4)                else:
493: (8)                    create_using.clear()
494: (8)                    G = create_using
495: (4)                _, nodes = n
496: (4)                G.add_nodes_from(nodes)
497: (4)                return G
498: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
499: (0)            def ladder_graph(n, create_using=None):
500: (4)                """Returns the Ladder graph of length n.
501: (4)                This is two paths of n nodes, with
502: (4)                each pair connected by a single edge.
503: (4)                Node labels are the integers 0 to 2*n - 1.
504: (4)                .. plot::
505: (8)                    >>> nx.draw(nx.ladder_graph(5))
506: (4)                """
507: (4)                G = empty_graph(2 * n, create_using)
508: (4)                if G.is_directed():
509: (8)                    raise NetworkXError("Directed Graph not supported")
510: (4)                G.add_edges_from(pairwise(range(n)))
511: (4)                G.add_edges_from(pairwise(range(n, 2 * n)))
512: (4)                G.add_edges_from((v, v + n) for v in range(n))
513: (4)                return G
514: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
515: (0)            @nodes_or_number([0, 1])
516: (0)            def lollipop_graph(m, n, create_using=None):
517: (4)                """Returns the Lollipop Graph; ``K_m`` connected to ``P_n``.
518: (4)                This is the Barbell Graph without the right barbell.
519: (4)                .. plot::
520: (8)                    >>> nx.draw(nx.lollipop_graph(3, 4))
521: (4)                Parameters
522: (4)                ----------
523: (4)                m, n : int or iterable container of nodes
524: (8)                    If an integer, nodes are from ``range(m)`` and ``range(m, m+n)``.
525: (8)                    If a container of nodes, those nodes appear in the graph.
```

```
(8)                 Warning: `m` and `n` are not checked for duplicates and if present the
(8)                 resulting graph may not be as desired. Make sure you have no duplicates.
(8)                 The nodes for `m` appear in the complete graph $K_m$ and the nodes
(8)                 for `n` appear in the path $P_n$
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             Networkx graph
(7)                A complete graph with `m` nodes connected to a path of length `n`.
(4)             Notes
(4)             -----
(4)             The 2 subgraphs are joined via an edge ``(m-1, m)``.
(4)             If ``n=0``, this is merely a complete graph.
(4)             (This graph is an extremal example in David Aldous and Jim
(4)             Fill's etext on Random Walks on Graphs.)
(4)             """
(4)             m, m_nodes = m
(4)             M = len(m_nodes)
(4)             if M < 2:
(8)                 raise NetworkXError("Invalid description: m should indicate at least 2 nodes")
(4)             n, n_nodes = n
(4)             if isinstance(m, numbers.Integral) and isinstance(n, numbers.Integral):
(8)                 n_nodes = list(range(M, M + n))
(4)             N = len(n_nodes)
(4)             G = complete_graph(m_nodes, create_using)
(4)             if G.is_directed():
(8)                 raise NetworkXError("Directed Graph not supported")
(4)             G.add_nodes_from(n_nodes)
(4)             if N > 1:
(8)                 G.add_edges_from(pairwise(n_nodes))
(4)             if len(G) != M + N:
(8)                 raise NetworkXError("Nodes must be distinct in containers m and n")
(4)             if M > 0 and N > 0:
(8)                 G.add_edge(m_nodes[-1], n_nodes[0])
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def null_graph(create_using=None):
(4)             """Returns the Null graph with no nodes or edges.
(4)             See empty_graph for the use of create_using.
(4)             """
(4)             G = empty_graph(0, create_using)
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         @nodes_or_number(0)
(0)         def path_graph(n, create_using=None):
(4)             """Returns the Path graph `P_n` of linearly connected nodes.
(4)             .. plot::
(8)                 >>> nx.draw(nx.path_graph(5))
(4)             Parameters
(4)             ----------
(4)             n : int or iterable
(8)                 If an integer, nodes are 0 to n - 1.
(8)                 If an iterable of nodes, in the order they appear in the path.
(8)                 Warning: n is not checked for duplicates and if present the
(8)                 resulting graph may not be as desired. Make sure you have no duplicates.
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                Graph type to create. If graph instance, then cleared before populated.
(4)             """
(4)             _, nodes = n
(4)             G = empty_graph(nodes, create_using)
(4)             G.add_edges_from(pairwise(nodes))
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         @nodes_or_number(0)
(0)         def star_graph(n, create_using=None):
```

```
    """Return the star graph
    The star graph consists of one center node connected to n outer nodes.
    .. plot::
        >>> nx.draw(nx.star_graph(6))
    Parameters
    ----------
    n : int or iterable
        If an integer, node labels are 0 to n with center 0.
        If an iterable of nodes, the center is the first.
        Warning: n is not checked for duplicates and if present the
        resulting graph may not be as desired. Make sure you have no duplicates.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Notes
    -----
    The graph has n+1 nodes for integer n.
    So star_graph(3) is the same as star_graph(range(4)).
    """
    n, nodes = n
    if isinstance(n, numbers.Integral):
        nodes.append(int(n))  # there should be n+1 nodes
    G = empty_graph(nodes, create_using)
    if G.is_directed():
        raise NetworkXError("Directed Graph not supported")
    if len(nodes) > 1:
        hub, *spokes = nodes
        G.add_edges_from((hub, node) for node in spokes)
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
@nodes_or_number([0, 1])
def tadpole_graph(m, n, create_using=None):
    """Returns the (m,n)-tadpole graph; ``C_m`` connected to ``P_n``.
    This graph on m+n nodes connects a cycle of size `m` to a path of length `n`.
    It looks like a tadpole. It is also called a kite graph or a dragon graph.
    .. plot::
        >>> nx.draw(nx.tadpole_graph(3, 5))
    Parameters
    ----------
    m, n : int or iterable container of nodes
        If an integer, nodes are from ``range(m)`` and ``range(m,m+n)``.
        If a container of nodes, those nodes appear in the graph.
        Warning: `m` and `n` are not checked for duplicates and if present the
        resulting graph may not be as desired.
        The nodes for `m` appear in the cycle graph $C_m$ and the nodes
        for `n` appear in the path $P_n$.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    Networkx graph
       A cycle of size `m` connected to a path of length `n`.
    Raises
    ------
    NetworkXError
        If ``m < 2``. The tadpole graph is undefined for ``m<2``.
    Notes
    -----
    The 2 subgraphs are joined via an edge ``(m-1, m)``.
    If ``n=0``, this is a cycle graph.
    `m` and/or `n` can be a container of nodes instead of an integer.
    """
    m, m_nodes = m
    M = len(m_nodes)
    if M < 2:
        raise NetworkXError("Invalid description: m should indicate at least 2 nodes")
    n, n_nodes = n
```

```
    if isinstance(m, numbers.Integral) and isinstance(n, numbers.Integral):
        n_nodes = list(range(M, M + n))
    G = cycle_graph(m_nodes, create_using)
    if G.is_directed():
        raise NetworkXError("Directed Graph not supported")
    nx.add_path(G, [m_nodes[-1]] + list(n_nodes))
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def trivial_graph(create_using=None):
    """Return the Trivial graph with one node (with label 0) and no edges.
    .. plot::
        >>> nx.draw(nx.trivial_graph(), with_labels=True)
    """
    G = empty_graph(1, create_using)
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def turan_graph(n, r):
    r"""Return the Turan Graph
    The Turan Graph is a complete multipartite graph on $n$ nodes
    with $r$ disjoint subsets. That is, edges connect each node to
    every node not in its subset.
    Given $n$ and $r$, we create a complete multipartite graph with
    $r-(n \mod r)$ partitions of size $n/r$, rounded down, and
    $n \mod r$ partitions of size $n/r+1$, rounded down.
    .. plot::
        >>> nx.draw(nx.turan_graph(6, 2))
    Parameters
    ----------
    n : int
        The number of nodes.
    r : int
        The number of partitions.
        Must be less than or equal to n.
    Notes
    -----
    Must satisfy $1 <= r <= n$.
    The graph has $(r-1)(n^2)/(2r)$ edges, rounded down.
    """
    if not 1 <= r <= n:
        raise NetworkXError("Must satisfy 1 <= r <= n")
    partitions = [n // r] * (r - (n % r)) + [n // r + 1] * (n % r)
    G = complete_multipartite_graph(*partitions)
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
@nodes_or_number(0)
def wheel_graph(n, create_using=None):
    """Return the wheel graph
    The wheel graph consists of a hub node connected to a cycle of (n-1) nodes.
    .. plot::
        >>> nx.draw(nx.wheel_graph(5))
    Parameters
    ----------
    n : int or iterable
        If an integer, node labels are 0 to n with center 0.
        If an iterable of nodes, the center is the first.
        Warning: n is not checked for duplicates and if present the
        resulting graph may not be as desired. Make sure you have no duplicates.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Node labels are the integers 0 to n - 1.
    """
    _, nodes = n
    G = empty_graph(nodes, create_using)
    if G.is_directed():
        raise NetworkXError("Directed Graph not supported")
    if len(nodes) > 1:
        hub, *rim = nodes
```

```
(8)                 G.add_edges_from((hub, node) for node in rim)
(8)                 if len(rim) > 1:
(12)                    G.add_edges_from(pairwise(rim, cyclic=True))
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def complete_multipartite_graph(*subset_sizes):
(4)             """Returns the complete multipartite graph with the specified subset
sizes.
(4)             .. plot::
(8)                 >>> nx.draw(nx.complete_multipartite_graph(1, 2, 3))
(4)             Parameters
(4)             ----------
(4)             subset_sizes : tuple of integers or tuple of node iterables
(7)                The arguments can either all be integer number of nodes or they
(7)                can all be iterables of nodes. If integers, they represent the
(7)                number of nodes in each subset of the multipartite graph.
(7)                If iterables, each is used to create the nodes for that subset.
(7)                The length of subset_sizes is the number of subsets.
(4)             Returns
(4)             -------
(4)             G : NetworkX Graph
(7)                Returns the complete multipartite graph with the specified subsets.
(7)                For each node, the node attribute 'subset' is an integer
(7)                indicating which subset contains the node.
(4)             Examples
(4)             --------
(4)             Creating a complete tripartite graph, with subsets of one, two, and three
(4)             nodes, respectively.
(8)                 >>> G = nx.complete_multipartite_graph(1, 2, 3)
(8)                 >>> [G.nodes[u]["subset"] for u in G]
(8)                 [0, 1, 1, 2, 2, 2]
(8)                 >>> list(G.edges(0))
(8)                 [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]
(8)                 >>> list(G.edges(2))
(8)                 [(2, 0), (2, 3), (2, 4), (2, 5)]
(8)                 >>> list(G.edges(4))
(8)                 [(4, 0), (4, 1), (4, 2)]
(8)                 >>> G = nx.complete_multipartite_graph("a", "bc", "def")
(8)                 >>> [G.nodes[u]["subset"] for u in sorted(G)]
(8)                 [0, 1, 1, 2, 2, 2]
(4)             Notes
(4)             -----
(4)             This function generalizes several other graph builder functions.
(4)             - If no subset sizes are given, this returns the null graph.
(4)             - If a single subset size `n` is given, this returns the empty graph on
(6)               `n` nodes.
(4)             - If two subset sizes `m` and `n` are given, this returns the complete
(6)               bipartite graph on `m + n` nodes.
(4)             - If subset sizes `1` and `n` are given, this returns the star graph on
(6)               `n + 1` nodes.
(4)             See also
(4)             --------
(4)             complete_bipartite_graph
(4)             """
(4)             G = Graph()
(4)             if len(subset_sizes) == 0:
(8)                 return G
(4)             try:
(8)                 extents = pairwise(itertools.accumulate((0,) + subset_sizes))
(8)                 subsets = [range(start, end) for start, end in extents]
(4)             except TypeError:
(8)                 subsets = subset_sizes
(4)             else:
(8)                 if any(size < 0 for size in subset_sizes):
(12)                    raise NetworkXError(f"Negative number of nodes not valid:
{subset_sizes}")
(4)             try:
(8)                 for i, subset in enumerate(subsets):
(12)                    G.add_nodes_from(subset, subset=i)
```

```
792: (4)                except TypeError as err:
793: (8)                    raise NetworkXError("Arguments must be all ints or all iterables")
from err
794: (4)                for subset1, subset2 in itertools.combinations(subsets, 2):
795: (8)                    G.add_edges_from(itertools.product(subset1, subset2))
796: (4)                return G
```

File 429 - cographs.py:

```
1: (0)              r"""Generators for cographs
2: (0)              A cograph is a graph containing no path on four vertices.
3: (0)              Cographs or $P_4$-free graphs can be obtained from a single vertex
4: (0)              by disjoint union and complementation operations.
5: (0)              References
6: (0)              ----------
7: (0)              .. [0] D.G. Corneil, H. Lerchs, L.Stewart Burlingham,
8: (4)                  "Complement reducible graphs",
9: (4)                  Discrete Applied Mathematics, Volume 3, Issue 3, 1981, Pages 163-174,
10: (4)                 ISSN 0166-218X.
11: (0)             """
12: (0)             import networkx as nx
13: (0)             from networkx.utils import py_random_state
14: (0)             __all__ = ["random_cograph"]
15: (0)             @py_random_state(1)
16: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
17: (0)             def random_cograph(n, seed=None):
18: (4)                 r"""Returns a random cograph with $2 ^ n$ nodes.
19: (4)                 A cograph is a graph containing no path on four vertices.
20: (4)                 Cographs or $P_4$-free graphs can be obtained from a single vertex
21: (4)                 by disjoint union and complementation operations.
22: (4)                 This generator starts off from a single vertex and performs disjoint
23: (4)                 union and full join operations on itself.
24: (4)                 The decision on which operation will take place is random.
25: (4)                 Parameters
26: (4)                 ----------
27: (4)                 n : int
28: (8)                     The order of the cograph.
29: (4)                 seed : integer, random_state, or None (default)
30: (8)                     Indicator of random number generation state.
31: (8)                     See :ref:`Randomness<randomness>`.
32: (4)                 Returns
33: (4)                 -------
34: (4)                 G : A random graph containing no path on four vertices.
35: (4)                 See Also
36: (4)                 --------
37: (4)                 full_join
38: (4)                 union
39: (4)                 References
40: (4)                 ----------
41: (4)                 .. [1] D.G. Corneil, H. Lerchs, L.Stewart Burlingham,
42: (7)                    "Complement reducible graphs",
43: (7)                    Discrete Applied Mathematics, Volume 3, Issue 3, 1981, Pages 163-174,
44: (7)                    ISSN 0166-218X.
45: (4)                 """
46: (4)                 R = nx.empty_graph(1)
47: (4)                 for i in range(n):
48: (8)                     RR = nx.relabel_nodes(R.copy(), lambda x: x + len(R))
49: (8)                     if seed.randint(0, 1) == 0:
50: (12)                        R = nx.full_join(R, RR)
51: (8)                     else:
52: (12)                        R = nx.disjoint_union(R, RR)
53: (4)                 return R
```

File 430 - community.py:

```
1: (0)              """Generators for classes of graphs used in studying social networks."""
2: (0)              import itertools
3: (0)              import math
4: (0)              import networkx as nx
5: (0)              from networkx.utils import py_random_state
6: (0)              __all__ = [
7: (4)                  "caveman_graph",
8: (4)                  "connected_caveman_graph",
9: (4)                  "relaxed_caveman_graph",
10: (4)                 "random_partition_graph",
11: (4)                 "planted_partition_graph",
12: (4)                 "gaussian_random_partition_graph",
13: (4)                 "ring_of_cliques",
14: (4)                 "windmill_graph",
15: (4)                 "stochastic_block_model",
16: (4)                 "LFR_benchmark_graph",
17: (0)             ]
18: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
19: (0)             def caveman_graph(l, k):
20: (4)                 """Returns a caveman graph of `l` cliques of size `k`.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 l : int
24: (6)                   Number of cliques
25: (4)                 k : int
26: (6)                   Size of cliques
27: (4)                 Returns
28: (4)                 -------
29: (4)                 G : NetworkX Graph
30: (6)                   caveman graph
31: (4)                 Notes
32: (4)                 -----
33: (4)                 This returns an undirected graph, it can be converted to a directed
34: (4)                 graph using :func:`nx.to_directed`, or a multigraph using
35: (4)                 ``nx.MultiGraph(nx.caveman_graph(l, k))``. Only the undirected version is
36: (4)                 described in [1]_ and it is unclear which of the directed
37: (4)                 generalizations is most useful.
38: (4)                 Examples
39: (4)                 --------
40: (4)                 >>> G = nx.caveman_graph(3, 3)
41: (4)                 See also
42: (4)                 --------
43: (4)                 connected_caveman_graph
44: (4)                 References
45: (4)                 ----------
46: (4)                 .. [1] Watts, D. J. 'Networks, Dynamics, and the Small-World Phenomenon.'
47: (7)                    Amer. J. Soc. 105, 493-527, 1999.
48: (4)                 """
49: (4)                 G = nx.empty_graph(l * k)
50: (4)                 if k > 1:
51: (8)                     for start in range(0, l * k, k):
52: (12)                        edges = itertools.combinations(range(start, start + k), 2)
53: (12)                        G.add_edges_from(edges)
54: (4)                 return G
55: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
56: (0)             def connected_caveman_graph(l, k):
57: (4)                 """Returns a connected caveman graph of `l` cliques of size `k`.
58: (4)                 The connected caveman graph is formed by creating `n` cliques of size
59: (4)                 `k`, then a single edge in each clique is rewired to a node in an
60: (4)                 adjacent clique.
61: (4)                 Parameters
62: (4)                 ----------
63: (4)                 l : int
64: (6)                   number of cliques
65: (4)                 k : int
66: (6)                   size of cliques (k at least 2 or NetworkXError is raised)
67: (4)                 Returns
68: (4)                 -------
69: (4)                 G : NetworkX Graph
```

```
 70: (6)                  connected caveman graph
 71: (4)                Raises
 72: (4)                ------
 73: (4)                NetworkXError
 74: (8)                    If the size of cliques `k` is smaller than 2.
 75: (4)                Notes
 76: (4)                -----
 77: (4)                This returns an undirected graph, it can be converted to a directed
 78: (4)                graph using :func:`nx.to_directed`, or a multigraph using
 79: (4)                ``nx.MultiGraph(nx.caveman_graph(l, k))``. Only the undirected version is
 80: (4)                described in [1]_ and it is unclear which of the directed
 81: (4)                generalizations is most useful.
 82: (4)                Examples
 83: (4)                --------
 84: (4)                >>> G = nx.connected_caveman_graph(3, 3)
 85: (4)                References
 86: (4)                ----------
 87: (4)                .. [1] Watts, D. J. 'Networks, Dynamics, and the Small-World Phenomenon.'
 88: (7)                   Amer. J. Soc. 105, 493-527, 1999.
 89: (4)                """
 90: (4)                if k < 2:
 91: (8)                    raise nx.NetworkXError(
 92: (12)                       "The size of cliques in a connected caveman graph must be at least
 2."
 93: (8)                    )
 94: (4)                G = nx.caveman_graph(l, k)
 95: (4)                for start in range(0, l * k, k):
 96: (8)                    G.remove_edge(start, start + 1)
 97: (8)                    G.add_edge(start, (start - 1) % (l * k))
 98: (4)                return G
 99: (0)            @py_random_state(3)
 100: (0)           @nx._dispatchable(graphs=None, returns_graph=True)
 101: (0)           def relaxed_caveman_graph(l, k, p, seed=None):
 102: (4)               """Returns a relaxed caveman graph.
 103: (4)               A relaxed caveman graph starts with `l` cliques of size `k`.  Edges are
 104: (4)               then randomly rewired with probability `p` to link different cliques.
 105: (4)               Parameters
 106: (4)               ----------
 107: (4)               l : int
 108: (6)                 Number of groups
 109: (4)               k : int
 110: (6)                 Size of cliques
 111: (4)               p : float
 112: (6)                 Probability of rewiring each edge.
 113: (4)               seed : integer, random_state, or None (default)
 114: (8)                   Indicator of random number generation state.
 115: (8)                   See :ref:`Randomness<randomness>`.
 116: (4)               Returns
 117: (4)               -------
 118: (4)               G : NetworkX Graph
 119: (6)                 Relaxed Caveman Graph
 120: (4)               Raises
 121: (4)               ------
 122: (4)               NetworkXError
 123: (5)                If p is not in [0,1]
 124: (4)               Examples
 125: (4)               --------
 126: (4)               >>> G = nx.relaxed_caveman_graph(2, 3, 0.1, seed=42)
 127: (4)               References
 128: (4)               ----------
 129: (4)               .. [1] Santo Fortunato, Community Detection in Graphs,
 130: (7)                  Physics Reports Volume 486, Issues 3-5, February 2010, Pages 75-174.
 131: (7)                  https://arxiv.org/abs/0906.0612
 132: (4)               """
 133: (4)               G = nx.caveman_graph(l, k)
 134: (4)               nodes = list(G)
 135: (4)               for u, v in G.edges():
 136: (8)                   if seed.random() < p:  # rewire the edge
 137: (12)                      x = seed.choice(nodes)
```

```
(12)                        if G.has_edge(u, x):
(16)                            continue
(12)                        G.remove_edge(u, v)
(12)                        G.add_edge(u, x)
(4)                 return G
(0)             @py_random_state(3)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_partition_graph(sizes, p_in, p_out, seed=None, directed=False):
(4)                 """Returns the random partition graph with a partition of sizes.
(4)                 A partition graph is a graph of communities with sizes defined by
(4)                 s in sizes. Nodes in the same group are connected with probability
(4)                 p_in and nodes of different groups are connected with probability
(4)                 p_out.
(4)                 Parameters
(4)                 ----------
(4)                 sizes : list of ints
(6)                   Sizes of groups
(4)                 p_in : float
(6)                   probability of edges with in groups
(4)                 p_out : float
(6)                   probability of edges between groups
(4)                 directed : boolean optional, default=False
(6)                   Whether to create a directed graph
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX Graph or DiGraph
(6)                   random partition graph of size sum(gs)
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(6)                   If p_in or p_out is not in [0,1]
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.random_partition_graph([10, 10, 10], 0.25, 0.01)
(4)                 >>> len(G)
(4)                 30
(4)                 >>> partition = G.graph["partition"]
(4)                 >>> len(partition)
(4)                 3
(4)                 Notes
(4)                 -----
(4)                 This is a generalization of the planted-l-partition described in
(4)                 [1]_.  It allows for the creation of groups of any size.
(4)                 The partition is store as a graph attribute 'partition'.
(4)                 References
(4)                 ----------
(4)                 .. [1] Santo Fortunato 'Community Detection in Graphs' Physical Reports
(7)                    Volume 486, Issue 3-5 p. 75-174. https://arxiv.org/abs/0906.0612
(4)                 """
(4)                 if not 0.0 <= p_in <= 1.0:
(8)                     raise nx.NetworkXError("p_in must be in [0,1]")
(4)                 if not 0.0 <= p_out <= 1.0:
(8)                     raise nx.NetworkXError("p_out must be in [0,1]")
(4)                 num_blocks = len(sizes)
(4)                 p = [[p_out for s in range(num_blocks)] for r in range(num_blocks)]
(4)                 for r in range(num_blocks):
(8)                     p[r][r] = p_in
(4)                 return stochastic_block_model(
(8)                     sizes,
(8)                     p,
(8)                     nodelist=None,
(8)                     seed=seed,
(8)                     directed=directed,
(8)                     selfloops=False,
(8)                     sparse=True,
(4)                 )
```

```
(0)            @py_random_state(4)
(0)            @nx._dispatchable(graphs=None, returns_graph=True)
(0)            def planted_partition_graph(l, k, p_in, p_out, seed=None, directed=False):
(4)                """Returns the planted l-partition graph.
(4)                This model partitions a graph with n=l*k vertices in
(4)                l groups with k vertices each. Vertices of the same
(4)                group are linked with a probability p_in, and vertices
(4)                of different groups are linked with probability p_out.
(4)                Parameters
(4)                ----------
(4)                l : int
(6)                  Number of groups
(4)                k : int
(6)                  Number of vertices in each group
(4)                p_in : float
(6)                  probability of connecting vertices within a group
(4)                p_out : float
(6)                  probability of connected vertices between groups
(4)                seed : integer, random_state, or None (default)
(8)                    Indicator of random number generation state.
(8)                    See :ref:`Randomness<randomness>`.
(4)                directed : bool,optional (default=False)
(6)                  If True return a directed graph
(4)                Returns
(4)                -------
(4)                G : NetworkX Graph or DiGraph
(6)                  planted l-partition graph
(4)                Raises
(4)                ------
(4)                NetworkXError
(6)                  If `p_in`, `p_out` are not in `[0, 1]`
(4)                Examples
(4)                --------
(4)                >>> G = nx.planted_partition_graph(4, 3, 0.5, 0.1, seed=42)
(4)                See Also
(4)                --------
(4)                random_partition_model
(4)                References
(4)                ----------
(4)                .. [1] A. Condon, R.M. Karp, Algorithms for graph partitioning
(8)                    on the planted partition model,
(8)                    Random Struct. Algor. 18 (2001) 116-140.
(4)                .. [2] Santo Fortunato 'Community Detection in Graphs' Physical Reports
(7)                   Volume 486, Issue 3-5 p. 75-174. https://arxiv.org/abs/0906.0612
(4)                """
(4)                return random_partition_graph([k] * l, p_in, p_out, seed=seed,
directed=directed)
(0)            @py_random_state(6)
(0)            @nx._dispatchable(graphs=None, returns_graph=True)
(0)            def gaussian_random_partition_graph(n, s, v, p_in, p_out, directed=False,
seed=None):
(4)                """Generate a Gaussian random partition graph.
(4)                A Gaussian random partition graph is created by creating k partitions
(4)                each with a size drawn from a normal distribution with mean s and variance
(4)                s/v. Nodes are connected within clusters with probability p_in and
(4)                between clusters with probability p_out[1]
(4)                Parameters
(4)                ----------
(4)                n : int
(6)                  Number of nodes in the graph
(4)                s : float
(6)                  Mean cluster size
(4)                v : float
(6)                  Shape parameter. The variance of cluster size distribution is s/v.
(4)                p_in : float
(6)                  Probability of intra cluster connection.
(4)                p_out : float
(6)                  Probability of inter cluster connection.
(4)                directed : boolean, optional default=False
```

```
(6)                 Whether to create a directed graph or not
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             G : NetworkX Graph or DiGraph
(6)               gaussian random partition graph
(4)             Raises
(4)             ------
(4)             NetworkXError
(6)               If s is > n
(6)               If p_in or p_out is not in [0,1]
(4)             Notes
(4)             -----
(4)             Note the number of partitions is dependent on s,v and n, and that the
(4)             last partition may be considerably smaller, as it is sized to simply
(4)             fill out the nodes [1]
(4)             See Also
(4)             --------
(4)             random_partition_graph
(4)             Examples
(4)             --------
(4)             >>> G = nx.gaussian_random_partition_graph(100, 10, 10, 0.25, 0.1)
(4)             >>> len(G)
(4)             100
(4)             References
(4)             ----------
(4)             .. [1] Ulrik Brandes, Marco Gaertler, Dorothea Wagner,
(7)                Experiments on Graph Clustering Algorithms,
(7)                In the proceedings of the 11th Europ. Symp. Algorithms, 2003.
(4)             """
(4)             if s > n:
(8)                 raise nx.NetworkXError("s must be <= n")
(4)             assigned = 0
(4)             sizes = []
(4)             while True:
(8)                 size = int(seed.gauss(s, s / v + 0.5))
(8)                 if size < 1:  # how to handle 0 or negative sizes?
(12)                    continue
(8)                 if assigned + size >= n:
(12)                    sizes.append(n - assigned)
(12)                    break
(8)                 assigned += size
(8)                 sizes.append(size)
(4)             return random_partition_graph(sizes, p_in, p_out, seed=seed,
directed=directed)
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def ring_of_cliques(num_cliques, clique_size):
(4)             """Defines a "ring of cliques" graph.
(4)             A ring of cliques graph is consisting of cliques, connected through single
(4)             links. Each clique is a complete graph.
(4)             Parameters
(4)             ----------
(4)             num_cliques : int
(8)                 Number of cliques
(4)             clique_size : int
(8)                 Size of cliques
(4)             Returns
(4)             -------
(4)             G : NetworkX Graph
(8)                 ring of cliques graph
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If the number of cliques is lower than 2 or
(8)                 if the size of cliques is smaller than 2.
(4)             Examples
(4)             --------
```

```
342: (4)                >>> G = nx.ring_of_cliques(8, 4)
343: (4)                See Also
344: (4)                --------
345: (4)                connected_caveman_graph
346: (4)                Notes
347: (4)                -----
348: (4)                The `connected_caveman_graph` graph removes a link from each clique to
349: (4)                connect it with the next clique. Instead, the `ring_of_cliques` graph
350: (4)                simply adds the link without removing any link from the cliques.
351: (4)                """
352: (4)                if num_cliques < 2:
353: (8)                    raise nx.NetworkXError("A ring of cliques must have at least two
cliques")
354: (4)                if clique_size < 2:
355: (8)                    raise nx.NetworkXError("The cliques must have at least two nodes")
356: (4)                G = nx.Graph()
357: (4)                for i in range(num_cliques):
358: (8)                    edges = itertools.combinations(
359: (12)                       range(i * clique_size, i * clique_size + clique_size), 2
360: (8)                    )
361: (8)                    G.add_edges_from(edges)
362: (8)                    G.add_edge(
363: (12)                       i * clique_size + 1, (i + 1) * clique_size % (num_cliques *
clique_size)
364: (8)                    )
365: (4)                return G
366: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
367: (0)            def windmill_graph(n, k):
368: (4)                """Generate a windmill graph.
369: (4)                A windmill graph is a graph of `n` cliques each of size `k` that are all
370: (4)                joined at one node.
371: (4)                It can be thought of as taking a disjoint union of `n` cliques of size
`k`,
372: (4)                selecting one point from each, and contracting all of the selected points.
373: (4)                Alternatively, one could generate `n` cliques of size `k-1` and one node
374: (4)                that is connected to all other nodes in the graph.
375: (4)                Parameters
376: (4)                ----------
377: (4)                n : int
378: (8)                    Number of cliques
379: (4)                k : int
380: (8)                    Size of cliques
381: (4)                Returns
382: (4)                -------
383: (4)                G : NetworkX Graph
384: (8)                    windmill graph with n cliques of size k
385: (4)                Raises
386: (4)                ------
387: (4)                NetworkXError
388: (8)                    If the number of cliques is less than two
389: (8)                    If the size of the cliques are less than two
390: (4)                Examples
391: (4)                --------
392: (4)                >>> G = nx.windmill_graph(4, 5)
393: (4)                Notes
394: (4)                -----
395: (4)                The node labeled `0` will be the node connected to all other nodes.
396: (4)                Note that windmill graphs are usually denoted `Wd(k,n)`, so the parameters
397: (4)                are in the opposite order as the parameters of this method.
398: (4)                """
399: (4)                if n < 2:
400: (8)                    msg = "A windmill graph must have at least two cliques"
401: (8)                    raise nx.NetworkXError(msg)
402: (4)                if k < 2:
403: (8)                    raise nx.NetworkXError("The cliques must have at least two nodes")
404: (4)                G = nx.disjoint_union_all(
405: (8)                    itertools.chain(
406: (12)                       [nx.complete_graph(k)], (nx.complete_graph(k - 1) for _ in range(n
- 1))
```

```
407: (8)                    )
408: (4)                )
409: (4)                G.add_edges_from((0, i) for i in range(k, G.number_of_nodes()))
410: (4)                return G
411: (0)            @py_random_state(3)
412: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
413: (0)            def stochastic_block_model(
414: (4)                sizes, p, nodelist=None, seed=None, directed=False, selfloops=False,
sparse=True
415: (0)            ):
416: (4)                """Returns a stochastic block model graph.
417: (4)                This model partitions the nodes in blocks of arbitrary sizes, and places
418: (4)                edges between pairs of nodes independently, with a probability that
depends
419: (4)                on the blocks.
420: (4)                Parameters
421: (4)                ----------
422: (4)                sizes : list of ints
423: (8)                    Sizes of blocks
424: (4)                p : list of list of floats
425: (8)                    Element (r,s) gives the density of edges going from the nodes
426: (8)                    of group r to nodes of group s.
427: (8)                    p must match the number of groups (len(sizes) == len(p)),
428: (8)                    and it must be symmetric if the graph is undirected.
429: (4)                nodelist : list, optional
430: (8)                    The block tags are assigned according to the node identifiers
431: (8)                    in nodelist. If nodelist is None, then the ordering is the
432: (8)                    range [0,sum(sizes)-1].
433: (4)                seed : integer, random_state, or None (default)
434: (8)                    Indicator of random number generation state.
435: (8)                    See :ref:`Randomness<randomness>`.
436: (4)                directed : boolean optional, default=False
437: (8)                    Whether to create a directed graph or not.
438: (4)                selfloops : boolean optional, default=False
439: (8)                    Whether to include self-loops or not.
440: (4)                sparse: boolean optional, default=True
441: (8)                    Use the sparse heuristic to speed up the generator.
442: (4)                Returns
443: (4)                -------
444: (4)                g : NetworkX Graph or DiGraph
445: (8)                    Stochastic block model graph of size sum(sizes)
446: (4)                Raises
447: (4)                ------
448: (4)                NetworkXError
449: (6)                  If probabilities are not in [0,1].
450: (6)                  If the probability matrix is not square (directed case).
451: (6)                  If the probability matrix is not symmetric (undirected case).
452: (6)                  If the sizes list does not match nodelist or the probability matrix.
453: (6)                  If nodelist contains duplicate.
454: (4)                Examples
455: (4)                --------
456: (4)                >>> sizes = [75, 75, 300]
457: (4)                >>> probs = [[0.25, 0.05, 0.02], [0.05, 0.35, 0.07], [0.02, 0.07, 0.40]]
458: (4)                >>> g = nx.stochastic_block_model(sizes, probs, seed=0)
459: (4)                >>> len(g)
460: (4)                450
461: (4)                >>> H = nx.quotient_graph(g, g.graph["partition"], relabel=True)
462: (4)                >>> for v in H.nodes(data=True):
463: (4)                ...     print(round(v[1]["density"], 3))
464: (4)                0.245
465: (4)                0.348
466: (4)                0.405
467: (4)                >>> for v in H.edges(data=True):
468: (4)                ...     print(round(1.0 * v[2]["weight"] / (sizes[v[0]] * sizes[v[1]]),
3))
469: (4)                0.051
470: (4)                0.022
471: (4)                0.07
472: (4)                See Also
```

```
(4)     --------
(4)     random_partition_graph
(4)     planted_partition_graph
(4)     gaussian_random_partition_graph
(4)     gnp_random_graph
(4)     References
(4)     ----------
(4)     .. [1] Holland, P. W., Laskey, K. B., & Leinhardt, S.,
(11)           "Stochastic blockmodels: First steps",
(11)           Social networks, 5(2), 109-137, 1983.
(4)     """
(4)     if len(sizes) != len(p):
(8)         raise nx.NetworkXException("'sizes' and 'p' do not match.")
(4)     for row in p:
(8)         if len(p) != len(row):
(12)            raise nx.NetworkXException("'p' must be a square matrix.")
(4)     if not directed:
(8)         p_transpose = [list(i) for i in zip(*p)]
(8)         for i in zip(p, p_transpose):
(12)            for j in zip(i[0], i[1]):
(16)                if abs(j[0] - j[1]) > 1e-08:
(20)                    raise nx.NetworkXException("'p' must be symmetric.")
(4)     for row in p:
(8)         for prob in row:
(12)            if prob < 0 or prob > 1:
(16)                raise nx.NetworkXException("Entries of 'p' not in [0,1].")
(4)     if nodelist is not None:
(8)         if len(nodelist) != sum(sizes):
(12)            raise nx.NetworkXException("'nodelist' and 'sizes' do not match.")
(8)         if len(nodelist) != len(set(nodelist)):
(12)            raise nx.NetworkXException("nodelist contains duplicate.")
(4)     else:
(8)         nodelist = range(sum(sizes))
(4)     block_range = range(len(sizes))
(4)     if directed:
(8)         g = nx.DiGraph()
(8)         block_iter = itertools.product(block_range, block_range)
(4)     else:
(8)         g = nx.Graph()
(8)         block_iter = itertools.combinations_with_replacement(block_range, 2)
(4)     size_cumsum = [sum(sizes[0:x]) for x in range(len(sizes) + 1)]
(4)     g.graph["partition"] = [
(8)         set(nodelist[size_cumsum[x] : size_cumsum[x + 1]])
(8)         for x in range(len(size_cumsum) - 1)
(4)     ]
(4)     for block_id, nodes in enumerate(g.graph["partition"]):
(8)         for node in nodes:
(12)            g.add_node(node, block=block_id)
(4)     g.name = "stochastic_block_model"
(4)     parts = g.graph["partition"]
(4)     for i, j in block_iter:
(8)         if i == j:
(12)            if directed:
(16)                if selfloops:
(20)                    edges = itertools.product(parts[i], parts[i])
(16)                else:
(20)                    edges = itertools.permutations(parts[i], 2)
(12)            else:
(16)                edges = itertools.combinations(parts[i], 2)
(16)                if selfloops:
(20)                    edges = itertools.chain(edges, zip(parts[i], parts[i]))
(12)            for e in edges:
(16)                if seed.random() < p[i][j]:
(20)                    g.add_edge(*e)
(8)         else:
(12)            edges = itertools.product(parts[i], parts[j])
(8)         if sparse:
(12)            if p[i][j] == 1:  # Test edges cases p_ij = 0 or 1
(16)                for e in edges:
```

```
(20)                                g.add_edge(*e)
(12)                        elif p[i][j] > 0:
(16)                            while True:
(20)                                try:
(24)                                    lograng = math.log(seed.random())
(24)                                    skip = math.floor(lograng / math.log(1 - p[i][j]))
(24)                                    next(itertools.islice(edges, skip, skip), None)
(24)                                    e = next(edges)
(24)                                    g.add_edge(*e)  # __safe
(20)                                except StopIteration:
(24)                                    break
(8)                     else:
(12)                        for e in edges:
(16)                            if seed.random() < p[i][j]:
(20)                                g.add_edge(*e)  # __safe
(4)                 return g
(0)             def _zipf_rv_below(gamma, xmin, threshold, seed):
(4)                 """Returns a random value chosen from the bounded Zipf distribution.
(4)                 Repeatedly draws values from the Zipf distribution until the
(4)                 threshold is met, then returns that value.
(4)                 """
(4)                 result = nx.utils.zipf_rv(gamma, xmin, seed)
(4)                 while result > threshold:
(8)                     result = nx.utils.zipf_rv(gamma, xmin, seed)
(4)                 return result
(0)             def _powerlaw_sequence(gamma, low, high, condition, length, max_iters, seed):
(4)                 """Returns a list of numbers obeying a constrained power law distribution.
(4)                 ``gamma`` and ``low`` are the parameters for the Zipf distribution.
(4)                 ``high`` is the maximum allowed value for values draw from the Zipf
(4)                 distribution. For more information, see :func:`_zipf_rv_below`.
(4)                 ``condition`` and ``length`` are Boolean-valued functions on
(4)                 lists. While generating the list, random values are drawn and
(4)                 appended to the list until ``length`` is satisfied by the created
(4)                 list. Once ``condition`` is satisfied, the sequence generated in
(4)                 this way is returned.
(4)                 ``max_iters`` indicates the number of times to generate a list
(4)                 satisfying ``length``. If the number of iterations exceeds this
(4)                 value, :exc:`~networkx.exception.ExceededMaxIterations` is raised.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 """
(4)                 for i in range(max_iters):
(8)                     seq = []
(8)                     while not length(seq):
(12)                        seq.append(_zipf_rv_below(gamma, low, high, seed))
(8)                     if condition(seq):
(12)                        return seq
(4)                 raise nx.ExceededMaxIterations("Could not create power law sequence")
(0)             def _hurwitz_zeta(x, q, tolerance):
(4)                 """The Hurwitz zeta function, or the Riemann zeta function of two
arguments.
(4)                 ``x`` must be greater than one and ``q`` must be positive.
(4)                 This function repeatedly computes subsequent partial sums until
(4)                 convergence, as decided by ``tolerance``.
(4)                 """
(4)                 z = 0
(4)                 z_prev = -float("inf")
(4)                 k = 0
(4)                 while abs(z - z_prev) > tolerance:
(8)                     z_prev = z
(8)                     z += 1 / ((k + q) ** x)
(8)                     k += 1
(4)                 return z
(0)             def _generate_min_degree(gamma, average_degree, max_degree, tolerance,
max_iters):
(4)                 """Returns a minimum degree from the given average degree."""
(4)                 try:
(8)                     from scipy.special import zeta
```

```
(4)     except ImportError:
(8)         def zeta(x, q):
(12)            return _hurwitz_zeta(x, q, tolerance)
(4)     min_deg_top = max_degree
(4)     min_deg_bot = 1
(4)     min_deg_mid = (min_deg_top - min_deg_bot) / 2 + min_deg_bot
(4)     itrs = 0
(4)     mid_avg_deg = 0
(4)     while abs(mid_avg_deg - average_degree) > tolerance:
(8)         if itrs > max_iters:
(12)            raise nx.ExceededMaxIterations("Could not match average_degree")
(8)         mid_avg_deg = 0
(8)         for x in range(int(min_deg_mid), max_degree + 1):
(12)            mid_avg_deg += (x ** (-gamma + 1)) / zeta(gamma, min_deg_mid)
(8)         if mid_avg_deg > average_degree:
(12)            min_deg_top = min_deg_mid
(12)            min_deg_mid = (min_deg_top - min_deg_bot) / 2 + min_deg_bot
(8)         else:
(12)            min_deg_bot = min_deg_mid
(12)            min_deg_mid = (min_deg_top - min_deg_bot) / 2 + min_deg_bot
(8)         itrs += 1
(4)     return round(min_deg_mid)
(0) def _generate_communities(degree_seq, community_sizes, mu, max_iters, seed):
(4)     """Returns a list of sets, each of which represents a community.
(4)     ``degree_seq`` is the degree sequence that must be met by the
(4)     graph.
(4)     ``community_sizes`` is the community size distribution that must be
(4)     met by the generated list of sets.
(4)     ``mu`` is a float in the interval [0, 1] indicating the fraction of
(4)     intra-community edges incident to each node.
(4)     ``max_iters`` is the number of times to try to add a node to a
(4)     community. This must be greater than the length of
(4)     ``degree_seq``, otherwise this function will always fail. If
(4)     the number of iterations exceeds this value,
(4)     :exc:`~networkx.exception.ExceededMaxIterations` is raised.
(4)     seed : integer, random_state, or None (default)
(8)         Indicator of random number generation state.
(8)         See :ref:`Randomness<randomness>`.
(4)     The communities returned by this are sets of integers in the set {0,
(4)     ..., *n* - 1}, where *n* is the length of ``degree_seq``.
(4)     """
(4)     result = [set() for _ in community_sizes]
(4)     n = len(degree_seq)
(4)     free = list(range(n))
(4)     for i in range(max_iters):
(8)         v = free.pop()
(8)         c = seed.choice(range(len(community_sizes)))
(8)         s = round(degree_seq[v] * (1 - mu))
(8)         if s < community_sizes[c]:
(12)            result[c].add(v)
(8)         else:
(12)            free.append(v)
(8)         if len(result[c]) > community_sizes[c]:
(12)            free.append(result[c].pop())
(8)         if not free:
(12)            return result
(4)     msg = "Could not assign communities; try increasing min_community"
(4)     raise nx.ExceededMaxIterations(msg)
(0) @py_random_state(11)
(0) @nx._dispatchable(graphs=None, returns_graph=True)
(0) def LFR_benchmark_graph(
(4)     n,
(4)     tau1,
(4)     tau2,
(4)     mu,
(4)     average_degree=None,
(4)     min_degree=None,
(4)     max_degree=None,
(4)     min_community=None,
```

```
(4)                 max_community=None,
(4)                 tol=1.0e-7,
(4)                 max_iters=500,
(4)                 seed=None,
(0)             ):
(4)                 r"""Returns the LFR benchmark graph.
(4)                 This algorithm proceeds as follows:
(4)                 1) Find a degree sequence with a power law distribution, and minimum
(7)                    value ``min_degree``, which has approximate average degree
(7)                    ``average_degree``. This is accomplished by either
(7)                    a) specifying ``min_degree`` and not ``average_degree``,
(7)                    b) specifying ``average_degree`` and not ``min_degree``, in which
(10)                      case a suitable minimum degree will be found.
(7)                    ``max_degree`` can also be specified, otherwise it will be set to
(7)                    ``n``. Each node *u* will have $\mu \mathrm{deg}(u)$ edges
(7)                    joining it to nodes in communities other than its own and $(1 -
(7)                    \mu) \mathrm{deg}(u)$ edges joining it to nodes in its own
(7)                    community.
(4)                 2) Generate community sizes according to a power law distribution
(7)                    with exponent ``tau2``. If ``min_community`` and
(7)                    ``max_community`` are not specified they will be selected to be
(7)                    ``min_degree`` and ``max_degree``, respectively.  Community sizes
(7)                    are generated until the sum of their sizes equals ``n``.
(4)                 3) Each node will be randomly assigned a community with the
(7)                    condition that the community is large enough for the node's
(7)                    intra-community degree, $(1 - \mu) \mathrm{deg}(u)$ as
(7)                    described in step 2. If a community grows too large, a random node
(7)                    will be selected for reassignment to a new community, until all
(7)                    nodes have been assigned a community.
(4)                 4) Each node *u* then adds $(1 - \mu) \mathrm{deg}(u)$
(7)                    intra-community edges and $\mu \mathrm{deg}(u)$ inter-community
(7)                    edges.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     Number of nodes in the created graph.
(4)                 tau1 : float
(8)                     Power law exponent for the degree distribution of the created
(8)                     graph. This value must be strictly greater than one.
(4)                 tau2 : float
(8)                     Power law exponent for the community size distribution in the
(8)                     created graph. This value must be strictly greater than one.
(4)                 mu : float
(8)                     Fraction of inter-community edges incident to each node. This
(8)                     value must be in the interval [0, 1].
(4)                 average_degree : float
(8)                     Desired average degree of nodes in the created graph. This value
(8)                     must be in the interval [0, *n*]. Exactly one of this and
(8)                     ``min_degree`` must be specified, otherwise a
(8)                     :exc:`NetworkXError` is raised.
(4)                 min_degree : int
(8)                     Minimum degree of nodes in the created graph. This value must be
(8)                     in the interval [0, *n*]. Exactly one of this and
(8)                     ``average_degree`` must be specified, otherwise a
(8)                     :exc:`NetworkXError` is raised.
(4)                 max_degree : int
(8)                     Maximum degree of nodes in the created graph. If not specified,
(8)                     this is set to ``n``, the total number of nodes in the graph.
(4)                 min_community : int
(8)                     Minimum size of communities in the graph. If not specified, this
(8)                     is set to ``min_degree``.
(4)                 max_community : int
(8)                     Maximum size of communities in the graph. If not specified, this
(8)                     is set to ``n``, the total number of nodes in the graph.
(4)                 tol : float
(8)                     Tolerance when comparing floats, specifically when comparing
(8)                     average degree values.
(4)                 max_iters : int
(8)                     Maximum number of iterations to try to create the community sizes,
```

```
    degree distribution, and community affiliations.
seed : integer, random_state, or None (default)
    Indicator of random number generation state.
    See :ref:`Randomness<randomness>`.
Returns
-------
G : NetworkX graph
    The LFR benchmark graph generated according to the specified
    parameters.
    Each node in the graph has a node attribute ``'community'`` that
    stores the community (that is, the set of nodes) that includes
    it.
Raises
------
NetworkXError
    If any of the parameters do not meet their upper and lower bounds:
    - ``tau1`` and ``tau2`` must be strictly greater than 1.
    - ``mu`` must be in [0, 1].
    - ``max_degree`` must be in {1, ..., *n*}.
    - ``min_community`` and ``max_community`` must be in {0, ...,
      *n*}.
    If not exactly one of ``average_degree`` and ``min_degree`` is
    specified.
    If ``min_degree`` is not specified and a suitable ``min_degree``
    cannot be found.
ExceededMaxIterations
    If a valid degree sequence cannot be created within
    ``max_iters`` number of iterations.
    If a valid set of community sizes cannot be created within
    ``max_iters`` number of iterations.
    If a valid community assignment cannot be created within ``10 *
    n * max_iters`` number of iterations.
Examples
--------
Basic usage::
    >>> from networkx.generators.community import LFR_benchmark_graph
    >>> n = 250
    >>> tau1 = 3
    >>> tau2 = 1.5
    >>> mu = 0.1
    >>> G = LFR_benchmark_graph(
    ...     n, tau1, tau2, mu, average_degree=5, min_community=20, seed=10
    ... )
Continuing the example above, you can get the communities from the
node attributes of the graph::
    >>> communities = {frozenset(G.nodes[v]["community"]) for v in G}
Notes
-----
This algorithm differs slightly from the original way it was
presented in [1].
1) Rather than connecting the graph via a configuration model then
   rewiring to match the intra-community and inter-community
   degrees, we do this wiring explicitly at the end, which should be
   equivalent.
2) The code posted on the author's website [2] calculates the random
   power law distributed variables and their average using
   continuous approximations, whereas we use the discrete
   distributions here as both degree and community size are
   discrete.
Though the authors describe the algorithm as quite robust, testing
during development indicates that a somewhat narrower parameter set
is likely to successfully produce a graph. Some suggestions have
been provided in the event of exceptions.
References
----------
.. [1] "Benchmark graphs for testing community detection algorithms",
       Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi,
       Phys. Rev. E 78, 046110 2008
.. [2] https://www.santofortunato.net/resources
```

```
816: (4)                """
817: (4)                if not tau1 > 1:
818: (8)                    raise nx.NetworkXError("tau1 must be greater than one")
819: (4)                if not tau2 > 1:
820: (8)                    raise nx.NetworkXError("tau2 must be greater than one")
821: (4)                if not 0 <= mu <= 1:
822: (8)                    raise nx.NetworkXError("mu must be in the interval [0, 1]")
823: (4)                if max_degree is None:
824: (8)                    max_degree = n
825: (4)                elif not 0 < max_degree <= n:
826: (8)                    raise nx.NetworkXError("max_degree must be in the interval (0, n]")
827: (4)                if not ((min_degree is None) ^ (average_degree is None)):
828: (8)                    raise nx.NetworkXError(
829: (12)                       "Must assign exactly one of min_degree and average_degree"
830: (8)                    )
831: (4)                if min_degree is None:
832: (8)                    min_degree = _generate_min_degree(
833: (12)                       tau1, average_degree, max_degree, tol, max_iters
834: (8)                    )
835: (4)                low, high = min_degree, max_degree
836: (4)                def condition(seq):
837: (8)                    return sum(seq) % 2 == 0
838: (4)                def length(seq):
839: (8)                    return len(seq) >= n
840: (4)                deg_seq = _powerlaw_sequence(tau1, low, high, condition, length,
max_iters, seed)
841: (4)                if min_community is None:
842: (8)                    min_community = min(deg_seq)
843: (4)                if max_community is None:
844: (8)                    max_community = max(deg_seq)
845: (4)                low, high = min_community, max_community
846: (4)                def condition(seq):
847: (8)                    return sum(seq) == n
848: (4)                def length(seq):
849: (8)                    return sum(seq) >= n
850: (4)                comms = _powerlaw_sequence(tau2, low, high, condition, length, max_iters,
seed)
851: (4)                max_iters *= 10 * n
852: (4)                communities = _generate_communities(deg_seq, comms, mu, max_iters, seed)
853: (4)                G = nx.Graph()
854: (4)                G.add_nodes_from(range(n))
855: (4)                for c in communities:
856: (8)                    for u in c:
857: (12)                       while G.degree(u) < round(deg_seq[u] * (1 - mu)):
858: (16)                           v = seed.choice(list(c))
859: (16)                           G.add_edge(u, v)
860: (12)                       while G.degree(u) < deg_seq[u]:
861: (16)                           v = seed.choice(range(n))
862: (16)                           if v not in c:
863: (20)                               G.add_edge(u, v)
864: (12)                       G.nodes[u]["community"] = c
865: (4)                return G
```

File 431 - degree_seq.py:

```
1: (0)              """Generate graphs with a given degree sequence or expected degree sequence.
2: (0)              """
3: (0)              import heapq
4: (0)              import math
5: (0)              from itertools import chain, combinations, zip_longest
6: (0)              from operator import itemgetter
7: (0)              import networkx as nx
8: (0)              from networkx.utils import py_random_state, random_weighted_sample
9: (0)              __all__ = [
10: (4)                 "configuration_model",
11: (4)                 "directed_configuration_model",
12: (4)                 "expected_degree_graph",
```

```
 13: (4)                 "havel_hakimi_graph",
 14: (4)                 "directed_havel_hakimi_graph",
 15: (4)                 "degree_sequence_tree",
 16: (4)                 "random_degree_sequence_graph",
 17: (0)             ]
 18: (0)             chaini = chain.from_iterable
 19: (0)             def _to_stublist(degree_sequence):
 20: (4)                 """Returns a list of degree-repeated node numbers.
 21: (4)                 ``degree_sequence`` is a list of nonnegative integers representing
 22: (4)                 the degrees of nodes in a graph.
 23: (4)                 This function returns a list of node numbers with multiplicities
 24: (4)                 according to the given degree sequence. For example, if the first
 25: (4)                 element of ``degree_sequence`` is ``3``, then the first node number,
 26: (4)                 ``0``, will appear at the head of the returned list three times. The
 27: (4)                 node numbers are assumed to be the numbers zero through
 28: (4)                 ``len(degree_sequence) - 1``.
 29: (4)                 Examples
 30: (4)                 --------
 31: (4)                 >>> degree_sequence = [1, 2, 3]
 32: (4)                 >>> _to_stublist(degree_sequence)
 33: (4)                 [0, 1, 1, 2, 2, 2]
 34: (4)                 If a zero appears in the sequence, that means the node exists but
 35: (4)                 has degree zero, so that number will be skipped in the returned
 36: (4)                 list::
 37: (4)                 >>> degree_sequence = [2, 0, 1]
 38: (4)                 >>> _to_stublist(degree_sequence)
 39: (4)                 [0, 0, 2]
 40: (4)                 """
 41: (4)                 return list(chaini([n] * d for n, d in enumerate(degree_sequence)))
 42: (0)             def _configuration_model(
 43: (4)                 deg_sequence, create_using, directed=False, in_deg_sequence=None,
seed=None
 44: (0)             ):
 45: (4)                 """Helper function for generating either undirected or directed
 46: (4)                 configuration model graphs.
 47: (4)                 ``deg_sequence`` is a list of nonnegative integers representing the
 48: (4)                 degree of the node whose label is the index of the list element.
 49: (4)                 ``create_using`` see :func:`~networkx.empty_graph`.
 50: (4)                 ``directed`` and ``in_deg_sequence`` are required if you want the
 51: (4)                 returned graph to be generated using the directed configuration
 52: (4)                 model algorithm. If ``directed`` is ``False``, then ``deg_sequence``
 53: (4)                 is interpreted as the degree sequence of an undirected graph and
 54: (4)                 ``in_deg_sequence`` is ignored. Otherwise, if ``directed`` is
 55: (4)                 ``True``, then ``deg_sequence`` is interpreted as the out-degree
 56: (4)                 sequence and ``in_deg_sequence`` as the in-degree sequence of a
 57: (4)                 directed graph.
 58: (4)                 .. note::
 59: (7)                    ``deg_sequence`` and ``in_deg_sequence`` need not be the same
 60: (7)                    length.
 61: (4)                 ``seed`` is a random.Random or numpy.random.RandomState instance
 62: (4)                 This function returns a graph, directed if and only if ``directed``
 63: (4)                 is ``True``, generated according to the configuration model
 64: (4)                 algorithm. For more information on the algorithm, see the
 65: (4)                 :func:`configuration_model` or :func:`directed_configuration_model`
 66: (4)                 functions.
 67: (4)                 """
 68: (4)                 n = len(deg_sequence)
 69: (4)                 G = nx.empty_graph(n, create_using)
 70: (4)                 if n == 0:
 71: (8)                     return G
 72: (4)                 if directed:
 73: (8)                     pairs = zip_longest(deg_sequence, in_deg_sequence, fillvalue=0)
 74: (8)                     out_deg, in_deg = zip(*pairs)
 75: (8)                     out_stublist = _to_stublist(out_deg)
 76: (8)                     in_stublist = _to_stublist(in_deg)
 77: (8)                     seed.shuffle(out_stublist)
 78: (8)                     seed.shuffle(in_stublist)
 79: (4)                 else:
 80: (8)                     stublist = _to_stublist(deg_sequence)
```

```
 81: (8)                    n = len(stublist)
 82: (8)                    half = n // 2
 83: (8)                    seed.shuffle(stublist)
 84: (8)                    out_stublist, in_stublist = stublist[:half], stublist[half:]
 85: (4)                G.add_edges_from(zip(out_stublist, in_stublist))
 86: (4)                return G
 87: (0)            @py_random_state(2)
 88: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
 89: (0)            def configuration_model(deg_sequence, create_using=None, seed=None):
 90: (4)                """Returns a random graph with the given degree sequence.
 91: (4)                The configuration model generates a random pseudograph (graph with
 92: (4)                parallel edges and self loops) by randomly assigning edges to
 93: (4)                match the given degree sequence.
 94: (4)                Parameters
 95: (4)                ----------
 96: (4)                deg_sequence :  list of nonnegative integers
 97: (8)                    Each list entry corresponds to the degree of a node.
 98: (4)                create_using : NetworkX graph constructor, optional (default MultiGraph)
 99: (8)                    Graph type to create. If graph instance, then cleared before
populated.
100: (4)                seed : integer, random_state, or None (default)
101: (8)                    Indicator of random number generation state.
102: (8)                    See :ref:`Randomness<randomness>`.
103: (4)                Returns
104: (4)                -------
105: (4)                G : MultiGraph
106: (8)                    A graph with the specified degree sequence.
107: (8)                    Nodes are labeled starting at 0 with an index
108: (8)                    corresponding to the position in deg_sequence.
109: (4)                Raises
110: (4)                ------
111: (4)                NetworkXError
112: (8)                    If the degree sequence does not have an even sum.
113: (4)                See Also
114: (4)                --------
115: (4)                is_graphical
116: (4)                Notes
117: (4)                -----
118: (4)                As described by Newman [1]_.
119: (4)                A non-graphical degree sequence (not realizable by some simple
120: (4)                graph) is allowed since this function returns graphs with self
121: (4)                loops and parallel edges.  An exception is raised if the degree
122: (4)                sequence does not have an even sum.
123: (4)                This configuration model construction process can lead to
124: (4)                duplicate edges and loops.  You can remove the self-loops and
125: (4)                parallel edges (see below) which will likely result in a graph
126: (4)                that doesn't have the exact degree sequence specified.
127: (4)                The density of self-loops and parallel edges tends to decrease as
128: (4)                the number of nodes increases. However, typically the number of
129: (4)                self-loops will approach a Poisson distribution with a nonzero mean,
130: (4)                and similarly for the number of parallel edges.  Consider a node
131: (4)                with *k* stubs. The probability of being joined to another stub of
132: (4)                the same node is basically (*k* - *1*) / *N*, where *k* is the
133: (4)                degree and *N* is the number of nodes. So the probability of a
134: (4)                self-loop scales like *c* / *N* for some constant *c*. As *N* grows,
135: (4)                this means we expect *c* self-loops. Similarly for parallel edges.
136: (4)                References
137: (4)                ----------
138: (4)                .. [1] M.E.J. Newman, "The structure and function of complex networks",
139: (7)                   SIAM REVIEW 45-2, pp 167-256, 2003.
140: (4)                Examples
141: (4)                --------
142: (4)                You can create a degree sequence following a particular distribution
143: (4)                by using the one of the distribution functions in
144: (4)                :mod:`~networkx.utils.random_sequence` (or one of your own). For
145: (4)                example, to create an undirected multigraph on one hundred nodes
146: (4)                with degree sequence chosen from the power law distribution:
147: (4)                >>> sequence = nx.random_powerlaw_tree_sequence(100, tries=5000)
148: (4)                >>> G = nx.configuration_model(sequence)
```

```
149: (4)                >>> len(G)
150: (4)                100
151: (4)                >>> actual_degrees = [d for v, d in G.degree()]
152: (4)                >>> actual_degrees == sequence
153: (4)                True
154: (4)                The returned graph is a multigraph, which may have parallel
155: (4)                edges. To remove any parallel edges from the returned graph:
156: (4)                >>> G = nx.Graph(G)
157: (4)                Similarly, to remove self-loops:
158: (4)                >>> G.remove_edges_from(nx.selfloop_edges(G))
159: (4)                """
160: (4)                if sum(deg_sequence) % 2 != 0:
161: (8)                    msg = "Invalid degree sequence: sum of degrees must be even, not odd"
162: (8)                    raise nx.NetworkXError(msg)
163: (4)                G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
164: (4)                if G.is_directed():
165: (8)                    raise nx.NetworkXNotImplemented("not implemented for directed graphs")
166: (4)                G = _configuration_model(deg_sequence, G, seed=seed)
167: (4)                return G
168: (0)            @py_random_state(3)
169: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
170: (0)            def directed_configuration_model(
171: (4)                in_degree_sequence, out_degree_sequence, create_using=None, seed=None
172: (0)            ):
173: (4)                """Returns a directed_random graph with the given degree sequences.
174: (4)                The configuration model generates a random directed pseudograph
175: (4)                (graph with parallel edges and self loops) by randomly assigning
176: (4)                edges to match the given degree sequences.
177: (4)                Parameters
178: (4)                ----------
179: (4)                in_degree_sequence :  list of nonnegative integers
180: (7)                   Each list entry corresponds to the in-degree of a node.
181: (4)                out_degree_sequence :  list of nonnegative integers
182: (7)                   Each list entry corresponds to the out-degree of a node.
183: (4)                create_using : NetworkX graph constructor, optional (default MultiDiGraph)
184: (8)                    Graph type to create. If graph instance, then cleared before
populated.
185: (4)                seed : integer, random_state, or None (default)
186: (8)                    Indicator of random number generation state.
187: (8)                    See :ref:`Randomness<randomness>`.
188: (4)                Returns
189: (4)                -------
190: (4)                G : MultiDiGraph
191: (8)                    A graph with the specified degree sequences.
192: (8)                    Nodes are labeled starting at 0 with an index
193: (8)                    corresponding to the position in deg_sequence.
194: (4)                Raises
195: (4)                ------
196: (4)                NetworkXError
197: (8)                    If the degree sequences do not have the same sum.
198: (4)                See Also
199: (4)                --------
200: (4)                configuration_model
201: (4)                Notes
202: (4)                -----
203: (4)                Algorithm as described by Newman [1]_.
204: (4)                A non-graphical degree sequence (not realizable by some simple
205: (4)                graph) is allowed since this function returns graphs with self
206: (4)                loops and parallel edges.  An exception is raised if the degree
207: (4)                sequences does not have the same sum.
208: (4)                This configuration model construction process can lead to
209: (4)                duplicate edges and loops.  You can remove the self-loops and
210: (4)                parallel edges (see below) which will likely result in a graph
211: (4)                that doesn't have the exact degree sequence specified.  This
212: (4)                "finite-size effect" decreases as the size of the graph increases.
213: (4)                References
214: (4)                ----------
215: (4)                .. [1] Newman, M. E. J. and Strogatz, S. H. and Watts, D. J.
216: (7)                   Random graphs with arbitrary degree distributions and their
```

```
applications
217: (7)                   Phys. Rev. E, 64, 026118 (2001)
218: (4)                Examples
219: (4)                --------
220: (4)                One can modify the in- and out-degree sequences from an existing
221: (4)                directed graph in order to create a new directed graph. For example,
222: (4)                here we modify the directed path graph:
223: (4)                >>> D = nx.DiGraph([(0, 1), (1, 2), (2, 3)])
224: (4)                >>> din = list(d for n, d in D.in_degree())
225: (4)                >>> dout = list(d for n, d in D.out_degree())
226: (4)                >>> din.append(1)
227: (4)                >>> dout[0] = 2
228: (4)                >>> # We now expect an edge from node 0 to a new node, node 3.
229: (4)                ... D = nx.directed_configuration_model(din, dout)
230: (4)                The returned graph is a directed multigraph, which may have parallel
231: (4)                edges. To remove any parallel edges from the returned graph:
232: (4)                >>> D = nx.DiGraph(D)
233: (4)                Similarly, to remove self-loops:
234: (4)                >>> D.remove_edges_from(nx.selfloop_edges(D))
235: (4)                """
236: (4)                if sum(in_degree_sequence) != sum(out_degree_sequence):
237: (8)                    msg = "Invalid degree sequences: sequences must have equal sums"
238: (8)                    raise nx.NetworkXError(msg)
239: (4)                if create_using is None:
240: (8)                    create_using = nx.MultiDiGraph
241: (4)                G = _configuration_model(
242: (8)                    out_degree_sequence,
243: (8)                    create_using,
244: (8)                    directed=True,
245: (8)                    in_deg_sequence=in_degree_sequence,
246: (8)                    seed=seed,
247: (4)                )
248: (4)                name = "directed configuration_model {} nodes {} edges"
249: (4)                return G
250: (0)            @py_random_state(1)
251: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
252: (0)            def expected_degree_graph(w, seed=None, selfloops=True):
253: (4)                r"""Returns a random graph with given expected degrees.
254: (4)                Given a sequence of expected degrees $W=(w_0,w_1,\ldots,w_{n-1})$
255: (4)                of length $n$ this algorithm assigns an edge between node $u$ and
256: (4)                node $v$ with probability
257: (4)                .. math::
258: (7)                   p_{uv} = \frac{w_u w_v}{\sum_k w_k} .
259: (4)                Parameters
260: (4)                ----------
261: (4)                w : list
262: (8)                    The list of expected degrees.
263: (4)                selfloops: bool (default=True)
264: (8)                    Set to False to remove the possibility of self-loop edges.
265: (4)                seed : integer, random_state, or None (default)
266: (8)                    Indicator of random number generation state.
267: (8)                    See :ref:`Randomness<randomness>`.
268: (4)                Returns
269: (4)                -------
270: (4)                Graph
271: (4)                Examples
272: (4)                --------
273: (4)                >>> z = [10 for i in range(100)]
274: (4)                >>> G = nx.expected_degree_graph(z)
275: (4)                Notes
276: (4)                -----
277: (4)                The nodes have integer labels corresponding to index of expected degrees
278: (4)                input sequence.
279: (4)                The complexity of this algorithm is $\mathcal{O}(n+m)$ where $n$ is the
280: (4)                number of nodes and $m$ is the expected number of edges.
281: (4)                The model in [1]_ includes the possibility of self-loop edges.
282: (4)                Set selfloops=False to produce a graph without self loops.
283: (4)                For finite graphs this model doesn't produce exactly the given
284: (4)                expected degree sequence.  Instead the expected degrees are as
```

```
(4)                 follows.
(4)                 For the case without self loops (selfloops=False),
(4)                 .. math::
(7)                    E[deg(u)] = \sum_{v \ne u} p_{uv}
(16)                            = w_u \left( 1 - \frac{w_u}{\sum_k w_k} \right) .
(4)                 NetworkX uses the standard convention that a self-loop edge counts 2
(4)                 in the degree of a node, so with self loops (selfloops=True),
(4)                 .. math::
(7)                    E[deg(u)] =  \sum_{v \ne u} p_{uv}  + 2 p_{uu}
(16)                            = w_u \left( 1 + \frac{w_u}{\sum_k w_k} \right) .
(4)                 References
(4)                 ----------
(4)                 .. [1] Fan Chung and L. Lu, Connected components in random graphs with
(7)                    given expected degree sequences, Ann. Combinatorics, 6,
(7)                    pp. 125-145, 2002.
(4)                 .. [2] Joel Miller and Aric Hagberg,
(7)                    Efficient generation of networks with given expected degrees,
(7)                    in Algorithms and Models for the Web-Graph (WAW 2011),
(7)                    Alan Frieze, Paul Horn, and Paweł Prałat (Eds), LNCS 6732,
(7)                    pp. 115-126, 2011.
(4)                 """
(4)                 n = len(w)
(4)                 G = nx.empty_graph(n)
(4)                 if n == 0 or max(w) == 0:
(8)                     return G
(4)                 rho = 1 / sum(w)
(4)                 order = sorted(enumerate(w), key=itemgetter(1), reverse=True)
(4)                 mapping = {c: u for c, (u, v) in enumerate(order)}
(4)                 seq = [v for u, v in order]
(4)                 last = n
(4)                 if not selfloops:
(8)                     last -= 1
(4)                 for u in range(last):
(8)                     v = u
(8)                     if not selfloops:
(12)                        v += 1
(8)                     factor = seq[u] * rho
(8)                     p = min(seq[v] * factor, 1)
(8)                     while v < n and p > 0:
(12)                        if p != 1:
(16)                            r = seed.random()
(16)                            v += math.floor(math.log(r, 1 - p))
(12)                        if v < n:
(16)                            q = min(seq[v] * factor, 1)
(16)                            if seed.random() < q / p:
(20)                                G.add_edge(mapping[u], mapping[v])
(16)                            v += 1
(16)                            p = q
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def havel_hakimi_graph(deg_sequence, create_using=None):
(4)                 """Returns a simple graph with given degree sequence constructed
(4)                 using the Havel-Hakimi algorithm.
(4)                 Parameters
(4)                 ----------
(4)                 deg_sequence: list of integers
(8)                     Each integer corresponds to the degree of a node (need not be sorted).
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(8)                     Graph type to create. If graph instance, then cleared before
populated.
(8)                     Directed graphs are not allowed.
(4)                 Raises
(4)                 ------
(4)                 NetworkXException
(8)                     For a non-graphical degree sequence (i.e. one
(8)                     not realizable by some simple graph).
(4)                 Notes
(4)                 -----
(4)                 The Havel-Hakimi algorithm constructs a simple graph by
```

```
353: (4)                successively connecting the node of highest degree to other nodes
354: (4)                of highest degree, resorting remaining nodes by degree, and
355: (4)                repeating the process. The resulting graph has a high
356: (4)                degree-associativity.  Nodes are labeled 1,.., len(deg_sequence),
357: (4)                corresponding to their position in deg_sequence.
358: (4)                The basic algorithm is from Hakimi [1]_ and was generalized by
359: (4)                Kleitman and Wang [2]_.
360: (4)                References
361: (4)                ----------
362: (4)                .. [1] Hakimi S., On Realizability of a Set of Integers as
363: (7)                   Degrees of the Vertices of a Linear Graph. I,
364: (7)                   Journal of SIAM, 10(3), pp. 496-506 (1962)
365: (4)                .. [2] Kleitman D.J. and Wang D.L.
366: (7)                   Algorithms for Constructing Graphs and Digraphs with Given Valences
367: (7)                   and Factors  Discrete Mathematics, 6(1), pp. 79-88 (1973)
368: (4)                """
369: (4)                if not nx.is_graphical(deg_sequence):
370: (8)                    raise nx.NetworkXError("Invalid degree sequence")
371: (4)                p = len(deg_sequence)
372: (4)                G = nx.empty_graph(p, create_using)
373: (4)                if G.is_directed():
374: (8)                    raise nx.NetworkXError("Directed graphs are not supported")
375: (4)                num_degs = [[] for i in range(p)]
376: (4)                dmax, dsum, n = 0, 0, 0
377: (4)                for d in deg_sequence:
378: (8)                    if d > 0:
379: (12)                       num_degs[d].append(n)
380: (12)                       dmax, dsum, n = max(dmax, d), dsum + d, n + 1
381: (4)                if n == 0:
382: (8)                    return G
383: (4)                modstubs = [(0, 0)] * (dmax + 1)
384: (4)                while n > 0:
385: (8)                    while len(num_degs[dmax]) == 0:
386: (12)                       dmax -= 1
387: (8)                    if dmax > n - 1:
388: (12)                       raise nx.NetworkXError("Non-graphical integer sequence")
389: (8)                    source = num_degs[dmax].pop()
390: (8)                    n -= 1
391: (8)                    mslen = 0
392: (8)                    k = dmax
393: (8)                    for i in range(dmax):
394: (12)                       while len(num_degs[k]) == 0:
395: (16)                           k -= 1
396: (12)                       target = num_degs[k].pop()
397: (12)                       G.add_edge(source, target)
398: (12)                       n -= 1
399: (12)                       if k > 1:
400: (16)                           modstubs[mslen] = (k - 1, target)
401: (16)                           mslen += 1
402: (8)                    for i in range(mslen):
403: (12)                       (stubval, stubtarget) = modstubs[i]
404: (12)                       num_degs[stubval].append(stubtarget)
405: (12)                       n += 1
406: (4)                return G
407: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
408: (0)            def directed_havel_hakimi_graph(in_deg_sequence, out_deg_sequence,
create_using=None):
409: (4)                """Returns a directed graph with the given degree sequences.
410: (4)                Parameters
411: (4)                ----------
412: (4)                in_deg_sequence :  list of integers
413: (8)                    Each list entry corresponds to the in-degree of a node.
414: (4)                out_deg_sequence : list of integers
415: (8)                    Each list entry corresponds to the out-degree of a node.
416: (4)                create_using : NetworkX graph constructor, optional (default DiGraph)
417: (8)                    Graph type to create. If graph instance, then cleared before
populated.
418: (4)                Returns
419: (4)                -------
```

```
    G : DiGraph
        A graph with the specified degree sequences.
        Nodes are labeled starting at 0 with an index
        corresponding to the position in deg_sequence
    Raises
    ------
    NetworkXError
        If the degree sequences are not digraphical.
    See Also
    --------
    configuration_model
    Notes
    -----
    Algorithm as described by Kleitman and Wang [1]_.
    References
    ----------
    .. [1] D.J. Kleitman and D.L. Wang
       Algorithms for Constructing Graphs and Digraphs with Given Valences
       and Factors Discrete Mathematics, 6(1), pp. 79-88 (1973)
    """
    in_deg_sequence = nx.utils.make_list_of_ints(in_deg_sequence)
    out_deg_sequence = nx.utils.make_list_of_ints(out_deg_sequence)
    sumin, sumout = 0, 0
    nin, nout = len(in_deg_sequence), len(out_deg_sequence)
    maxn = max(nin, nout)
    G = nx.empty_graph(maxn, create_using, default=nx.DiGraph)
    if maxn == 0:
        return G
    maxin = 0
    stubheap, zeroheap = [], []
    for n in range(maxn):
        in_deg, out_deg = 0, 0
        if n < nout:
            out_deg = out_deg_sequence[n]
        if n < nin:
            in_deg = in_deg_sequence[n]
        if in_deg < 0 or out_deg < 0:
            raise nx.NetworkXError(
                "Invalid degree sequences. Sequence values must be positive."
            )
        sumin, sumout, maxin = sumin + in_deg, sumout + out_deg, max(maxin, in_deg)
        if in_deg > 0:
            stubheap.append((-1 * out_deg, -1 * in_deg, n))
        elif out_deg > 0:
            zeroheap.append((-1 * out_deg, n))
    if sumin != sumout:
        raise nx.NetworkXError(
            "Invalid degree sequences. Sequences must have equal sums."
        )
    heapq.heapify(stubheap)
    heapq.heapify(zeroheap)
    modstubs = [(0, 0, 0)] * (maxin + 1)
    while stubheap:
        (freeout, freein, target) = heapq.heappop(stubheap)
        freein *= -1
        if freein > len(stubheap) + len(zeroheap):
            raise nx.NetworkXError("Non-digraphical integer sequence")
        mslen = 0
        for i in range(freein):
            if zeroheap and (not stubheap or stubheap[0][0] > zeroheap[0][0]):
                (stubout, stubsource) = heapq.heappop(zeroheap)
                stubin = 0
            else:
                (stubout, stubin, stubsource) = heapq.heappop(stubheap)
            if stubout == 0:
                raise nx.NetworkXError("Non-digraphical integer sequence")
            G.add_edge(stubsource, target)
            if stubout + 1 < 0 or stubin < 0:
```

```
488: (16)                            modstubs[mslen] = (stubout + 1, stubin, stubsource)
489: (16)                            mslen += 1
490: (8)                    for i in range(mslen):
491: (12)                       stub = modstubs[i]
492: (12)                       if stub[1] < 0:
493: (16)                           heapq.heappush(stubheap, stub)
494: (12)                       else:
495: (16)                           heapq.heappush(zeroheap, (stub[0], stub[2]))
496: (8)                    if freeout < 0:
497: (12)                       heapq.heappush(zeroheap, (freeout, target))
498: (4)                return G
499: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
500: (0)            def degree_sequence_tree(deg_sequence, create_using=None):
501: (4)                """Make a tree for the given degree sequence.
502: (4)                A tree has #nodes-#edges=1 so
503: (4)                the degree sequence must have
504: (4)                len(deg_sequence)-sum(deg_sequence)/2=1
505: (4)                """
506: (4)                degree_sum = sum(deg_sequence)
507: (4)                if degree_sum % 2 != 0:
508: (8)                    msg = "Invalid degree sequence: sum of degrees must be even, not odd"
509: (8)                    raise nx.NetworkXError(msg)
510: (4)                if len(deg_sequence) - degree_sum // 2 != 1:
511: (8)                    msg = (
512: (12)                       "Invalid degree sequence: tree must have number of nodes equal"
513: (12)                       " to one less than the number of edges"
514: (8)                    )
515: (8)                    raise nx.NetworkXError(msg)
516: (4)                G = nx.empty_graph(0, create_using)
517: (4)                if G.is_directed():
518: (8)                    raise nx.NetworkXError("Directed Graph not supported")
519: (4)                deg = sorted((s for s in deg_sequence if s > 1), reverse=True)
520: (4)                n = len(deg) + 2
521: (4)                nx.add_path(G, range(n))
522: (4)                last = n
523: (4)                for source in range(1, n - 1):
524: (8)                    nedges = deg.pop() - 2
525: (8)                    for target in range(last, last + nedges):
526: (12)                       G.add_edge(source, target)
527: (8)                    last += nedges
528: (4)                if len(G) > len(deg_sequence):
529: (8)                    G.remove_node(0)
530: (4)                return G
531: (0)            @py_random_state(1)
532: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
533: (0)            def random_degree_sequence_graph(sequence, seed=None, tries=10):
534: (4)                r"""Returns a simple random graph with the given degree sequence.
535: (4)                If the maximum degree $d_m$ in the sequence is $O(m^{1/4})$ then the
536: (4)                algorithm produces almost uniform random graphs in $O(m d_m)$ time
537: (4)                where $m$ is the number of edges.
538: (4)                Parameters
539: (4)                ----------
540: (4)                sequence :  list of integers
541: (8)                    Sequence of degrees
542: (4)                seed : integer, random_state, or None (default)
543: (8)                    Indicator of random number generation state.
544: (8)                    See :ref:`Randomness<randomness>`.
545: (4)                tries : int, optional
546: (8)                    Maximum number of tries to create a graph
547: (4)                Returns
548: (4)                -------
549: (4)                G : Graph
550: (8)                    A graph with the specified degree sequence.
551: (8)                    Nodes are labeled starting at 0 with an index
552: (8)                    corresponding to the position in the sequence.
553: (4)                Raises
554: (4)                ------
555: (4)                NetworkXUnfeasible
556: (8)                    If the degree sequence is not graphical.
```

```
(4)                 NetworkXError
(8)                     If a graph is not produced in specified number of tries
(4)                 See Also
(4)                 --------
(4)                 is_graphical, configuration_model
(4)                 Notes
(4)                 -----
(4)                 The generator algorithm [1]_ is not guaranteed to produce a graph.
(4)                 References
(4)                 ----------
(4)                 .. [1] Moshen Bayati, Jeong Han Kim, and Amin Saberi,
(7)                    A sequential algorithm for generating random graphs.
(7)                    Algorithmica, Volume 58, Number 4, 860-910,
(7)                    DOI: 10.1007/s00453-009-9340-1
(4)                 Examples
(4)                 --------
(4)                 >>> sequence = [1, 2, 2, 3]
(4)                 >>> G = nx.random_degree_sequence_graph(sequence, seed=42)
(4)                 >>> sorted(d for n, d in G.degree())
(4)                 [1, 2, 2, 3]
(4)                 """
(4)                 DSRG = DegreeSequenceRandomGraph(sequence, seed)
(4)                 for try_n in range(tries):
(8)                     try:
(12)                        return DSRG.generate()
(8)                     except nx.NetworkXUnfeasible:
(12)                        pass
(4)                 raise nx.NetworkXError(f"failed to generate graph in {tries} tries")
(0)             class DegreeSequenceRandomGraph:
(4)                 def __init__(self, degree, rng):
(8)                     if not nx.is_graphical(degree):
(12)                        raise nx.NetworkXUnfeasible("degree sequence is not graphical")
(8)                     self.rng = rng
(8)                     self.degree = list(degree)
(8)                     self.m = sum(self.degree) / 2.0  # number of edges
(8)                     try:
(12)                        self.dmax = max(self.degree)  # maximum degree
(8)                     except ValueError:
(12)                        self.dmax = 0
(4)                 def generate(self):
(8)                     self.remaining_degree = dict(enumerate(self.degree))
(8)                     self.graph = nx.Graph()
(8)                     self.graph.add_nodes_from(self.remaining_degree)
(8)                     for n, d in list(self.remaining_degree.items()):
(12)                        if d == 0:
(16)                            del self.remaining_degree[n]
(8)                     if len(self.remaining_degree) > 0:
(12)                        self.phase1()
(12)                        self.phase2()
(12)                        self.phase3()
(8)                     return self.graph
(4)                 def update_remaining(self, u, v, aux_graph=None):
(8)                     if aux_graph is not None:
(12)                        aux_graph.remove_edge(u, v)
(8)                     if self.remaining_degree[u] == 1:
(12)                        del self.remaining_degree[u]
(12)                        if aux_graph is not None:
(16)                            aux_graph.remove_node(u)
(8)                     else:
(12)                        self.remaining_degree[u] -= 1
(8)                     if self.remaining_degree[v] == 1:
(12)                        del self.remaining_degree[v]
(12)                        if aux_graph is not None:
(16)                            aux_graph.remove_node(v)
(8)                     else:
(12)                        self.remaining_degree[v] -= 1
(4)                 def p(self, u, v):
(8)                     return 1 - self.degree[u] * self.degree[v] / (4.0 * self.m)
(4)                 def q(self, u, v):
```

```
626: (8)                    norm = max(self.remaining_degree.values()) ** 2
627: (8)                    return self.remaining_degree[u] * self.remaining_degree[v] / norm
628: (4)                def suitable_edge(self):
629: (8)                    """Returns True if and only if an arbitrary remaining node can
630: (8)                    potentially be joined with some other remaining node.
631: (8)                    """
632: (8)                    nodes = iter(self.remaining_degree)
633: (8)                    u = next(nodes)
634: (8)                    return any(v not in self.graph[u] for v in nodes)
635: (4)                def phase1(self):
636: (8)                    rem_deg = self.remaining_degree
637: (8)                    while sum(rem_deg.values()) >= 2 * self.dmax**2:
638: (12)                       u, v = sorted(random_weighted_sample(rem_deg, 2, self.rng))
639: (12)                       if self.graph.has_edge(u, v):
640: (16)                           continue
641: (12)                       if self.rng.random() < self.p(u, v):  # accept edge
642: (16)                           self.graph.add_edge(u, v)
643: (16)                           self.update_remaining(u, v)
644: (4)                def phase2(self):
645: (8)                    remaining_deg = self.remaining_degree
646: (8)                    rng = self.rng
647: (8)                    while len(remaining_deg) >= 2 * self.dmax:
648: (12)                       while True:
649: (16)                           u, v = sorted(rng.sample(list(remaining_deg.keys()), 2))
650: (16)                           if self.graph.has_edge(u, v):
651: (20)                               continue
652: (16)                           if rng.random() < self.q(u, v):
653: (20)                               break
654: (12)                       if rng.random() < self.p(u, v):  # accept edge
655: (16)                           self.graph.add_edge(u, v)
656: (16)                           self.update_remaining(u, v)
657: (4)                def phase3(self):
658: (8)                    potential_edges = combinations(self.remaining_degree, 2)
659: (8)                    H = nx.Graph(
660: (12)                       [(u, v) for (u, v) in potential_edges if not
self.graph.has_edge(u, v)]
661: (8)                    )
662: (8)                    rng = self.rng
663: (8)                    while self.remaining_degree:
664: (12)                       if not self.suitable_edge():
665: (16)                           raise nx.NetworkXUnfeasible("no suitable edges left")
666: (12)                       while True:
667: (16)                           u, v = sorted(rng.choice(list(H.edges())))
668: (16)                           if rng.random() < self.q(u, v):
669: (20)                               break
670: (12)                       if rng.random() < self.p(u, v):  # accept edge
671: (16)                           self.graph.add_edge(u, v)
672: (16)                           self.update_remaining(u, v, aux_graph=H)
```

File 432 - directed.py:

```
1: (0)              """
2: (0)              Generators for some directed graphs, including growing network (GN) graphs and
3: (0)              scale-free graphs.
4: (0)              """
5: (0)              import numbers
6: (0)              from collections import Counter
7: (0)              import networkx as nx
8: (0)              from networkx.generators.classic import empty_graph
9: (0)              from networkx.utils import discrete_sequence, py_random_state, weighted_choice
10: (0)             __all__ = [
11: (4)                 "gn_graph",
12: (4)                 "gnc_graph",
13: (4)                 "gnr_graph",
14: (4)                 "random_k_out_graph",
15: (4)                 "scale_free_graph",
16: (0)             ]
```

```
 17: (0)             @py_random_state(3)
 18: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 19: (0)             def gn_graph(n, kernel=None, create_using=None, seed=None):
 20: (4)                 """Returns the growing network (GN) digraph with `n` nodes.
 21: (4)                 The GN graph is built by adding nodes one at a time with a link to one
 22: (4)                 previously added node.  The target node for the link is chosen with
 23: (4)                 probability based on degree.  The default attachment kernel is a linear
 24: (4)                 function of the degree of a node.
 25: (4)                 The graph is always a (directed) tree.
 26: (4)                 Parameters
 27: (4)                 ----------
 28: (4)                 n : int
 29: (8)                     The number of nodes for the generated graph.
 30: (4)                 kernel : function
 31: (8)                     The attachment kernel.
 32: (4)                 create_using : NetworkX graph constructor, optional (default DiGraph)
 33: (8)                     Graph type to create. If graph instance, then cleared before
populated.
 34: (4)                 seed : integer, random_state, or None (default)
 35: (8)                     Indicator of random number generation state.
 36: (8)                     See :ref:`Randomness<randomness>`.
 37: (4)                 Examples
 38: (4)                 --------
 39: (4)                 To create the undirected GN graph, use the :meth:`~DiGraph.to_directed`
 40: (4)                 method::
 41: (4)                 >>> D = nx.gn_graph(10)  # the GN graph
 42: (4)                 >>> G = D.to_undirected()  # the undirected version
 43: (4)                 To specify an attachment kernel, use the `kernel` keyword argument::
 44: (4)                 >>> D = nx.gn_graph(10, kernel=lambda x: x**1.5)  # A_k = k^1.5
 45: (4)                 References
 46: (4)                 ----------
 47: (4)                 .. [1] P. L. Krapivsky and S. Redner,
 48: (11)                       Organization of Growing Random Networks,
 49: (11)                       Phys. Rev. E, 63, 066123, 2001.
 50: (4)                 """
 51: (4)                 G = empty_graph(1, create_using, default=nx.DiGraph)
 52: (4)                 if not G.is_directed():
 53: (8)                     raise nx.NetworkXError("create_using must indicate a Directed Graph")
 54: (4)                 if kernel is None:
 55: (8)                     def kernel(x):
 56: (12)                        return x
 57: (4)                 if n == 1:
 58: (8)                     return G
 59: (4)                 G.add_edge(1, 0)  # get started
 60: (4)                 ds = [1, 1]  # degree sequence
 61: (4)                 for source in range(2, n):
 62: (8)                     dist = [kernel(d) for d in ds]
 63: (8)                     target = discrete_sequence(1, distribution=dist, seed=seed)[0]
 64: (8)                     G.add_edge(source, target)
 65: (8)                     ds.append(1)  # the source has only one link (degree one)
 66: (8)                     ds[target] += 1  # add one to the target link degree
 67: (4)                 return G
 68: (0)             @py_random_state(3)
 69: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 70: (0)             def gnr_graph(n, p, create_using=None, seed=None):
 71: (4)                 """Returns the growing network with redirection (GNR) digraph with `n`
 72: (4)                 nodes and redirection probability `p`.
 73: (4)                 The GNR graph is built by adding nodes one at a time with a link to one
 74: (4)                 previously added node.  The previous target node is chosen uniformly at
 75: (4)                 random.  With probability `p` the link is instead "redirected" to the
 76: (4)                 successor node of the target.
 77: (4)                 The graph is always a (directed) tree.
 78: (4)                 Parameters
 79: (4)                 ----------
 80: (4)                 n : int
 81: (8)                     The number of nodes for the generated graph.
 82: (4)                 p : float
 83: (8)                     The redirection probability.
 84: (4)                 create_using : NetworkX graph constructor, optional (default DiGraph)
```

```
85: (8)                     Graph type to create. If graph instance, then cleared before
populated.
86: (4)                 seed : integer, random_state, or None (default)
87: (8)                     Indicator of random number generation state.
88: (8)                     See :ref:`Randomness<randomness>`.
89: (4)                 Examples
90: (4)                 --------
91: (4)                 To create the undirected GNR graph, use the :meth:`~DiGraph.to_directed`
92: (4)                 method::
93: (4)                 >>> D = nx.gnr_graph(10, 0.5)  # the GNR graph
94: (4)                 >>> G = D.to_undirected()  # the undirected version
95: (4)                 References
96: (4)                 ----------
97: (4)                 .. [1] P. L. Krapivsky and S. Redner,
98: (11)                       Organization of Growing Random Networks,
99: (11)                       Phys. Rev. E, 63, 066123, 2001.
100: (4)                """
101: (4)                G = empty_graph(1, create_using, default=nx.DiGraph)
102: (4)                if not G.is_directed():
103: (8)                    raise nx.NetworkXError("create_using must indicate a Directed Graph")
104: (4)                if n == 1:
105: (8)                    return G
106: (4)                for source in range(1, n):
107: (8)                    target = seed.randrange(0, source)
108: (8)                    if seed.random() < p and target != 0:
109: (12)                       target = next(G.successors(target))
110: (8)                    G.add_edge(source, target)
111: (4)                return G
112: (0)            @py_random_state(2)
113: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
114: (0)            def gnc_graph(n, create_using=None, seed=None):
115: (4)                """Returns the growing network with copying (GNC) digraph with `n` nodes.
116: (4)                The GNC graph is built by adding nodes one at a time with a link to one
117: (4)                previously added node (chosen uniformly at random) and to all of that
118: (4)                node's successors.
119: (4)                Parameters
120: (4)                ----------
121: (4)                n : int
122: (8)                    The number of nodes for the generated graph.
123: (4)                create_using : NetworkX graph constructor, optional (default DiGraph)
124: (8)                    Graph type to create. If graph instance, then cleared before
populated.
125: (4)                seed : integer, random_state, or None (default)
126: (8)                    Indicator of random number generation state.
127: (8)                    See :ref:`Randomness<randomness>`.
128: (4)                References
129: (4)                ----------
130: (4)                .. [1] P. L. Krapivsky and S. Redner,
131: (11)                      Network Growth by Copying,
132: (11)                      Phys. Rev. E, 71, 036118, 2005k.},
133: (4)                """
134: (4)                G = empty_graph(1, create_using, default=nx.DiGraph)
135: (4)                if not G.is_directed():
136: (8)                    raise nx.NetworkXError("create_using must indicate a Directed Graph")
137: (4)                if n == 1:
138: (8)                    return G
139: (4)                for source in range(1, n):
140: (8)                    target = seed.randrange(0, source)
141: (8)                    for succ in G.successors(target):
142: (12)                       G.add_edge(source, succ)
143: (8)                    G.add_edge(source, target)
144: (4)                return G
145: (0)            @py_random_state(6)
146: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
147: (0)            def scale_free_graph(
148: (4)                n,
149: (4)                alpha=0.41,
150: (4)                beta=0.54,
151: (4)                gamma=0.05,
```

```
(4)                 delta_in=0.2,
(4)                 delta_out=0,
(4)                 seed=None,
(4)                 initial_graph=None,
(0)             ):
(4)                 """Returns a scale-free directed graph.
(4)                 Parameters
(4)                 ----------
(4)                 n : integer
(8)                     Number of nodes in graph
(4)                 alpha : float
(8)                     Probability for adding a new node connected to an existing node
(8)                     chosen randomly according to the in-degree distribution.
(4)                 beta : float
(8)                     Probability for adding an edge between two existing nodes.
(8)                     One existing node is chosen randomly according the in-degree
(8)                     distribution and the other chosen randomly according to the out-degree
(8)                     distribution.
(4)                 gamma : float
(8)                     Probability for adding a new node connected to an existing node
(8)                     chosen randomly according to the out-degree distribution.
(4)                 delta_in : float
(8)                     Bias for choosing nodes from in-degree distribution.
(4)                 delta_out : float
(8)                     Bias for choosing nodes from out-degree distribution.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 initial_graph : MultiDiGraph instance, optional
(8)                     Build the scale-free graph starting from this initial MultiDiGraph,
(8)                     if provided.
(4)                 Returns
(4)                 -------
(4)                 MultiDiGraph
(4)                 Examples
(4)                 --------
(4)                 Create a scale-free graph on one hundred nodes::
(4)                 >>> G = nx.scale_free_graph(100)
(4)                 Notes
(4)                 -----
(4)                 The sum of `alpha`, `beta`, and `gamma` must be 1.
(4)                 References
(4)                 ----------
(4)                 .. [1] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan,
(11)                       Directed scale-free graphs,
(11)                       Proceedings of the fourteenth annual ACM-SIAM Symposium on
(11)                       Discrete Algorithms, 132--139, 2003.
(4)                 """
(4)                 def _choose_node(candidates, node_list, delta):
(8)                     if delta > 0:
(12)                        bias_sum = len(node_list) * delta
(12)                        p_delta = bias_sum / (bias_sum + len(candidates))
(12)                        if seed.random() < p_delta:
(16)                            return seed.choice(node_list)
(8)                     return seed.choice(candidates)
(4)                 if initial_graph is not None and hasattr(initial_graph, "_adj"):
(8)                     if not isinstance(initial_graph, nx.MultiDiGraph):
(12)                        raise nx.NetworkXError("initial_graph must be a MultiDiGraph.")
(8)                     G = initial_graph
(4)                 else:
(8)                     G = nx.MultiDiGraph([(0, 1), (1, 2), (2, 0)])
(4)                 if alpha <= 0:
(8)                     raise ValueError("alpha must be > 0.")
(4)                 if beta <= 0:
(8)                     raise ValueError("beta must be > 0.")
(4)                 if gamma <= 0:
(8)                     raise ValueError("gamma must be > 0.")
(4)                 if abs(alpha + beta + gamma - 1.0) >= 1e-9:
(8)                     raise ValueError("alpha+beta+gamma must equal 1.")
```

```
221: (4)                if delta_in < 0:
222: (8)                    raise ValueError("delta_in must be >= 0.")
223: (4)                if delta_out < 0:
224: (8)                    raise ValueError("delta_out must be >= 0.")
225: (4)                vs = sum((count * [idx] for idx, count in G.out_degree()), [])
226: (4)                ws = sum((count * [idx] for idx, count in G.in_degree()), [])
227: (4)                node_list = list(G.nodes())
228: (4)                numeric_nodes = [n for n in node_list if isinstance(n, numbers.Number)]
229: (4)                if len(numeric_nodes) > 0:
230: (8)                    cursor = max(int(n.real) for n in numeric_nodes) + 1
231: (4)                else:
232: (8)                    cursor = 0
233: (4)                while len(G) < n:
234: (8)                    r = seed.random()
235: (8)                    if r < alpha:
236: (12)                       v = cursor
237: (12)                       cursor += 1
238: (12)                       node_list.append(v)
239: (12)                       w = _choose_node(ws, node_list, delta_in)
240: (8)                    elif r < alpha + beta:
241: (12)                       v = _choose_node(vs, node_list, delta_out)
242: (12)                       w = _choose_node(ws, node_list, delta_in)
243: (8)                    else:
244: (12)                       v = _choose_node(vs, node_list, delta_out)
245: (12)                       w = cursor
246: (12)                       cursor += 1
247: (12)                       node_list.append(w)
248: (8)                    G.add_edge(v, w)
249: (8)                    vs.append(v)
250: (8)                    ws.append(w)
251: (4)                return G
252: (0)            @py_random_state(4)
253: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
254: (0)            def random_uniform_k_out_graph(n, k, self_loops=True, with_replacement=True,
seed=None):
255: (4)                """Returns a random `k`-out graph with uniform attachment.
256: (4)                A random `k`-out graph with uniform attachment is a multidigraph
257: (4)                generated by the following algorithm. For each node *u*, choose
258: (4)                `k` nodes *v* uniformly at random (with replacement). Add a
259: (4)                directed edge joining *u* to *v*.
260: (4)                Parameters
261: (4)                ----------
262: (4)                n : int
263: (8)                    The number of nodes in the returned graph.
264: (4)                k : int
265: (8)                    The out-degree of each node in the returned graph.
266: (4)                self_loops : bool
267: (8)                    If True, self-loops are allowed when generating the graph.
268: (4)                with_replacement : bool
269: (8)                    If True, neighbors are chosen with replacement and the
270: (8)                    returned graph will be a directed multigraph. Otherwise,
271: (8)                    neighbors are chosen without replacement and the returned graph
272: (8)                    will be a directed graph.
273: (4)                seed : integer, random_state, or None (default)
274: (8)                    Indicator of random number generation state.
275: (8)                    See :ref:`Randomness<randomness>`.
276: (4)                Returns
277: (4)                -------
278: (4)                NetworkX graph
279: (8)                    A `k`-out-regular directed graph generated according to the
280: (8)                    above algorithm. It will be a multigraph if and only if
281: (8)                    `with_replacement` is True.
282: (4)                Raises
283: (4)                ------
284: (4)                ValueError
285: (8)                    If `with_replacement` is False and `k` is greater than
286: (8)                    `n`.
287: (4)                See also
288: (4)                --------
```

```
(4)                 random_k_out_graph
(4)                 Notes
(4)                 -----
(4)                 The return digraph or multidigraph may not be strongly connected, or
(4)                 even weakly connected.
(4)                 If `with_replacement` is True, this function is similar to
(4)                 :func:`random_k_out_graph`, if that function had parameter `alpha`
(4)                 set to positive infinity.
(4)                 """
(4)                 if with_replacement:
(8)                     create_using = nx.MultiDiGraph()
(8)                     def sample(v, nodes):
(12)                        if not self_loops:
(16)                            nodes = nodes - {v}
(12)                        return (seed.choice(list(nodes)) for i in range(k))
(4)                 else:
(8)                     create_using = nx.DiGraph()
(8)                     def sample(v, nodes):
(12)                        if not self_loops:
(16)                            nodes = nodes - {v}
(12)                        return seed.sample(list(nodes), k)
(4)                 G = nx.empty_graph(n, create_using)
(4)                 nodes = set(G)
(4)                 for u in G:
(8)                     G.add_edges_from((u, v) for v in sample(u, nodes))
(4)                 return G
(0)             @py_random_state(4)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_k_out_graph(n, k, alpha, self_loops=True, seed=None):
(4)                 """Returns a random `k`-out graph with preferential attachment.
(4)                 A random `k`-out graph with preferential attachment is a
(4)                 multidigraph generated by the following algorithm.
(4)                 1. Begin with an empty digraph, and initially set each node to have
(7)                    weight `alpha`.
(4)                 2. Choose a node `u` with out-degree less than `k` uniformly at
(7)                    random.
(4)                 3. Choose a node `v` from with probability proportional to its
(7)                    weight.
(4)                 4. Add a directed edge from `u` to `v`, and increase the weight
(7)                    of `v` by one.
(4)                 5. If each node has out-degree `k`, halt, otherwise repeat from
(7)                    step 2.
(4)                 For more information on this model of random graph, see [1].
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes in the returned graph.
(4)                 k : int
(8)                     The out-degree of each node in the returned graph.
(4)                 alpha : float
(8)                     A positive :class:`float` representing the initial weight of
(8)                     each vertex. A higher number means that in step 3 above, nodes
(8)                     will be chosen more like a true uniformly random sample, and a
(8)                     lower number means that nodes are more likely to be chosen as
(8)                     their in-degree increases. If this parameter is not positive, a
(8)                     :exc:`ValueError` is raised.
(4)                 self_loops : bool
(8)                     If True, self-loops are allowed when generating the graph.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 :class:`~networkx.classes.MultiDiGraph`
(8)                     A `k`-out-regular multidigraph generated according to the above
(8)                     algorithm.
(4)                 Raises
(4)                 ------
(4)                 ValueError
```

```
358: (8)                    If `alpha` is not positive.
359: (4)                Notes
360: (4)                -----
361: (4)                The returned multidigraph may not be strongly connected, or even
362: (4)                weakly connected.
363: (4)                References
364: (4)                ----------
365: (4)                [1]: Peterson, Nicholas R., and Boris Pittel.
366: (9)                     "Distance between two random `k`-out digraphs, with and without
367: (9)                     preferential attachment."
368: (9)                     arXiv preprint arXiv:1311.5961 (2013).
369: (9)                     <https://arxiv.org/abs/1311.5961>
370: (4)                """
371: (4)                if alpha < 0:
372: (8)                    raise ValueError("alpha must be positive")
373: (4)                G = nx.empty_graph(n, create_using=nx.MultiDiGraph)
374: (4)                weights = Counter({v: alpha for v in G})
375: (4)                for i in range(k * n):
376: (8)                    u = seed.choice([v for v, d in G.out_degree() if d < k])
377: (8)                    if not self_loops:
378: (12)                       adjustment = Counter({u: weights[u]})
379: (8)                    else:
380: (12)                       adjustment = Counter()
381: (8)                    v = weighted_choice(weights - adjustment, seed=seed)
382: (8)                    G.add_edge(u, v)
383: (8)                    weights[v] += 1
384: (4)                return G
```

File 433 - duplication.py:

```
1: (0)              """Functions for generating graphs based on the "duplication" method.
2: (0)              These graph generators start with a small initial graph then duplicate
3: (0)              nodes and (partially) duplicate their edges. These functions are
4: (0)              generally inspired by biological networks.
5: (0)              """
6: (0)              import networkx as nx
7: (0)              from networkx.exception import NetworkXError
8: (0)              from networkx.utils import py_random_state
9: (0)              __all__ = ["partial_duplication_graph", "duplication_divergence_graph"]
10: (0)             @py_random_state(4)
11: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
12: (0)             def partial_duplication_graph(N, n, p, q, seed=None):
13: (4)                 """Returns a random graph using the partial duplication model.
14: (4)                 Parameters
15: (4)                 ----------
16: (4)                 N : int
17: (8)                     The total number of nodes in the final graph.
18: (4)                 n : int
19: (8)                     The number of nodes in the initial clique.
20: (4)                 p : float
21: (8)                     The probability of joining each neighbor of a node to the
22: (8)                     duplicate node. Must be a number in the between zero and one,
23: (8)                     inclusive.
24: (4)                 q : float
25: (8)                     The probability of joining the source node to the duplicate
26: (8)                     node. Must be a number in the between zero and one, inclusive.
27: (4)                 seed : integer, random_state, or None (default)
28: (8)                     Indicator of random number generation state.
29: (8)                     See :ref:`Randomness<randomness>`.
30: (4)                 Notes
31: (4)                 -----
32: (4)                 A graph of nodes is grown by creating a fully connected graph
33: (4)                 of size `n`. The following procedure is then repeated until
34: (4)                 a total of `N` nodes have been reached.
35: (4)                 1. A random node, *u*, is picked and a new node, *v*, is created.
36: (4)                 2. For each neighbor of *u* an edge from the neighbor to *v* is created
37: (7)                    with probability `p`.
```

```
(4)                 3. An edge from *u* to *v* is created with probability `q`.
(4)                 This algorithm appears in [1].
(4)                 This implementation allows the possibility of generating
(4)                 disconnected graphs.
(4)                 References
(4)                 ----------
(4)                 .. [1] Knudsen Michael, and Carsten Wiuf. "A Markov chain approach to
(11)                       randomly grown graphs." Journal of Applied Mathematics 2008.
(11)                       <https://doi.org/10.1155/2008/190836>
(4)                 """
(4)                 if p < 0 or p > 1 or q < 0 or q > 1:
(8)                     msg = "partial duplication graph must have 0 <= p, q <= 1."
(8)                     raise NetworkXError(msg)
(4)                 if n > N:
(8)                     raise NetworkXError("partial duplication graph must have n <= N.")
(4)                 G = nx.complete_graph(n)
(4)                 for new_node in range(n, N):
(8)                     src_node = seed.randint(0, new_node - 1)
(8)                     G.add_node(new_node)
(8)                     for nbr_node in list(nx.all_neighbors(G, src_node)):
(12)                        if seed.random() < p:
(16)                            G.add_edge(new_node, nbr_node)
(8)                     if seed.random() < q:
(12)                        G.add_edge(new_node, src_node)
(4)                 return G
(0)             @py_random_state(2)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def duplication_divergence_graph(n, p, seed=None):
(4)                 """Returns an undirected graph using the duplication-divergence model.
(4)                 A graph of `n` nodes is created by duplicating the initial nodes
(4)                 and retaining edges incident to the original nodes with a retention
(4)                 probability `p`.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The desired number of nodes in the graph.
(4)                 p : float
(8)                     The probability for retaining the edge of the replicated node.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 G : Graph
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If `p` is not a valid probability.
(8)                     If `n` is less than 2.
(4)                 Notes
(4)                 -----
(4)                 This algorithm appears in [1].
(4)                 This implementation disallows the possibility of generating
(4)                 disconnected graphs.
(4)                 References
(4)                 ----------
(4)                 .. [1] I. Ispolatov, P. L. Krapivsky, A. Yuryev,
(7)                    "Duplication-divergence model of protein interaction network",
(7)                    Phys. Rev. E, 71, 061911, 2005.
(4)                 """
(4)                 if p > 1 or p < 0:
(8)                     msg = f"NetworkXError p={p} is not in [0,1]."
(8)                     raise nx.NetworkXError(msg)
(4)                 if n < 2:
(8)                     msg = "n must be greater than or equal to 2"
(8)                     raise nx.NetworkXError(msg)
(4)                 G = nx.Graph()
(4)                 G.add_edge(0, 1)
(4)                 i = 2
```

```
107: (4)                while i < n:
108: (8)                    random_node = seed.choice(list(G))
109: (8)                    G.add_node(i)
110: (8)                    flag = False
111: (8)                    for nbr in G.neighbors(random_node):
112: (12)                       if seed.random() < p:
113: (16)                           G.add_edge(i, nbr)
114: (16)                           flag = True
115: (8)                    if not flag:
116: (12)                       G.remove_node(i)
117: (8)                    else:
118: (12)                       i += 1
119: (4)                return G
```

File 434 - ego.py:

```
1: (0)              """
2: (0)              Ego graph.
3: (0)              """
4: (0)              __all__ = ["ego_graph"]
5: (0)              import networkx as nx
6: (0)              @nx._dispatchable(preserve_all_attrs=True, returns_graph=True)
7: (0)              def ego_graph(G, n, radius=1, center=True, undirected=False, distance=None):
8: (4)                  """Returns induced subgraph of neighbors centered at node n within
9: (4)                  a given radius.
10: (4)                 Parameters
11: (4)                 ----------
12: (4)                 G : graph
13: (6)                   A NetworkX Graph or DiGraph
14: (4)                 n : node
15: (6)                   A single node
16: (4)                 radius : number, optional
17: (6)                   Include all neighbors of distance<=radius from n.
18: (4)                 center : bool, optional
19: (6)                   If False, do not include center node in graph
20: (4)                 undirected : bool, optional
21: (6)                   If True use both in- and out-neighbors of directed graphs.
22: (4)                 distance : key, optional
23: (6)                   Use specified edge data key as distance.  For example, setting
24: (6)                   distance='weight' will use the edge weight to measure the
25: (6)                   distance from the node n.
26: (4)                 Notes
27: (4)                 -----
28: (4)                 For directed graphs D this produces the "out" neighborhood
29: (4)                 or successors.  If you want the neighborhood of predecessors
30: (4)                 first reverse the graph with D.reverse().  If you want both
31: (4)                 directions use the keyword argument undirected=True.
32: (4)                 Node, edge, and graph attributes are copied to the returned subgraph.
33: (4)                 """
34: (4)                 if undirected:
35: (8)                     if distance is not None:
36: (12)                        sp, _ = nx.single_source_dijkstra(
37: (16)                            G.to_undirected(), n, cutoff=radius, weight=distance
38: (12)                        )
39: (8)                     else:
40: (12)                        sp = dict(
41: (16)                            nx.single_source_shortest_path_length(
42: (20)                                G.to_undirected(), n, cutoff=radius
43: (16)                            )
44: (12)                        )
45: (4)                 else:
46: (8)                     if distance is not None:
47: (12)                        sp, _ = nx.single_source_dijkstra(G, n, cutoff=radius,
weight=distance)
48: (8)                     else:
49: (12)                        sp = dict(nx.single_source_shortest_path_length(G, n,
cutoff=radius))
```

```
50: (4)                 H = G.subgraph(sp).copy()
51: (4)                 if not center:
52: (8)                     H.remove_node(n)
53: (4)                 return H
```

File 435 - expanders.py:

```
1: (0)              """Provides explicit constructions of expander graphs.
2: (0)              """
3: (0)              import itertools
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "margulis_gabber_galil_graph",
7: (4)                  "chordal_cycle_graph",
8: (4)                  "paley_graph",
9: (4)                  "maybe_regular_expander",
10: (4)                 "is_regular_expander",
11: (4)                 "random_regular_expander_graph",
12: (0)             ]
13: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
14: (0)             def margulis_gabber_galil_graph(n, create_using=None):
15: (4)                 r"""Returns the Margulis-Gabber-Galil undirected MultiGraph on `n^2`
nodes.
16: (4)                 The undirected MultiGraph is regular with degree `8`. Nodes are integer
17: (4)                 pairs. The second-largest eigenvalue of the adjacency matrix of the graph
18: (4)                 is at most `5 \sqrt{2}`, regardless of `n`.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 n : int
22: (8)                     Determines the number of nodes in the graph: `n^2`.
23: (4)                 create_using : NetworkX graph constructor, optional (default MultiGraph)
24: (7)                    Graph type to create. If graph instance, then cleared before populated.
25: (4)                 Returns
26: (4)                 -------
27: (4)                 G : graph
28: (8)                     The constructed undirected multigraph.
29: (4)                 Raises
30: (4)                 ------
31: (4)                 NetworkXError
32: (8)                     If the graph is directed or not a multigraph.
33: (4)                 """
34: (4)                 G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
35: (4)                 if G.is_directed() or not G.is_multigraph():
36: (8)                     msg = "`create_using` must be an undirected multigraph."
37: (8)                     raise nx.NetworkXError(msg)
38: (4)                 for x, y in itertools.product(range(n), repeat=2):
39: (8)                     for u, v in (
40: (12)                        ((x + 2 * y) % n, y),
41: (12)                        ((x + (2 * y + 1)) % n, y),
42: (12)                        (x, (y + 2 * x) % n),
43: (12)                        (x, (y + (2 * x + 1)) % n),
44: (8)                     ):
45: (12)                        G.add_edge((x, y), (u, v))
46: (4)                 G.graph["name"] = f"margulis_gabber_galil_graph({n})"
47: (4)                 return G
48: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
49: (0)             def chordal_cycle_graph(p, create_using=None):
50: (4)                 """Returns the chordal cycle graph on `p` nodes.
51: (4)                 The returned graph is a cycle graph on `p` nodes with chords joining each
52: (4)                 vertex `x` to its inverse modulo `p`. This graph is a (mildly explicit)
53: (4)                 3-regular expander [1]_.
54: (4)                 `p` *must* be a prime number.
55: (4)                 Parameters
56: (4)                 ----------
57: (4)                 p : a prime number
58: (8)                     The number of vertices in the graph. This also indicates where the
59: (8)                     chordal edges in the cycle will be created.
```

```
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : graph
        The constructed undirected multigraph.
    Raises
    ------
    NetworkXError
        If `create_using` indicates directed or not a multigraph.
    References
    ----------
    .. [1] Theorem 4.4.2 in A. Lubotzky. "Discrete groups, expanding graphs and
           invariant measures", volume 125 of Progress in Mathematics.
           Birkhäuser Verlag, Basel, 1994.
    """
    G = nx.empty_graph(0, create_using, default=nx.MultiGraph)
    if G.is_directed() or not G.is_multigraph():
        msg = "`create_using` must be an undirected multigraph."
        raise nx.NetworkXError(msg)
    for x in range(p):
        left = (x - 1) % p
        right = (x + 1) % p
        chord = pow(x, p - 2, p) if x > 0 else 0
        for y in (left, right, chord):
            G.add_edge(x, y)
    G.graph["name"] = f"chordal_cycle_graph({p})"
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def paley_graph(p, create_using=None):
    r"""Returns the Paley $\frac{(p-1)}{2}$ -regular graph on $p$ nodes.
    The returned graph is a graph on $\mathbb{Z}/p\mathbb{Z}$ with edges between $x$ and $y$
    if and only if $x-y$ is a nonzero square in $\mathbb{Z}/p\mathbb{Z}$.
    If $p \equiv 1  \pmod 4$, $-1$ is a square in $\mathbb{Z}/p\mathbb{Z}$ and therefore $x-y$ is a square if and
    only if $y-x$ is also a square, i.e the edges in the Paley graph are symmetric.
    If $p \equiv 3 \pmod 4$, $-1$ is not a square in $\mathbb{Z}/p\mathbb{Z}$ and therefore either $x-y$ or $y-x$
    is a square in $\mathbb{Z}/p\mathbb{Z}$ but not both.
    Note that a more general definition of Paley graphs extends this construction
    to graphs over $q=p^n$ vertices, by using the finite field $F_q$ instead of $\mathbb{Z}/p\mathbb{Z}$.
    This construction requires to compute squares in general finite fields and is
    not what is implemented here (i.e `paley_graph(25)` does not return the true
    Paley graph associated with $5^2$).
    Parameters
    ----------
    p : int, an odd prime number.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : graph
        The constructed directed graph.
    Raises
    ------
    NetworkXError
        If the graph is a multigraph.
    References
    ----------
    Chapter 13 in B. Bollobas, Random Graphs. Second edition.
    Cambridge Studies in Advanced Mathematics, 73.
    Cambridge University Press, Cambridge (2001).
```

```
    """
    G = nx.empty_graph(0, create_using, default=nx.DiGraph)
    if G.is_multigraph():
        msg = "`create_using` cannot be a multigraph."
        raise nx.NetworkXError(msg)
    square_set = {(x**2) % p for x in range(1, p) if (x**2) % p != 0}
    for x in range(p):
        for x2 in square_set:
            G.add_edge(x, (x + x2) % p)
    G.graph["name"] = f"paley({p})"
    return G
@nx.utils.decorators.np_random_state("seed")
@nx._dispatchable(graphs=None, returns_graph=True)
def maybe_regular_expander(n, d, *, create_using=None, max_tries=100, seed=None):
    r"""Utility for creating a random regular expander.
    Returns a random $d$-regular graph on $n$ nodes which is an expander
    graph with very good probability.
    Parameters
    ----------
    n : int
      The number of nodes.
    d : int
      The degree of each node.
    create_using : Graph Instance or Constructor
      Indicator of type of graph to return.
      If a Graph-type instance, then clear and use it.
      If a constructor, call it to create an empty graph.
      Use the Graph constructor by default.
    max_tries : int. (default: 100)
      The number of allowed loops when generating each independent cycle
    seed : (default: None)
      Seed used to set random number generation state. See :ref`Randomness<randomness>`.
    Notes
    -----
    The nodes are numbered from $0$ to $n - 1$.
    The graph is generated by taking $d / 2$ random independent cycles.
    Joel Friedman proved that in this model the resulting
    graph is an expander with probability
    $1 - O(n^{-\tau})$ where $\tau = \lceil (\sqrt{d - 1}) / 2 \rceil - 1$. [1]_
    Examples
    --------
    >>> G = nx.maybe_regular_expander(n=200, d=6, seed=8020)
    Returns
    -------
    G : graph
        The constructed undirected graph.
    Raises
    ------
    NetworkXError
        If $d % 2 != 0$ as the degree must be even.
        If $n - 1$ is less than $ 2d $ as the graph is complete at most.
        If max_tries is reached
    See Also
    --------
    is_regular_expander
    random_regular_expander_graph
    References
    ----------
    .. [1] Joel Friedman,
       A Proof of Alon’s Second Eigenvalue Conjecture and Related Problems, 2004
       https://arxiv.org/abs/cs/0405020
    """
    import numpy as np
    if n < 1:
        raise nx.NetworkXError("n must be a positive integer")
```

```
185: (4)                if not (d >= 2):
186: (8)                    raise nx.NetworkXError("d must be greater than or equal to 2")
187: (4)                if not (d % 2 == 0):
188: (8)                    raise nx.NetworkXError("d must be even")
189: (4)                if not (n - 1 >= d):
190: (8)                    raise nx.NetworkXError(
191: (12)                       f"Need n-1>= d to have room for {d//2} independent cycles with {n}
nodes"
192: (8)                    )
193: (4)                G = nx.empty_graph(n, create_using)
194: (4)                if n < 2:
195: (8)                    return G
196: (4)                cycles = []
197: (4)                edges = set()
198: (4)                for i in range(d // 2):
199: (8)                    iterations = max_tries
200: (8)                    while len(edges) != (i + 1) * n:
201: (12)                       iterations -= 1
202: (12)                       cycle = seed.permutation(n - 1).tolist()
203: (12)                       cycle.append(n - 1)
204: (12)                       new_edges = {
205: (16)                           (u, v)
206: (16)                           for u, v in nx.utils.pairwise(cycle, cyclic=True)
207: (16)                           if (u, v) not in edges and (v, u) not in edges
208: (12)                       }
209: (12)                       if len(new_edges) == n:
210: (16)                           cycles.append(cycle)
211: (16)                           edges.update(new_edges)
212: (12)                       if iterations == 0:
213: (16)                           raise nx.NetworkXError("Too many iterations in
maybe_regular_expander")
214: (4)                G.add_edges_from(edges)
215: (4)                return G
216: (0)            @nx.utils.not_implemented_for("directed")
217: (0)            @nx.utils.not_implemented_for("multigraph")
218: (0)            @nx._dispatchable(preserve_edge_attrs={"G": {"weight": 1}})
219: (0)            def is_regular_expander(G, *, epsilon=0):
220: (4)                r"""Determines whether the graph G is a regular expander. [1]_
221: (4)                An expander graph is a sparse graph with strong connectivity properties.
222: (4)                More precisely, this helper checks whether the graph is a
223: (4)                regular $(n, d, \lambda)$-expander with $\lambda$ close to
224: (4)                the Alon-Boppana bound and given by
225: (4)                $\lambda = 2 \sqrt{d - 1} + \epsilon$. [2]_
226: (4)                In the case where $\epsilon = 0$ then if the graph successfully passes the
test
227: (4)                it is a Ramanujan graph. [3]_
228: (4)                A Ramanujan graph has spectral gap almost as large as possible, which
makes them
229: (4)                excellent expanders.
230: (4)                Parameters
231: (4)                ----------
232: (4)                G : NetworkX graph
233: (4)                epsilon : int, float, default=0
234: (4)                Returns
235: (4)                -------
236: (4)                bool
237: (8)                    Whether the given graph is a regular $(n, d, \lambda)$-expander
238: (8)                    where $\lambda = 2 \sqrt{d - 1} + \epsilon$.
239: (4)                Examples
240: (4)                --------
241: (4)                >>> G = nx.random_regular_expander_graph(20, 4)
242: (4)                >>> nx.is_regular_expander(G)
243: (4)                True
244: (4)                See Also
245: (4)                --------
246: (4)                maybe_regular_expander
247: (4)                random_regular_expander_graph
248: (4)                References
249: (4)                ----------
```

```
250: (4)                .. [1] Expander graph, https://en.wikipedia.org/wiki/Expander_graph
251: (4)                .. [2] Alon-Boppana bound,
https://en.wikipedia.org/wiki/Alon%E2%80%93Boppana_bound
252: (4)                .. [3] Ramanujan graphs, https://en.wikipedia.org/wiki/Ramanujan_graph
253: (4)                """
254: (4)                import numpy as np
255: (4)                from scipy.sparse.linalg import eigsh
256: (4)                if epsilon < 0:
257: (8)                    raise nx.NetworkXError("epsilon must be non negative")
258: (4)                if not nx.is_regular(G):
259: (8)                    return False
260: (4)                _, d = nx.utils.arbitrary_element(G.degree)
261: (4)                A = nx.adjacency_matrix(G, dtype=float)
262: (4)                lams = eigsh(A, which="LM", k=2, return_eigenvectors=False)
263: (4)                lambda2 = min(lams)
264: (4)                return bool(abs(lambda2) < 2 ** np.sqrt(d - 1) + epsilon)
265: (0)            @nx.utils.decorators.np_random_state("seed")
266: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
267: (0)            def random_regular_expander_graph(
268: (4)                n, d, *, epsilon=0, create_using=None, max_tries=100, seed=None
269: (0)            ):
270: (4)                r"""Returns a random regular expander graph on $n$ nodes with degree $d$.
271: (4)                An expander graph is a sparse graph with strong connectivity properties.
[1]_
272: (4)                More precisely the returned graph is a $(n, d, \lambda)$-expander with
273: (4)                $\lambda = 2 \sqrt{d - 1} + \epsilon$, close to the Alon-Boppana bound.
[2]_
274: (4)                In the case where $\epsilon = 0$ it returns a Ramanujan graph.
275: (4)                A Ramanujan graph has spectral gap almost as large as possible,
276: (4)                which makes them excellent expanders. [3]_
277: (4)                Parameters
278: (4)                ----------
279: (4)                n : int
280: (6)                  The number of nodes.
281: (4)                d : int
282: (6)                  The degree of each node.
283: (4)                epsilon : int, float, default=0
284: (4)                max_tries : int, (default: 100)
285: (6)                  The number of allowed loops, also used in the maybe_regular_expander
utility
286: (4)                seed : (default: None)
287: (6)                  Seed used to set random number generation state. See
:ref`Randomness<randomness>`.
288: (4)                Raises
289: (4)                ------
290: (4)                NetworkXError
291: (8)                    If max_tries is reached
292: (4)                Examples
293: (4)                --------
294: (4)                >>> G = nx.random_regular_expander_graph(20, 4)
295: (4)                >>> nx.is_regular_expander(G)
296: (4)                True
297: (4)                Notes
298: (4)                -----
299: (4)                This loops over `maybe_regular_expander` and can be slow when
300: (4)                $n$ is too big or $\epsilon$ too small.
301: (4)                See Also
302: (4)                --------
303: (4)                maybe_regular_expander
304: (4)                is_regular_expander
305: (4)                References
306: (4)                ----------
307: (4)                .. [1] Expander graph, https://en.wikipedia.org/wiki/Expander_graph
308: (4)                .. [2] Alon-Boppana bound,
https://en.wikipedia.org/wiki/Alon%E2%80%93Boppana_bound
309: (4)                .. [3] Ramanujan graphs, https://en.wikipedia.org/wiki/Ramanujan_graph
310: (4)                """
311: (4)                G = maybe_regular_expander(
312: (8)                    n, d, create_using=create_using, max_tries=max_tries, seed=seed
```

```
313: (4)                )
314: (4)                iterations = max_tries
315: (4)                while not is_regular_expander(G, epsilon=epsilon):
316: (8)                    iterations -= 1
317: (8)                    G = maybe_regular_expander(
318: (12)                       n=n, d=d, create_using=create_using, max_tries=max_tries,
seed=seed
319: (8)                    )
320: (8)                    if iterations == 0:
321: (12)                       raise nx.NetworkXError(
322: (16)                           "Too many iterations in random_regular_expander_graph"
323: (12)                       )
324: (4)                return G
```

File 436 - geometric.py:

```
1: (0)              """Generators for geometric graphs.
2: (0)              """
3: (0)              import math
4: (0)              from bisect import bisect_left
5: (0)              from itertools import accumulate, combinations, product
6: (0)              import networkx as nx
7: (0)              from networkx.utils import py_random_state
8: (0)              __all__ = [
9: (4)                  "geometric_edges",
10: (4)                 "geographical_threshold_graph",
11: (4)                 "navigable_small_world_graph",
12: (4)                 "random_geometric_graph",
13: (4)                 "soft_random_geometric_graph",
14: (4)                 "thresholded_random_geometric_graph",
15: (4)                 "waxman_graph",
16: (4)                 "geometric_soft_configuration_graph",
17: (0)             ]
18: (0)             @nx._dispatchable(node_attrs="pos_name")
19: (0)             def geometric_edges(G, radius, p=2, *, pos_name="pos"):
20: (4)                 """Returns edge list of node pairs within `radius` of each other.
21: (4)                 Parameters
22: (4)                 ----------
23: (4)                 G : networkx graph
24: (8)                     The graph from which to generate the edge list. The nodes in `G`
should
25: (8)                     have an attribute ``pos`` corresponding to the node position, which is
26: (8)                     used to compute the distance to other nodes.
27: (4)                 radius : scalar
28: (8)                     The distance threshold. Edges are included in the edge list if the
29: (8)                     distance between the two nodes is less than `radius`.
30: (4)                 pos_name : string, default="pos"
31: (8)                     The name of the node attribute which represents the position of each
32: (8)                     node in 2D coordinates. Every node in the Graph must have this
attribute.
33: (4)                 p : scalar, default=2
34: (8)                     The `Minkowski distance metric
35: (8)                     <https://en.wikipedia.org/wiki/Minkowski_distance>`_ used to compute
36: (8)                     distances. The default value is 2, i.e. Euclidean distance.
37: (4)                 Returns
38: (4)                 -------
39: (4)                 edges : list
40: (8)                     List of edges whose distances are less than `radius`
41: (4)                 Notes
42: (4)                 -----
43: (4)                 Radius uses Minkowski distance metric `p`.
44: (4)                 If scipy is available, `scipy.spatial.cKDTree` is used to speed
computation.
45: (4)                 Examples
46: (4)                 --------
47: (4)                 Create a graph with nodes that have a "pos" attribute representing 2D
48: (4)                 coordinates.
```

```
49: (4)                 >>> G = nx.Graph()
50: (4)                 >>> G.add_nodes_from(
51: (4)                 ...     [
52: (4)                 ...         (0, {"pos": (0, 0)}),
53: (4)                 ...         (1, {"pos": (3, 0)}),
54: (4)                 ...         (2, {"pos": (8, 0)}),
55: (4)                 ...     ]
56: (4)                 ... )
57: (4)                 >>> nx.geometric_edges(G, radius=1)
58: (4)                 []
59: (4)                 >>> nx.geometric_edges(G, radius=4)
60: (4)                 [(0, 1)]
61: (4)                 >>> nx.geometric_edges(G, radius=6)
62: (4)                 [(0, 1), (1, 2)]
63: (4)                 >>> nx.geometric_edges(G, radius=9)
64: (4)                 [(0, 1), (0, 2), (1, 2)]
65: (4)                 """
66: (4)                 for n, pos in G.nodes(data=pos_name):
67: (8)                     if pos is None:
68: (12)                        raise nx.NetworkXError(
69: (16)                            f"Node {n} (and all nodes) must have a '{pos_name}'
attribute."
70: (12)                        )
71: (4)                 return _geometric_edges(G, radius, p, pos_name)
72: (0)             def _geometric_edges(G, radius, p, pos_name):
73: (4)                 """
74: (4)                 Implements `geometric_edges` without input validation. See
`geometric_edges`
75: (4)                 for complete docstring.
76: (4)                 """
77: (4)                 nodes_pos = G.nodes(data=pos_name)
78: (4)                 try:
79: (8)                     import scipy as sp
80: (4)                 except ImportError:
81: (8)                     radius_p = radius**p
82: (8)                     edges = [
83: (12)                        (u, v)
84: (12)                        for (u, pu), (v, pv) in combinations(nodes_pos, 2)
85: (12)                        if sum(abs(a - b) ** p for a, b in zip(pu, pv)) <= radius_p
86: (8)                     ]
87: (8)                     return edges
88: (4)                 nodes, coords = list(zip(*nodes_pos))
89: (4)                 kdtree = sp.spatial.cKDTree(coords)  # Cannot provide generator.
90: (4)                 edge_indexes = kdtree.query_pairs(radius, p)
91: (4)                 edges = [(nodes[u], nodes[v]) for u, v in sorted(edge_indexes)]
92: (4)                 return edges
93: (0)             @py_random_state(5)
94: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
95: (0)             def random_geometric_graph(
96: (4)                 n, radius, dim=2, pos=None, p=2, seed=None, *, pos_name="pos"
97: (0)             ):
98: (4)                 """Returns a random geometric graph in the unit cube of dimensions `dim`.
99: (4)                 The random geometric graph model places `n` nodes uniformly at
100: (4)                random in the unit cube. Two nodes are joined by an edge if the
101: (4)                distance between the nodes is at most `radius`.
102: (4)                Edges are determined using a KDTree when SciPy is available.
103: (4)                This reduces the time complexity from $O(n^2)$ to $O(n)$.
104: (4)                Parameters
105: (4)                ----------
106: (4)                n : int or iterable
107: (8)                    Number of nodes or iterable of nodes
108: (4)                radius: float
109: (8)                    Distance threshold value
110: (4)                dim : int, optional
111: (8)                    Dimension of graph
112: (4)                pos : dict, optional
113: (8)                    A dictionary keyed by node with node positions as values.
114: (4)                p : float, optional
115: (8)                    Which Minkowski distance metric to use.  `p` has to meet the condition
```

```
116: (8)                    ``1 <= p <= infinity``.
117: (8)                    If this argument is not specified, the :math:`L^2` metric
118: (8)                    (the Euclidean distance metric), p = 2 is used.
119: (8)                    This should not be confused with the `p` of an Erdős-Rényi random
120: (8)                    graph, which represents probability.
121: (4)                seed : integer, random_state, or None (default)
122: (8)                    Indicator of random number generation state.
123: (8)                    See :ref:`Randomness<randomness>`.
124: (4)                pos_name : string, default="pos"
125: (8)                    The name of the node attribute which represents the position
126: (8)                    in 2D coordinates of the node in the returned graph.
127: (4)                Returns
128: (4)                -------
129: (4)                Graph
130: (8)                    A random geometric graph, undirected and without self-loops.
131: (8)                    Each node has a node attribute ``'pos'`` that stores the
132: (8)                    position of that node in Euclidean space as provided by the
133: (8)                    ``pos`` keyword argument or, if ``pos`` was not provided, as
134: (8)                    generated by this function.
135: (4)                Examples
136: (4)                --------
137: (4)                Create a random geometric graph on twenty nodes where nodes are joined by
138: (4)                an edge if their distance is at most 0.1::
139: (4)                >>> G = nx.random_geometric_graph(20, 0.1)
140: (4)                Notes
141: (4)                -----
142: (4)                This uses a *k*-d tree to build the graph.
143: (4)                The `pos` keyword argument can be used to specify node positions so you
144: (4)                can create an arbitrary distribution and domain for positions.
145: (4)                For example, to use a 2D Gaussian distribution of node positions with mean
146: (4)                (0, 0) and standard deviation 2::
147: (4)                >>> import random
148: (4)                >>> n = 20
149: (4)                >>> pos = {i: (random.gauss(0, 2), random.gauss(0, 2)) for i in range(n)}
150: (4)                >>> G = nx.random_geometric_graph(n, 0.2, pos=pos)
151: (4)                References
152: (4)                ----------
153: (4)                .. [1] Penrose, Mathew, *Random Geometric Graphs*,
154: (11)                      Oxford Studies in Probability, 5, 2003.
155: (4)                """
156: (4)                G = nx.empty_graph(n)
157: (4)                if pos is None:
158: (8)                    pos = {v: [seed.random() for i in range(dim)] for v in G}
159: (4)                nx.set_node_attributes(G, pos, pos_name)
160: (4)                G.add_edges_from(_geometric_edges(G, radius, p, pos_name))
161: (4)                return G
162: (0)            @py_random_state(6)
163: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
164: (0)            def soft_random_geometric_graph(
165: (4)                n, radius, dim=2, pos=None, p=2, p_dist=None, seed=None, *, pos_name="pos"
166: (0)            ):
167: (4)                r"""Returns a soft random geometric graph in the unit cube.
168: (4)                The soft random geometric graph [1] model places `n` nodes uniformly at
169: (4)                random in the unit cube in dimension `dim`. Two nodes of distance, `dist`,
170: (4)                computed by the `p`-Minkowski distance metric are joined by an edge with
171: (4)                probability `p_dist` if the computed distance metric value of the nodes
172: (4)                is at most `radius`, otherwise they are not joined.
173: (4)                Edges within `radius` of each other are determined using a KDTree when
174: (4)                SciPy is available. This reduces the time complexity from :math:`O(n^2)`
175: (4)                to :math:`O(n)`.
176: (4)                Parameters
177: (4)                ----------
178: (4)                n : int or iterable
179: (8)                    Number of nodes or iterable of nodes
180: (4)                radius: float
181: (8)                    Distance threshold value
182: (4)                dim : int, optional
183: (8)                    Dimension of graph
184: (4)                pos : dict, optional
```

```
        A dictionary keyed by node with node positions as values.
    p : float, optional
        Which Minkowski distance metric to use.
        `p` has to meet the condition ``1 <= p <= infinity``.
        If this argument is not specified, the :math:`L^2` metric
        (the Euclidean distance metric), p = 2 is used.
        This should not be confused with the `p` of an Erdős-Rényi random
        graph, which represents probability.
    p_dist : function, optional
        A probability density function computing the probability of
        connecting two nodes that are of distance, dist, computed by the
        Minkowski distance metric. The probability density function, `p_dist`,
        must be any function that takes the metric value as input
        and outputs a single probability value between 0-1. The scipy.stats
        package has many probability distribution functions implemented and
        tools for custom probability distribution definitions [2], and passing
        the .pdf method of scipy.stats distributions can be used here.  If the
        probability function, `p_dist`, is not supplied, the default function
        is an exponential distribution with rate parameter :math:`\lambda=1`.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    pos_name : string, default="pos"
        The name of the node attribute which represents the position
        in 2D coordinates of the node in the returned graph.
    Returns
    -------
    Graph
        A soft random geometric graph, undirected and without self-loops.
        Each node has a node attribute ``'pos'`` that stores the
        position of that node in Euclidean space as provided by the
        ``pos`` keyword argument or, if ``pos`` was not provided, as
        generated by this function.
    Examples
    --------
    Default Graph:
    G = nx.soft_random_geometric_graph(50, 0.2)
    Custom Graph:
    Create a soft random geometric graph on 100 uniformly distributed nodes
    where nodes are joined by an edge with probability computed from an
    exponential distribution with rate parameter :math:`\lambda=1` if their
    Euclidean distance is at most 0.2.
    Notes
    -----
    This uses a *k*-d tree to build the graph.
    The `pos` keyword argument can be used to specify node positions so you
    can create an arbitrary distribution and domain for positions.
    For example, to use a 2D Gaussian distribution of node positions with mean
    (0, 0) and standard deviation 2
    The scipy.stats package can be used to define the probability distribution
    with the .pdf method used as `p_dist`.
    ::
    >>> import random
    >>> import math
    >>> n = 100
    >>> pos = {i: (random.gauss(0, 2), random.gauss(0, 2)) for i in range(n)}
    >>> p_dist = lambda dist: math.exp(-dist)
    >>> G = nx.soft_random_geometric_graph(n, 0.2, pos=pos, p_dist=p_dist)
    References
    ----------
    .. [1] Penrose, Mathew D. "Connectivity of soft random geometric graphs."
           The Annals of Applied Probability 26.2 (2016): 986-1028.
    .. [2] scipy.stats -
           https://docs.scipy.org/doc/scipy/reference/tutorial/stats.html
    """
    G = nx.empty_graph(n)
    G.name = f"soft_random_geometric_graph({n}, {radius}, {dim})"
    if pos is None:
        pos = {v: [seed.random() for i in range(dim)] for v in G}
```

```
(4)                 nx.set_node_attributes(G, pos, pos_name)
(4)                 if p_dist is None:
(8)                     def p_dist(dist):
(12)                        return math.exp(-dist)
(4)                 def should_join(edge):
(8)                     u, v = edge
(8)                     dist = (sum(abs(a - b) ** p for a, b in zip(pos[u], pos[v]))) ** (1 /
p)
(8)                     return seed.random() < p_dist(dist)
(4)                 G.add_edges_from(filter(should_join, _geometric_edges(G, radius, p,
pos_name)))
(4)                 return G
(0)             @py_random_state(7)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def geographical_threshold_graph(
(4)                 n,
(4)                 theta,
(4)                 dim=2,
(4)                 pos=None,
(4)                 weight=None,
(4)                 metric=None,
(4)                 p_dist=None,
(4)                 seed=None,
(4)                 *,
(4)                 pos_name="pos",
(4)                 weight_name="weight",
(0)             ):
(4)                 r"""Returns a geographical threshold graph.
(4)                 The geographical threshold graph model places $n$ nodes uniformly at
(4)                 random in a rectangular domain.  Each node $u$ is assigned a weight
(4)                 $w_u$. Two nodes $u$ and $v$ are joined by an edge if
(4)                 .. math::
(7)                    (w_u + w_v)p_{dist}(r) \ge \theta
(4)                 where `r` is the distance between `u` and `v`, `p_dist` is any function of
(4)                 `r`, and :math:`\theta` as the threshold parameter. `p_dist` is used to
(4)                 give weight to the distance between nodes when deciding whether or not
(4)                 they should be connected. The larger `p_dist` is, the more prone nodes
(4)                 separated by `r` are to be connected, and vice versa.
(4)                 Parameters
(4)                 ----------
(4)                 n : int or iterable
(8)                     Number of nodes or iterable of nodes
(4)                 theta: float
(8)                     Threshold value
(4)                 dim : int, optional
(8)                     Dimension of graph
(4)                 pos : dict
(8)                     Node positions as a dictionary of tuples keyed by node.
(4)                 weight : dict
(8)                     Node weights as a dictionary of numbers keyed by node.
(4)                 metric : function
(8)                     A metric on vectors of numbers (represented as lists or
(8)                     tuples). This must be a function that accepts two lists (or
(8)                     tuples) as input and yields a number as output. The function
(8)                     must also satisfy the four requirements of a `metric`_.
(8)                     Specifically, if $d$ is the function and $x$, $y$,
(8)                     and $z$ are vectors in the graph, then $d$ must satisfy
(8)                     1. $d(x, y) \ge 0$,
(8)                     2. $d(x, y) = 0$ if and only if $x = y$,
(8)                     3. $d(x, y) = d(y, x)$,
(8)                     4. $d(x, z) \le d(x, y) + d(y, z)$.
(8)                     If this argument is not specified, the Euclidean distance metric is
(8)                     used.
(8)                     .. _metric: https://en.wikipedia.org/wiki/Metric_%28mathematics%29
(4)                 p_dist : function, optional
(8)                     Any function used to give weight to the distance between nodes when
(8)                     deciding whether or not they should be connected. `p_dist` was
(8)                     originally conceived as a probability density function giving the
(8)                     probability of connecting two nodes that are of metric distance `r`
```

```
            apart. The implementation here allows for more arbitrary definitions
            of `p_dist` that do not need to correspond to valid probability
            density functions. The :mod:`scipy.stats` package has many
            probability density functions implemented and tools for custom
            probability density definitions, and passing the ``.pdf`` method of
            scipy.stats distributions can be used here. If ``p_dist=None``
            (the default), the exponential function :math:`r^{-2}` is used.
        seed : integer, random_state, or None (default)
            Indicator of random number generation state.
            See :ref:`Randomness<randomness>`.
        pos_name : string, default="pos"
            The name of the node attribute which represents the position
            in 2D coordinates of the node in the returned graph.
        weight_name : string, default="weight"
            The name of the node attribute which represents the weight
            of the node in the returned graph.
        Returns
        -------
        Graph
            A random geographic threshold graph, undirected and without
            self-loops.
            Each node has a node attribute ``pos`` that stores the
            position of that node in Euclidean space as provided by the
            ``pos`` keyword argument or, if ``pos`` was not provided, as
            generated by this function. Similarly, each node has a node
            attribute ``weight`` that stores the weight of that node as
            provided or as generated.
        Examples
        --------
        Specify an alternate distance metric using the ``metric`` keyword
        argument. For example, to use the `taxicab metric`_ instead of the
        default `Euclidean metric`_::
            >>> dist = lambda x, y: sum(abs(a - b) for a, b in zip(x, y))
            >>> G = nx.geographical_threshold_graph(10, 0.1, metric=dist)
        .. _taxicab metric: https://en.wikipedia.org/wiki/Taxicab_geometry
        .. _Euclidean metric: https://en.wikipedia.org/wiki/Euclidean_distance
        Notes
        -----
        If weights are not specified they are assigned to nodes by drawing randomly
        from the exponential distribution with rate parameter $\lambda=1$.
        To specify weights from a different distribution, use the `weight` keyword
        argument::
        >>> import random
        >>> n = 20
        >>> w = {i: random.expovariate(5.0) for i in range(n)}
        >>> G = nx.geographical_threshold_graph(20, 50, weight=w)
        If node positions are not specified they are randomly assigned from the
        uniform distribution.
        References
        ----------
        .. [1] Masuda, N., Miwa, H., Konno, N.:
           Geographical threshold graphs with small-world and scale-free
           properties.
           Physical Review E 71, 036108 (2005)
        .. [2]  Milan Bradonjić, Aric Hagberg and Allon G. Percus,
           Giant component and connectivity in geographical threshold graphs,
           in Algorithms and Models for the Web-Graph (WAW 2007),
           Antony Bonato and Fan Chung (Eds), pp. 209--216, 2007
        """
        G = nx.empty_graph(n)
        if weight is None:
            weight = {v: seed.expovariate(1) for v in G}
        if pos is None:
            pos = {v: [seed.random() for i in range(dim)] for v in G}
        if metric is None:
            metric = math.dist
        nx.set_node_attributes(G, weight, weight_name)
        nx.set_node_attributes(G, pos, pos_name)
```

```
389: (4)                if p_dist is None:
390: (8)                    def p_dist(r):
391: (12)                       return r**-2
392: (4)                def should_join(pair):
393: (8)                    u, v = pair
394: (8)                    u_pos, v_pos = pos[u], pos[v]
395: (8)                    u_weight, v_weight = weight[u], weight[v]
396: (8)                    return (u_weight + v_weight) * p_dist(metric(u_pos, v_pos)) >= theta
397: (4)                G.add_edges_from(filter(should_join, combinations(G, 2)))
398: (4)                return G
399: (0)            @py_random_state(6)
400: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
401: (0)            def waxman_graph(
402: (4)                n,
403: (4)                beta=0.4,
404: (4)                alpha=0.1,
405: (4)                L=None,
406: (4)                domain=(0, 0, 1, 1),
407: (4)                metric=None,
408: (4)                seed=None,
409: (4)                *,
410: (4)                pos_name="pos",
411: (0)            ):
412: (4)                r"""Returns a Waxman random graph.
413: (4)                The Waxman random graph model places `n` nodes uniformly at random
414: (4)                in a rectangular domain. Each pair of nodes at distance `d` is
415: (4)                joined by an edge with probability
416: (4)                .. math::
417: (12)                       p = \beta \exp(-d / \alpha L).
418: (4)                This function implements both Waxman models, using the `L` keyword
419: (4)                argument.
420: (4)                * Waxman-1: if `L` is not specified, it is set to be the maximum distance
421: (6)                  between any pair of nodes.
422: (4)                * Waxman-2: if `L` is specified, the distance between a pair of nodes is
423: (6)                  chosen uniformly at random from the interval `[0, L]`.
424: (4)                Parameters
425: (4)                ----------
426: (4)                n : int or iterable
427: (8)                    Number of nodes or iterable of nodes
428: (4)                beta: float
429: (8)                    Model parameter
430: (4)                alpha: float
431: (8)                    Model parameter
432: (4)                L : float, optional
433: (8)                    Maximum distance between nodes.  If not specified, the actual distance
434: (8)                    is calculated.
435: (4)                domain : four-tuple of numbers, optional
436: (8)                    Domain size, given as a tuple of the form `(x_min, y_min, x_max,
437: (8)                    y_max)`.
438: (4)                metric : function
439: (8)                    A metric on vectors of numbers (represented as lists or
440: (8)                    tuples). This must be a function that accepts two lists (or
441: (8)                    tuples) as input and yields a number as output. The function
442: (8)                    must also satisfy the four requirements of a `metric`_.
443: (8)                    Specifically, if $d$ is the function and $x$, $y$,
444: (8)                    and $z$ are vectors in the graph, then $d$ must satisfy
445: (8)                    1. $d(x, y) \ge 0$,
446: (8)                    2. $d(x, y) = 0$ if and only if $x = y$,
447: (8)                    3. $d(x, y) = d(y, x)$,
448: (8)                    4. $d(x, z) \le d(x, y) + d(y, z)$.
449: (8)                    If this argument is not specified, the Euclidean distance metric is
450: (8)                    used.
451: (8)                    .. _metric: https://en.wikipedia.org/wiki/Metric_%28mathematics%29
452: (4)                seed : integer, random_state, or None (default)
453: (8)                    Indicator of random number generation state.
454: (8)                    See :ref:`Randomness<randomness>`.
455: (4)                pos_name : string, default="pos"
456: (8)                    The name of the node attribute which represents the position
457: (8)                    in 2D coordinates of the node in the returned graph.
```

```
    Returns
    -------
    Graph
        A random Waxman graph, undirected and without self-loops. Each
        node has a node attribute ``'pos'`` that stores the position of
        that node in Euclidean space as generated by this function.
    Examples
    --------
    Specify an alternate distance metric using the ``metric`` keyword
    argument. For example, to use the "`taxicab metric`_" instead of the
    default `Euclidean metric`_::
        >>> dist = lambda x, y: sum(abs(a - b) for a, b in zip(x, y))
        >>> G = nx.waxman_graph(10, 0.5, 0.1, metric=dist)
    .. _taxicab metric: https://en.wikipedia.org/wiki/Taxicab_geometry
    .. _Euclidean metric: https://en.wikipedia.org/wiki/Euclidean_distance
    Notes
    -----
    Starting in NetworkX 2.0 the parameters alpha and beta align with their
    usual roles in the probability distribution. In earlier versions their
    positions in the expression were reversed. Their position in the calling
    sequence reversed as well to minimize backward incompatibility.
    References
    ----------
    .. [1]  B. M. Waxman, *Routing of multipoint connections*.
       IEEE J. Select. Areas Commun. 6(9),(1988) 1617--1622.
    """
    G = nx.empty_graph(n)
    (xmin, ymin, xmax, ymax) = domain
    pos = {v: (seed.uniform(xmin, xmax), seed.uniform(ymin, ymax)) for v in G}
    nx.set_node_attributes(G, pos, pos_name)
    if metric is None:
        metric = math.dist
    if L is None:
        L = max(metric(x, y) for x, y in combinations(pos.values(), 2))
        def dist(u, v):
            return metric(pos[u], pos[v])
    else:
        def dist(u, v):
            return seed.random() * L
    def should_join(pair):
        return seed.random() < beta * math.exp(-dist(*pair) / (alpha * L))
    G.add_edges_from(filter(should_join, combinations(G, 2)))
    return G
@py_random_state(5)
@nx._dispatchable(graphs=None, returns_graph=True)
def navigable_small_world_graph(n, p=1, q=1, r=2, dim=2, seed=None):
    r"""Returns a navigable small-world graph.
    A navigable small-world graph is a directed grid with additional long-range

    connections that are chosen randomly.
      [...] we begin with a set of nodes [...] that are identified with the set

      of lattice points in an $n \times n$ square,
      $\{(i, j): i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, n\}\}$,
      and we define the *lattice distance* between two nodes $(i, j)$ and
      $(k, l)$ to be the number of "lattice steps" separating them:
      $d((i, j), (k, l)) = |k - i| + |l - j|$.
      For a universal constant $p >= 1$, the node $u$ has a directed edge to
      every other node within lattice distance $p$---these are its *local
      contacts*. For universal constants $q >= 0$ and $r >= 0$ we also
      construct directed edges from $u$ to $q$ other nodes (the *long-range
      contacts*) using independent random trials; the $i$th directed edge from
      $u$ has endpoint $v$ with probability proportional to $[d(u,v)]^{-r}$.
      -- [1]_
    Parameters
    ----------
    n : int
        The length of one side of the lattice; the number of nodes in
        the graph is therefore $n^2$.
```

```
(4)                 p : int
(8)                     The diameter of short range connections. Each node is joined with
every
(8)                     other node within this lattice distance.
(4)                 q : int
(8)                     The number of long-range connections for each node.
(4)                 r : float
(8)                     Exponent for decaying probability of connections.  The probability of
(8)                     connecting to a node at lattice distance $d$ is $1/d^r$.
(4)                 dim : int
(8)                     Dimension of grid
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 References
(4)                 ----------
(4)                 .. [1] J. Kleinberg. The small-world phenomenon: An algorithmic
(7)                    perspective. Proc. 32nd ACM Symposium on Theory of Computing, 2000.
(4)                 """
(4)                 if p < 1:
(8)                     raise nx.NetworkXException("p must be >= 1")
(4)                 if q < 0:
(8)                     raise nx.NetworkXException("q must be >= 0")
(4)                 if r < 0:
(8)                     raise nx.NetworkXException("r must be >= 0")
(4)                 G = nx.DiGraph()
(4)                 nodes = list(product(range(n), repeat=dim))
(4)                 for p1 in nodes:
(8)                     probs = [0]
(8)                     for p2 in nodes:
(12)                        if p1 == p2:
(16)                            continue
(12)                        d = sum((abs(b - a) for a, b in zip(p1, p2)))
(12)                        if d <= p:
(16)                            G.add_edge(p1, p2)
(12)                        probs.append(d**-r)
(8)                     cdf = list(accumulate(probs))
(8)                     for _ in range(q):
(12)                        target = nodes[bisect_left(cdf, seed.uniform(0, cdf[-1]))]
(12)                        G.add_edge(p1, target)
(4)                 return G
(0)             @py_random_state(7)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def thresholded_random_geometric_graph(
(4)                 n,
(4)                 radius,
(4)                 theta,
(4)                 dim=2,
(4)                 pos=None,
(4)                 weight=None,
(4)                 p=2,
(4)                 seed=None,
(4)                 *,
(4)                 pos_name="pos",
(4)                 weight_name="weight",
(0)             ):
(4)                 r"""Returns a thresholded random geometric graph in the unit cube.
(4)                 The thresholded random geometric graph [1] model places `n` nodes
(4)                 uniformly at random in the unit cube of dimensions `dim`. Each node
(4)                 `u` is assigned a weight :math:`w_u`. Two nodes `u` and `v` are
(4)                 joined by an edge if they are within the maximum connection distance,
(4)                 `radius` computed by the `p`-Minkowski distance and the summation of
(4)                 weights :math:`w_u` + :math:`w_v` is greater than or equal
(4)                 to the threshold parameter `theta`.
(4)                 Edges within `radius` of each other are determined using a KDTree when
(4)                 SciPy is available. This reduces the time complexity from :math:`O(n^2)`
(4)                 to :math:`O(n)`.
(4)                 Parameters
(4)                 ----------
```

```
    n : int or iterable
        Number of nodes or iterable of nodes
    radius: float
        Distance threshold value
    theta: float
        Threshold value
    dim : int, optional
        Dimension of graph
    pos : dict, optional
        A dictionary keyed by node with node positions as values.
    weight : dict, optional
        Node weights as a dictionary of numbers keyed by node.
    p : float, optional (default 2)
        Which Minkowski distance metric to use.  `p` has to meet the condition
        ``1 <= p <= infinity``.
        If this argument is not specified, the :math:`L^2` metric
        (the Euclidean distance metric), p = 2 is used.
        This should not be confused with the `p` of an Erdős-Rényi random
        graph, which represents probability.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    pos_name : string, default="pos"
        The name of the node attribute which represents the position
        in 2D coordinates of the node in the returned graph.
    weight_name : string, default="weight"
        The name of the node attribute which represents the weight
        of the node in the returned graph.
    Returns
    -------
    Graph
        A thresholded random geographic graph, undirected and without
        self-loops.
        Each node has a node attribute ``'pos'`` that stores the
        position of that node in Euclidean space as provided by the
        ``pos`` keyword argument or, if ``pos`` was not provided, as
        generated by this function. Similarly, each node has a nodethre
        attribute ``'weight'`` that stores the weight of that node as
        provided or as generated.
    Examples
    --------
    Default Graph:
    G = nx.thresholded_random_geometric_graph(50, 0.2, 0.1)
    Custom Graph:
    Create a thresholded random geometric graph on 50 uniformly distributed
    nodes where nodes are joined by an edge if their sum weights drawn from
    a exponential distribution with rate = 5 are >= theta = 0.1 and their
    Euclidean distance is at most 0.2.
    Notes
    -----
    This uses a *k*-d tree to build the graph.
    The `pos` keyword argument can be used to specify node positions so you
    can create an arbitrary distribution and domain for positions.
    For example, to use a 2D Gaussian distribution of node positions with mean
    (0, 0) and standard deviation 2
    If weights are not specified they are assigned to nodes by drawing
randomly
    from the exponential distribution with rate parameter :math:`\lambda=1`.
    To specify weights from a different distribution, use the `weight` keyword
    argument::
    ::
    >>> import random
    >>> import math
    >>> n = 50
    >>> pos = {i: (random.gauss(0, 2), random.gauss(0, 2)) for i in range(n)}
    >>> w = {i: random.expovariate(5.0) for i in range(n)}
    >>> G = nx.thresholded_random_geometric_graph(n, 0.2, 0.1, 2, pos, w)
    References
    ----------
```

```
    .. [1] http://cole-maclean.github.io/blog/files/thesis.pdf
    """
    G = nx.empty_graph(n)
    G.name = f"thresholded_random_geometric_graph({n}, {radius}, {theta}, {dim})"
    if weight is None:
        weight = {v: seed.expovariate(1) for v in G}
    if pos is None:
        pos = {v: [seed.random() for i in range(dim)] for v in G}
    nx.set_node_attributes(G, weight, weight_name)
    nx.set_node_attributes(G, pos, pos_name)
    edges = (
        (u, v)
        for u, v in _geometric_edges(G, radius, p, pos_name)
        if weight[u] + weight[v] >= theta
    )
    G.add_edges_from(edges)
    return G
@py_random_state(5)
@nx._dispatchable(graphs=None, returns_graph=True)
def geometric_soft_configuration_graph(
    *, beta, n=None, gamma=None, mean_degree=None, kappas=None, seed=None
):
    r"""Returns a random graph from the geometric soft configuration model.
    The $\mathbb{S}^1$ model [1]_ is the geometric soft configuration model
    which is able to explain many fundamental features of real networks such as
    small-world property, heteregenous degree distributions, high level of
    clustering, and self-similarity.
    In the geometric soft configuration model, a node $i$ is assigned two hidden
    variables: a hidden degree $\kappa_i$, quantifying its popularity, influence,
    or importance, and an angular position $\theta_i$ in a circle abstracting the
    similarity space, where angular distances between nodes are a proxy for their
    similarity. Focusing on the angular position, this model is often called
    the $\mathbb{S}^1$ model (a one-dimensional sphere). The circle's radius is
    adjusted to $R = N/2\pi$, where $N$ is the number of nodes, so that the density
    is set to 1 without loss of generality.
    The connection probability between any pair of nodes increases with
    the product of their hidden degrees (i.e., their combined popularities),
    and decreases with the angular distance between the two nodes.
    Specifically, nodes $i$ and $j$ are connected with the probability
    $p_{ij} = \frac{1}{1 + \frac{d_{ij}^\beta}{\left(\mu \kappa_i \kappa_j\right)^{\max(1, \beta)}}}$
    where $d_{ij} = R\Delta\theta_{ij}$ is the arc length of the circle between
    nodes $i$ and $j$ separated by an angular distance $\Delta\theta_{ij}$.
    Parameters $\mu$ and $\beta$ (also called inverse temperature) control the
    average degree and the clustering coefficient, respectively.
    It can be shown [2]_ that the model undergoes a structural phase transition
    at $\beta=1$ so that for $\beta<1$ networks are unclustered in the thermodynamic
    limit (when $N\to \infty$) whereas for $\beta>1$ the ensemble generates
    networks with finite clustering coefficient.
    The $\mathbb{S}^1$ model can be expressed as a purely geometric model
    $\mathbb{H}^2$ in the hyperbolic plane [3]_ by mapping the hidden degree of
    each node into a radial coordinate as
    $r_i = \hat{R} - \frac{2 \max(1, \beta)}{\beta \zeta} \ln \left(\frac{\kappa_i}{\kappa_0}\right)$
    where $\hat{R}$ is the radius of the hyperbolic disk and $\zeta$ is the curvature,
    $\hat{R} = \frac{2}{\zeta} \ln \left(\frac{N}{\pi}\right)
```

```
715: (4)                - \frac{2\max(1, \beta)}{\beta \zeta} \ln (\mu \kappa_0^2)$
716: (4)                The connection probability then reads
717: (4)                $p_{ij} = \frac{1}{1 + \exp\left({\frac{\beta\zeta}{2} (x_{ij} -
\hat{R})}\right)}$
718: (4)                where
719: (4)                $x_{ij} = r_i + r_j + \frac{2}{\zeta} \ln \frac{\Delta\theta_{ij}}{2}$
720: (4)                is a good approximation of the hyperbolic distance between two nodes
separated
721: (4)                by an angular distance $\Delta\theta_{ij}$ with radial coordinates $r_i$
and $r_j$.
722: (4)                For $\beta > 1$, the curvature $\zeta = 1$, for $\beta < 1$, $\zeta =
\beta^{-1}$.
723: (4)                Parameters
724: (4)                ----------
725: (4)                Either `n`, `gamma`, `mean_degree` are provided or `kappas`. The values of
726: (4)                `n`, `gamma`, `mean_degree` (if provided) are used to construct a random
727: (4)                kappa-dict keyed by node with values sampled from a power-law
distribution.
728: (4)                beta : positive number
729: (8)                    Inverse temperature, controlling the clustering coefficient.
730: (4)                n : int (default: None)
731: (8)                    Size of the network (number of nodes).
732: (8)                    If not provided, `kappas` must be provided and holds the nodes.
733: (4)                gamma : float (default: None)
734: (8)                    Exponent of the power-law distribution for hidden degrees `kappas`.
735: (8)                    If not provided, `kappas` must be provided directly.
736: (4)                mean_degree : float (default: None)
737: (8)                    The mean degree in the network.
738: (8)                    If not provided, `kappas` must be provided directly.
739: (4)                kappas : dict (default: None)
740: (8)                    A dict keyed by node to its hidden degree value.
741: (8)                    If not provided, random values are computed based on a power-law
742: (8)                    distribution using `n`, `gamma` and `mean_degree`.
743: (4)                seed : int, random_state, or None (default)
744: (8)                    Indicator of random number generation state.
745: (8)                    See :ref:`Randomness<randomness>`.
746: (4)                Returns
747: (4)                -------
748: (4)                Graph
749: (8)                    A random geometric soft configuration graph (undirected with no self-
loops).
750: (8)                    Each node has three node-attributes:
751: (8)                    - ``kappa`` that represents the hidden degree.
752: (8)                    - ``theta`` the position in the similarity space ($\mathbb{S}^1$)
which is
753: (10)                     also the angular position in the hyperbolic plane.
754: (8)                    - ``radius`` the radial position in the hyperbolic plane
755: (10)                     (based on the hidden degree).
756: (4)                Examples
757: (4)                --------
758: (4)                Generate a network with specified parameters:
759: (4)                >>> G = nx.geometric_soft_configuration_graph(beta=1.5, n=100, gamma=2.7,
mean_degree=5)
760: (4)                Create a geometric soft configuration graph with 100 nodes. The $\beta$
parameter
761: (4)                is set to 1.5 and the exponent of the powerlaw distribution of the hidden
762: (4)                degrees is 2.7 with mean value of 5.
763: (4)                Generate a network with predefined hidden degrees:
764: (4)                >>> kappas = {i: 10 for i in range(100)}
765: (4)                >>> G = nx.geometric_soft_configuration_graph(beta=2.5, kappas=kappas)
766: (4)                Create a geometric soft configuration graph with 100 nodes. The $\beta$
parameter
767: (4)                is set to 2.5 and all nodes with hidden degree $\kappa=10$.
768: (4)                References
769: (4)                ----------
770: (4)                .. [1] Serrano, M. Á., Krioukov, D., & Boguñá, M. (2008). Self-similarity
771: (7)                   of complex networks and hidden metric spaces. Physical review letters,
100(7), 078701.
772: (4)                .. [2] van der Kolk, J., Serrano, M. Á., & Boguñá, M. (2022). An anomalous
```

```
773: (7)                    topological phase transition in spatial random graphs. Communications
Physics, 5(1), 245.
774: (4)                .. [3] Krioukov, D., Papadopoulos, F., Kitsak, M., Vahdat, A., & Boguná,
M. (2010).
775: (7)                    Hyperbolic geometry of complex networks. Physical Review E, 82(3),
036106.
776: (4)                """
777: (4)                if beta <= 0:
778: (8)                    raise nx.NetworkXError("The parameter beta cannot be smaller or equal
to 0.")
779: (4)                if kappas is not None:
780: (8)                    if not all((n is None, gamma is None, mean_degree is None)):
781: (12)                       raise nx.NetworkXError(
782: (16)                           "When kappas is input, n, gamma and mean_degree must not be."
783: (12)                       )
784: (8)                    n = len(kappas)
785: (8)                    mean_degree = sum(kappas) / len(kappas)
786: (4)                else:
787: (8)                    if any((n is None, gamma is None, mean_degree is None)):
788: (12)                       raise nx.NetworkXError(
789: (16)                           "Please provide either kappas, or all 3 of: n, gamma and
mean_degree."
790: (12)                       )
791: (8)                    gam_ratio = (gamma - 2) / (gamma - 1)
792: (8)                    kappa_0 = mean_degree * gam_ratio * (1 - 1 / n) / (1 - 1 /
n**gam_ratio)
793: (8)                    base = 1 - 1 / n
794: (8)                    power = 1 / (1 - gamma)
795: (8)                    kappas = {i: kappa_0 * (1 - seed.random() * base) ** power for i in
range(n)}
796: (4)                G = nx.Graph()
797: (4)                R = n / (2 * math.pi)
798: (4)                if beta > 1:
799: (8)                    mu = beta * math.sin(math.pi / beta) / (2 * math.pi * mean_degree)
800: (4)                elif beta == 1:
801: (8)                    mu = 1 / (2 * mean_degree * math.log(n))
802: (4)                else:
803: (8)                    mu = (1 - beta) / (2**beta * mean_degree * n ** (1 - beta))
804: (4)                thetas = {k: seed.uniform(0, 2 * math.pi) for k in kappas}
805: (4)                for u in kappas:
806: (8)                    for v in list(G):
807: (12)                       angle = math.pi - math.fabs(math.pi - math.fabs(thetas[u] -
thetas[v]))
808: (12)                       dij = math.pow(R * angle, beta)
809: (12)                       mu_kappas = math.pow(mu * kappas[u] * kappas[v], max(1, beta))
810: (12)                       p_ij = 1 / (1 + dij / mu_kappas)
811: (12)                       if seed.random() < p_ij:
812: (16)                           G.add_edge(u, v)
813: (8)                    G.add_node(u)
814: (4)                nx.set_node_attributes(G, thetas, "theta")
815: (4)                nx.set_node_attributes(G, kappas, "kappa")
816: (4)                zeta = 1 if beta > 1 else 1 / beta
817: (4)                kappa_min = min(kappas.values())
818: (4)                R_c = 2 * max(1, beta) / (beta * zeta)
819: (4)                R_hat = (2 / zeta) * math.log(n / math.pi) - R_c * math.log(mu *
kappa_min)
820: (4)                radii = {node: R_hat - R_c * math.log(kappa) for node, kappa in
kappas.items()}
821: (4)                nx.set_node_attributes(G, radii, "radius")
822: (4)                return G
```

File 437 - harary_graph.py:

```
1: (0)              """Generators for Harary graphs
2: (0)              This module gives two generators for the Harary graph, which was
3: (0)              introduced by the famous mathematician Frank Harary in his 1962 work [H]_.
4: (0)              The first generator gives the Harary graph that maximizes the node
```

```
(0)             connectivity with given number of nodes and given number of edges.
(0)             The second generator gives the Harary graph that minimizes
(0)             the number of edges in the graph with given node connectivity and
(0)             number of nodes.
(0)             References
(0)             ----------
(0)             .. [H] Harary, F. "The Maximum Connectivity of a Graph."
(7)                    Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.
(0)             """
(0)             import networkx as nx
(0)             from networkx.exception import NetworkXError
(0)             __all__ = ["hnm_harary_graph", "hkn_harary_graph"]
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def hnm_harary_graph(n, m, create_using=None):
(4)                 """Returns the Harary graph with given numbers of nodes and edges.
(4)                 The Harary graph $H_{n,m}$ is the graph that maximizes node connectivity
(4)                 with $n$ nodes and $m$ edges.
(4)                 This maximum node connectivity is known to be floor($2m/n$). [1]_
(4)                 Parameters
(4)                 ----------
(4)                 n: integer
(7)                    The number of nodes the generated graph is to contain
(4)                 m: integer
(7)                    The number of edges the generated graph is to contain
(4)                 create_using : NetworkX graph constructor, optional Graph type
(5)                  to create (default=nx.Graph). If graph instance, then cleared
(5)                  before populated.
(4)                 Returns
(4)                 -------
(4)                 NetworkX graph
(8)                     The Harary graph $H_{n,m}$.
(4)                 See Also
(4)                 --------
(4)                 hkn_harary_graph
(4)                 Notes
(4)                 -----
(4)                 This algorithm runs in $O(m)$ time.
(4)                 It is implemented by following the Reference [2]_.
(4)                 References
(4)                 ----------
(4)                 .. [1] F. T. Boesch, A. Satyanarayana, and C. L. Suffel,
(7)                    "A Survey of Some Network Reliability Analysis and Synthesis Results,"
(7)                    Networks, pp. 99-107, 2009.
(4)                 .. [2] Harary, F. "The Maximum Connectivity of a Graph."
(7)                    Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.
(4)                 """
(4)                 if n < 1:
(8)                     raise NetworkXError("The number of nodes must be >= 1!")
(4)                 if m < n - 1:
(8)                     raise NetworkXError("The number of edges must be >= n - 1 !")
(4)                 if m > n * (n - 1) // 2:
(8)                     raise NetworkXError("The number of edges must be <= n(n-1)/2")
(4)                 H = nx.empty_graph(n, create_using)
(4)                 d = 2 * m // n
(4)                 if (n % 2 == 0) or (d % 2 == 0):
(8)                     offset = d // 2
(8)                     for i in range(n):
(12)                        for j in range(1, offset + 1):
(16)                            H.add_edge(i, (i - j) % n)
(16)                            H.add_edge(i, (i + j) % n)
(8)                     if d & 1:
(12)                        half = n // 2
(12)                        for i in range(half):
(16)                            H.add_edge(i, i + half)
(8)                     r = 2 * m % n
(8)                     if r > 0:
(12)                        for i in range(r // 2):
(16)                            H.add_edge(i, i + offset + 1)
(4)                 else:
```

```
(8)                     offset = (d - 1) // 2
(8)                     for i in range(n):
(12)                        for j in range(1, offset + 1):
(16)                            H.add_edge(i, (i - j) % n)
(16)                            H.add_edge(i, (i + j) % n)
(8)                     half = n // 2
(8)                     for i in range(m - n * offset):
(12)                        H.add_edge(i, (i + half) % n)
(4)                 return H
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def hkn_harary_graph(k, n, create_using=None):
(4)                 """Returns the Harary graph with given node connectivity and node number.
(4)                 The Harary graph $H_{k,n}$ is the graph that minimizes the number of
(4)                 edges needed with given node connectivity $k$ and node number $n$.
(4)                 This smallest number of edges is known to be ceil($kn/2$) [1]_.
(4)                 Parameters
(4)                 ----------
(4)                 k: integer
(7)                    The node connectivity of the generated graph
(4)                 n: integer
(7)                    The number of nodes the generated graph is to contain
(4)                 create_using : NetworkX graph constructor, optional Graph type
(5)                  to create (default=nx.Graph). If graph instance, then cleared
(5)                  before populated.
(4)                 Returns
(4)                 -------
(4)                 NetworkX graph
(8)                     The Harary graph $H_{k,n}$.
(4)                 See Also
(4)                 --------
(4)                 hnm_harary_graph
(4)                 Notes
(4)                 -----
(4)                 This algorithm runs in $O(kn)$ time.
(4)                 It is implemented by following the Reference [2]_.
(4)                 References
(4)                 ----------
(4)                 .. [1] Weisstein, Eric W. "Harary Graph." From MathWorld--A Wolfram Web
(5)                  Resource. http://mathworld.wolfram.com/HararyGraph.html.
(4)                 .. [2] Harary, F. "The Maximum Connectivity of a Graph."
(6)                   Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.
(4)                 """
(4)                 if k < 1:
(8)                     raise NetworkXError("The node connectivity must be >= 1!")
(4)                 if n < k + 1:
(8)                     raise NetworkXError("The number of nodes must be >= k+1 !")
(4)                 if k == 1:
(8)                     H = nx.path_graph(n, create_using)
(8)                     return H
(4)                 H = nx.empty_graph(n, create_using)
(4)                 if (k % 2 == 0) or (n % 2 == 0):
(8)                     offset = k // 2
(8)                     for i in range(n):
(12)                        for j in range(1, offset + 1):
(16)                            H.add_edge(i, (i - j) % n)
(16)                            H.add_edge(i, (i + j) % n)
(8)                     if k & 1:
(12)                        half = n // 2
(12)                        for i in range(half):
(16)                            H.add_edge(i, i + half)
(4)                 else:
(8)                     offset = (k - 1) // 2
(8)                     for i in range(n):
(12)                        for j in range(1, offset + 1):
(16)                            H.add_edge(i, (i - j) % n)
(16)                            H.add_edge(i, (i + j) % n)
(8)                     half = n // 2
(8)                     for i in range(half + 1):
(12)                        H.add_edge(i, (i + half) % n)
```

```
143: (4)                return H

----------------------------------------

File 438 - intersection.py:

1: (0)              """
2: (0)              Generators for random intersection graphs.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import py_random_state
6: (0)              __all__ = [
7: (4)                  "uniform_random_intersection_graph",
8: (4)                  "k_random_intersection_graph",
9: (4)                  "general_random_intersection_graph",
10: (0)             ]
11: (0)             @py_random_state(3)
12: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
13: (0)             def uniform_random_intersection_graph(n, m, p, seed=None):
14: (4)                 """Returns a uniform random intersection graph.
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 n : int
18: (8)                     The number of nodes in the first bipartite set (nodes)
19: (4)                 m : int
20: (8)                     The number of nodes in the second bipartite set (attributes)
21: (4)                 p : float
22: (8)                     Probability of connecting nodes between bipartite sets
23: (4)                 seed : integer, random_state, or None (default)
24: (8)                     Indicator of random number generation state.
25: (8)                     See :ref:`Randomness<randomness>`.
26: (4)                 See Also
27: (4)                 --------
28: (4)                 gnp_random_graph
29: (4)                 References
30: (4)                 ----------
31: (4)                 .. [1] K.B. Singer-Cohen, Random Intersection Graphs, 1995,
32: (7)                    PhD thesis, Johns Hopkins University
33: (4)                 .. [2] Fill, J. A., Scheinerman, E. R., and Singer-Cohen, K. B.,
34: (7)                    Random intersection graphs when m = !(n):
35: (7)                    An equivalence theorem relating the evolution of the g(n, m, p)
36: (7)                    and g(n, p) models. Random Struct. Algorithms 16, 2 (2000), 156–176.
37: (4)                 """
38: (4)                 from networkx.algorithms import bipartite
39: (4)                 G = bipartite.random_graph(n, m, p, seed)
40: (4)                 return nx.projected_graph(G, range(n))
41: (0)             @py_random_state(3)
42: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
43: (0)             def k_random_intersection_graph(n, m, k, seed=None):
44: (4)                 """Returns a intersection graph with randomly chosen attribute sets for
45: (4)                 each node that are of equal size (k).
46: (4)                 Parameters
47: (4)                 ----------
48: (4)                 n : int
49: (8)                     The number of nodes in the first bipartite set (nodes)
50: (4)                 m : int
51: (8)                     The number of nodes in the second bipartite set (attributes)
52: (4)                 k : float
53: (8)                     Size of attribute set to assign to each node.
54: (4)                 seed : integer, random_state, or None (default)
55: (8)                     Indicator of random number generation state.
56: (8)                     See :ref:`Randomness<randomness>`.
57: (4)                 See Also
58: (4)                 --------
59: (4)                 gnp_random_graph, uniform_random_intersection_graph
60: (4)                 References
61: (4)                 ----------
62: (4)                 .. [1] Godehardt, E., and Jaworski, J.
63: (7)                    Two models of random intersection graphs and their applications.
```

```
 64: (7)                    Electronic Notes in Discrete Mathematics 10 (2001), 129--132.
 65: (4)                 """
 66: (4)                 G = nx.empty_graph(n + m)
 67: (4)                 mset = range(n, n + m)
 68: (4)                 for v in range(n):
 69: (8)                     targets = seed.sample(mset, k)
 70: (8)                     G.add_edges_from(zip([v] * len(targets), targets))
 71: (4)                 return nx.projected_graph(G, range(n))
 72: (0)             @py_random_state(3)
 73: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 74: (0)             def general_random_intersection_graph(n, m, p, seed=None):
 75: (4)                 """Returns a random intersection graph with independent probabilities
 76: (4)                 for connections between node and attribute sets.
 77: (4)                 Parameters
 78: (4)                 ----------
 79: (4)                 n : int
 80: (8)                     The number of nodes in the first bipartite set (nodes)
 81: (4)                 m : int
 82: (8)                     The number of nodes in the second bipartite set (attributes)
 83: (4)                 p : list of floats of length m
 84: (8)                     Probabilities for connecting nodes to each attribute
 85: (4)                 seed : integer, random_state, or None (default)
 86: (8)                     Indicator of random number generation state.
 87: (8)                     See :ref:`Randomness<randomness>`.
 88: (4)                 See Also
 89: (4)                 --------
 90: (4)                 gnp_random_graph, uniform_random_intersection_graph
 91: (4)                 References
 92: (4)                 ----------
 93: (4)                 .. [1] Nikoletseas, S. E., Raptopoulos, C., and Spirakis, P. G.
 94: (7)                    The existence and efficient construction of large independent sets
 95: (7)                    in general random intersection graphs. In ICALP (2004), J. D´ıaz,
 96: (7)                    J. Karhum¨aki, A. Lepist¨o, and D. Sannella, Eds., vol. 3142
 97: (7)                    of Lecture Notes in Computer Science, Springer, pp. 1029–1040.
 98: (4)                 """
 99: (4)                 if len(p) != m:
100: (8)                     raise ValueError("Probability list p must have m elements.")
101: (4)                 G = nx.empty_graph(n + m)
102: (4)                 mset = range(n, n + m)
103: (4)                 for u in range(n):
104: (8)                     for v, q in zip(mset, p):
105: (12)                        if seed.random() < q:
106: (16)                            G.add_edge(u, v)
107: (4)                 return nx.projected_graph(G, range(n))

----------------------------------------

File 439 - internet_as_graphs.py:

 1: (0)              """Generates graphs resembling the Internet Autonomous System network"""
 2: (0)              import networkx as nx
 3: (0)              from networkx.utils import py_random_state
 4: (0)              __all__ = ["random_internet_as_graph"]
 5: (0)              def uniform_int_from_avg(a, m, seed):
 6: (4)                  """Pick a random integer with uniform probability.
 7: (4)                  Returns a random integer uniformly taken from a distribution with
 8: (4)                  minimum value 'a' and average value 'm', X~U(a,b), E[X]=m, X in N where
 9: (4)                  b = 2*m - a.
10: (4)                  Notes
11: (4)                  -----
12: (4)                  p = (b-floor(b))/2
13: (4)                  X = X1 + X2; X1~U(a,floor(b)), X2~B(p)
14: (4)                  E[X] = E[X1] + E[X2] = (floor(b)+a)/2 + (b-floor(b))/2 = (b+a)/2 = m
15: (4)                  """
16: (4)                  from math import floor
17: (4)                  assert m >= a
18: (4)                  b = 2 * m - a
19: (4)                  p = (b - floor(b)) / 2
20: (4)                  X1 = round(seed.random() * (floor(b) - a) + a)
```

```
21: (4)                 if seed.random() < p:
22: (8)                     X2 = 1
23: (4)                 else:
24: (8)                     X2 = 0
25: (4)                 return X1 + X2
26: (0)             def choose_pref_attach(degs, seed):
27: (4)                 """Pick a random value, with a probability given by its weight.
28: (4)                 Returns a random choice among degs keys, each of which has a
29: (4)                 probability proportional to the corresponding dictionary value.
30: (4)                 Parameters
31: (4)                 ----------
32: (4)                 degs: dictionary
33: (8)                     It contains the possible values (keys) and the corresponding
34: (8)                     probabilities (values)
35: (4)                 seed: random state
36: (4)                 Returns
37: (4)                 -------
38: (4)                 v: object
39: (8)                     A key of degs or None if degs is empty
40: (4)                 """
41: (4)                 if len(degs) == 0:
42: (8)                     return None
43: (4)                 s = sum(degs.values())
44: (4)                 if s == 0:
45: (8)                     return seed.choice(list(degs.keys()))
46: (4)                 v = seed.random() * s
47: (4)                 nodes = list(degs.keys())
48: (4)                 i = 0
49: (4)                 acc = degs[nodes[i]]
50: (4)                 while v > acc:
51: (8)                     i += 1
52: (8)                     acc += degs[nodes[i]]
53: (4)                 return nodes[i]
54: (0)             class AS_graph_generator:
55: (4)                 """Generates random internet AS graphs."""
56: (4)                 def __init__(self, n, seed):
57: (8)                     """Initializes variables. Immediate numbers are taken from [1].
58: (8)                     Parameters
59: (8)                     ----------
60: (8)                     n: integer
61: (12)                        Number of graph nodes
62: (8)                     seed: random state
63: (12)                        Indicator of random number generation state.
64: (12)                        See :ref:`Randomness<randomness>`.
65: (8)                     Returns
66: (8)                     -------
67: (8)                     GG: AS_graph_generator object
68: (8)                     References
69: (8)                     ----------
70: (8)                     [1] A. Elmokashfi, A. Kvalbein and C. Dovrolis, "On the Scalability of
71: (8)                     BGP: The Role of Topology Growth," in IEEE Journal on Selected Areas
72: (8)                     in Communications, vol. 28, no. 8, pp. 1250-1261, October 2010.
73: (8)                     """
74: (8)                     self.seed = seed
75: (8)                     self.n_t = min(n, round(self.seed.random() * 2 + 4))  # num of T nodes
76: (8)                     self.n_m = round(0.15 * n)  # number of M nodes
77: (8)                     self.n_cp = round(0.05 * n)  # number of CP nodes
78: (8)                     self.n_c = max(0, n - self.n_t - self.n_m - self.n_cp)  # number of C
nodes
79: (8)                     self.d_m = 2 + (2.5 * n) / 10000  # average multihoming degree for M
nodes
80: (8)                     self.d_cp = 2 + (1.5 * n) / 10000  # avg multihoming degree for CP
nodes
81: (8)                     self.d_c = 1 + (5 * n) / 100000  # average multihoming degree for C
nodes
82: (8)                     self.p_m_m = 1 + (2 * n) / 10000  # avg num of peer edges between M
and M
83: (8)                     self.p_cp_m = 0.2 + (2 * n) / 10000  # avg num of peer edges between
CP, M
```

```
84: (8)                     self.p_cp_cp = 0.05 + (2 * n) / 100000  # avg num of peer edges btwn
CP, CP
85: (8)                     self.t_m = 0.375  # probability M's provider is T
86: (8)                     self.t_cp = 0.375  # probability CP's provider is T
87: (8)                     self.t_c = 0.125  # probability C's provider is T
88: (4)                 def t_graph(self):
89: (8)                     """Generates the core mesh network of tier one nodes of a AS graph.
90: (8)                     Returns
91: (8)                     -------
92: (8)                     G: Networkx Graph
93: (12)                        Core network
94: (8)                     """
95: (8)                     self.G = nx.Graph()
96: (8)                     for i in range(self.n_t):
97: (12)                        self.G.add_node(i, type="T")
98: (12)                        for r in self.regions:
99: (16)                            self.regions[r].add(i)
100: (12)                       for j in self.G.nodes():
101: (16)                           if i != j:
102: (20)                               self.add_edge(i, j, "peer")
103: (12)                       self.customers[i] = set()
104: (12)                       self.providers[i] = set()
105: (8)                    return self.G
106: (4)                def add_edge(self, i, j, kind):
107: (8)                    if kind == "transit":
108: (12)                       customer = str(i)
109: (8)                    else:
110: (12)                       customer = "none"
111: (8)                    self.G.add_edge(i, j, type=kind, customer=customer)
112: (4)                def choose_peer_pref_attach(self, node_list):
113: (8)                    """Pick a node with a probability weighted by its peer degree.
114: (8)                    Pick a node from node_list with preferential attachment
115: (8)                    computed only on their peer degree
116: (8)                    """
117: (8)                    d = {}
118: (8)                    for n in node_list:
119: (12)                       d[n] = self.G.nodes[n]["peers"]
120: (8)                    return choose_pref_attach(d, self.seed)
121: (4)                def choose_node_pref_attach(self, node_list):
122: (8)                    """Pick a node with a probability weighted by its degree.
123: (8)                    Pick a node from node_list with preferential attachment
124: (8)                    computed on their degree
125: (8)                    """
126: (8)                    degs = dict(self.G.degree(node_list))
127: (8)                    return choose_pref_attach(degs, self.seed)
128: (4)                def add_customer(self, i, j):
129: (8)                    """Keep the dictionaries 'customers' and 'providers' consistent."""
130: (8)                    self.customers[j].add(i)
131: (8)                    self.providers[i].add(j)
132: (8)                    for z in self.providers[j]:
133: (12)                       self.customers[z].add(i)
134: (12)                       self.providers[i].add(z)
135: (4)                def add_node(self, i, kind, reg2prob, avg_deg, t_edge_prob):
136: (8)                    """Add a node and its customer transit edges to the graph.
137: (8)                    Parameters
138: (8)                    ----------
139: (8)                    i: object
140: (12)                       Identifier of the new node
141: (8)                    kind: string
142: (12)                       Type of the new node. Options are: 'M' for middle node, 'CP' for
143: (12)                       content provider and 'C' for customer.
144: (8)                    reg2prob: float
145: (12)                       Probability the new node can be in two different regions.
146: (8)                    avg_deg: float
147: (12)                       Average number of transit nodes of which node i is customer.
148: (8)                    t_edge_prob: float
149: (12)                       Probability node i establish a customer transit edge with a tier
150: (12)                       one (T) node
151: (8)                    Returns
```

```
(8)                     -------
(8)                     i: object
(12)                        Identifier of the new node
(8)                     """
(8)                     regs = 1  # regions in which node resides
(8)                     if self.seed.random() < reg2prob:  # node is in two regions
(12)                        regs = 2
(8)                     node_options = set()
(8)                     self.G.add_node(i, type=kind, peers=0)
(8)                     self.customers[i] = set()
(8)                     self.providers[i] = set()
(8)                     self.nodes[kind].add(i)
(8)                     for r in self.seed.sample(list(self.regions), regs):
(12)                        node_options = node_options.union(self.regions[r])
(12)                        self.regions[r].add(i)
(8)                     edge_num = uniform_int_from_avg(1, avg_deg, self.seed)
(8)                     t_options = node_options.intersection(self.nodes["T"])
(8)                     m_options = node_options.intersection(self.nodes["M"])
(8)                     if i in m_options:
(12)                        m_options.remove(i)
(8)                     d = 0
(8)                     while d < edge_num and (len(t_options) > 0 or len(m_options) > 0):
(12)                        if len(m_options) == 0 or (
(16)                            len(t_options) > 0 and self.seed.random() < t_edge_prob
(12)                        ):  # add edge to a T node
(16)                            j = self.choose_node_pref_attach(t_options)
(16)                            t_options.remove(j)
(12)                        else:
(16)                            j = self.choose_node_pref_attach(m_options)
(16)                            m_options.remove(j)
(12)                        self.add_edge(i, j, "transit")
(12)                        self.add_customer(i, j)
(12)                        d += 1
(8)                     return i
(4)                 def add_m_peering_link(self, m, to_kind):
(8)                     """Add a peering link between two middle tier (M) nodes.
(8)                     Target node j is drawn considering a preferential attachment based on
(8)                     other M node peering degree.
(8)                     Parameters
(8)                     ----------
(8)                     m: object
(12)                        Node identifier
(8)                     to_kind: string
(12)                        type for target node j (must be always M)
(8)                     Returns
(8)                     -------
(8)                     success: boolean
(8)                     """
(8)                     node_options = self.nodes["M"].difference(self.customers[m])
(8)                     node_options = node_options.difference(self.providers[m])
(8)                     if m in node_options:
(12)                        node_options.remove(m)
(8)                     for j in self.G.neighbors(m):
(12)                        if j in node_options:
(16)                            node_options.remove(j)
(8)                     if len(node_options) > 0:
(12)                        j = self.choose_peer_pref_attach(node_options)
(12)                        self.add_edge(m, j, "peer")
(12)                        self.G.nodes[m]["peers"] += 1
(12)                        self.G.nodes[j]["peers"] += 1
(12)                        return True
(8)                     else:
(12)                        return False
(4)                 def add_cp_peering_link(self, cp, to_kind):
(8)                     """Add a peering link to a content provider (CP) node.
(8)                     Target node j can be CP or M and it is drawn uniformly among the nodes
(8)                     belonging to the same region as cp.
(8)                     Parameters
(8)                     ----------
```

```
(8)                     cp: object
(12)                        Node identifier
(8)                     to_kind: string
(12)                        type for target node j (must be M or CP)
(8)                     Returns
(8)                     -------
(8)                     success: boolean
(8)                     """
(8)                     node_options = set()
(8)                     for r in self.regions:  # options include nodes in the same region(s)
(12)                        if cp in self.regions[r]:
(16)                            node_options = node_options.union(self.regions[r])
(8)                     node_options = self.nodes[to_kind].intersection(node_options)
(8)                     if cp in node_options:
(12)                        node_options.remove(cp)
(8)                     node_options = node_options.difference(self.providers[cp])
(8)                     for j in self.G.neighbors(cp):
(12)                        if j in node_options:
(16)                            node_options.remove(j)
(8)                     if len(node_options) > 0:
(12)                        j = self.seed.sample(list(node_options), 1)[0]
(12)                        self.add_edge(cp, j, "peer")
(12)                        self.G.nodes[cp]["peers"] += 1
(12)                        self.G.nodes[j]["peers"] += 1
(12)                        return True
(8)                     else:
(12)                        return False
(4)                 def graph_regions(self, rn):
(8)                     """Initializes AS network regions.
(8)                     Parameters
(8)                     ----------
(8)                     rn: integer
(12)                        Number of regions
(8)                     """
(8)                     self.regions = {}
(8)                     for i in range(rn):
(12)                        self.regions["REG" + str(i)] = set()
(4)                 def add_peering_links(self, from_kind, to_kind):
(8)                     """Utility function to add peering links among node groups."""
(8)                     peer_link_method = None
(8)                     if from_kind == "M":
(12)                        peer_link_method = self.add_m_peering_link
(12)                        m = self.p_m_m
(8)                     if from_kind == "CP":
(12)                        peer_link_method = self.add_cp_peering_link
(12)                        if to_kind == "M":
(16)                            m = self.p_cp_m
(12)                        else:
(16)                            m = self.p_cp_cp
(8)                     for i in self.nodes[from_kind]:
(12)                        num = uniform_int_from_avg(0, m, self.seed)
(12)                        for _ in range(num):
(16)                            peer_link_method(i, to_kind)
(4)                 def generate(self):
(8)                     """Generates a random AS network graph as described in [1].
(8)                     Returns
(8)                     -------
(8)                     G: Graph object
(8)                     Notes
(8)                     -----
(8)                     The process steps are the following: first we create the core network
(8)                     of tier one nodes, then we add the middle tier (M), the content
(8)                     provider (CP) and the customer (C) nodes along with their transit
edges
(8)                     (link i,j means i is customer of j). Finally we add peering links
(8)                     between M nodes, between M and CP nodes and between CP node couples.
(8)                     For a detailed description of the algorithm, please refer to [1].
(8)                     References
(8)                     ----------
```

```
289: (8)                    [1] A. Elmokashfi, A. Kvalbein and C. Dovrolis, "On the Scalability of
290: (8)                    BGP: The Role of Topology Growth," in IEEE Journal on Selected Areas
291: (8)                    in Communications, vol. 28, no. 8, pp. 1250-1261, October 2010.
292: (8)                    """
293: (8)                    self.graph_regions(5)
294: (8)                    self.customers = {}
295: (8)                    self.providers = {}
296: (8)                    self.nodes = {"T": set(), "M": set(), "CP": set(), "C": set()}
297: (8)                    self.t_graph()
298: (8)                    self.nodes["T"] = set(self.G.nodes())
299: (8)                    i = len(self.nodes["T"])
300: (8)                    for _ in range(self.n_m):
301: (12)                       self.nodes["M"].add(self.add_node(i, "M", 0.2, self.d_m,
self.t_m))
302: (12)                       i += 1
303: (8)                    for _ in range(self.n_cp):
304: (12)                       self.nodes["CP"].add(self.add_node(i, "CP", 0.05, self.d_cp,
self.t_cp))
305: (12)                       i += 1
306: (8)                    for _ in range(self.n_c):
307: (12)                       self.nodes["C"].add(self.add_node(i, "C", 0, self.d_c, self.t_c))
308: (12)                       i += 1
309: (8)                    self.add_peering_links("M", "M")
310: (8)                    self.add_peering_links("CP", "M")
311: (8)                    self.add_peering_links("CP", "CP")
312: (8)                    return self.G
313: (0)            @py_random_state(1)
314: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
315: (0)            def random_internet_as_graph(n, seed=None):
316: (4)                """Generates a random undirected graph resembling the Internet AS network
317: (4)                Parameters
318: (4)                ----------
319: (4)                n: integer in [1000, 10000]
320: (8)                    Number of graph nodes
321: (4)                seed : integer, random_state, or None (default)
322: (8)                    Indicator of random number generation state.
323: (8)                    See :ref:`Randomness<randomness>`.
324: (4)                Returns
325: (4)                -------
326: (4)                G: Networkx Graph object
327: (8)                    A randomly generated undirected graph
328: (4)                Notes
329: (4)                -----
330: (4)                This algorithm returns an undirected graph resembling the Internet
331: (4)                Autonomous System (AS) network, it uses the approach by Elmokashfi et al.
332: (4)                [1]_ and it grants the properties described in the related paper [1]_.
333: (4)                Each node models an autonomous system, with an attribute 'type' specifying
334: (4)                its kind; tier-1 (T), mid-level (M), customer (C) or content-provider
(CP).
335: (4)                Each edge models an ADV communication link (hence, bidirectional) with
336: (4)                attributes:
337: (6)                  - type: transit|peer, the kind of commercial agreement between nodes;
338: (6)                  - customer: <node id>, the identifier of the node acting as customer
339: (8)                    ('none' if type is peer).
340: (4)                References
341: (4)                ----------
342: (4)                .. [1] A. Elmokashfi, A. Kvalbein and C. Dovrolis, "On the Scalability of
343: (7)                   BGP: The Role of Topology Growth," in IEEE Journal on Selected Areas
344: (7)                   in Communications, vol. 28, no. 8, pp. 1250-1261, October 2010.
345: (4)                """
346: (4)                GG = AS_graph_generator(n, seed)
347: (4)                G = GG.generate()
348: (4)                return G

----------------------------------------

File 440 - interval_graph.py:

1: (0)              """
```

```
 2: (0)              Generators for interval graph.
 3: (0)              """
 4: (0)              from collections.abc import Sequence
 5: (0)              import networkx as nx
 6: (0)              __all__ = ["interval_graph"]
 7: (0)              @nx._dispatchable(graphs=None, returns_graph=True)
 8: (0)              def interval_graph(intervals):
 9: (4)                  """Generates an interval graph for a list of intervals given.
 10: (4)                 In graph theory, an interval graph is an undirected graph formed from a
 set
 11: (4)                 of closed intervals on the real line, with a vertex for each interval
 12: (4)                 and an edge between vertices whose intervals intersect.
 13: (4)                 It is the intersection graph of the intervals.
 14: (4)                 More information can be found at:
 15: (4)                 https://en.wikipedia.org/wiki/Interval_graph
 16: (4)                 Parameters
 17: (4)                 ----------
 18: (4)                 intervals : a sequence of intervals, say (l, r) where l is the left end,
 19: (4)                 and r is the right end of the closed interval.
 20: (4)                 Returns
 21: (4)                 -------
 22: (4)                 G : networkx graph
 23: (4)                 Examples
 24: (4)                 --------
 25: (4)                 >>> intervals = [(-2, 3), [1, 4], (2, 3), (4, 6)]
 26: (4)                 >>> G = nx.interval_graph(intervals)
 27: (4)                 >>> sorted(G.edges)
 28: (4)                 [((-2, 3), (1, 4)), ((-2, 3), (2, 3)), ((1, 4), (2, 3)), ((1, 4), (4, 6))]
 29: (4)                 Raises
 30: (4)                 ------
 31: (4)                 :exc:`TypeError`
 32: (8)                     if `intervals` contains None or an element which is not
 33: (8)                     collections.abc.Sequence or not a length of 2.
 34: (4)                 :exc:`ValueError`
 35: (8)                     if `intervals` contains an interval such that min1 > max1
 36: (8)                     where min1,max1 = interval
 37: (4)                 """
 38: (4)                 intervals = list(intervals)
 39: (4)                 for interval in intervals:
 40: (8)                     if not (isinstance(interval, Sequence) and len(interval) == 2):
 41: (12)                        raise TypeError(
 42: (16)                            "Each interval must have length 2, and be a "
 43: (16)                            "collections.abc.Sequence such as tuple or list."
 44: (12)                        )
 45: (8)                     if interval[0] > interval[1]:
 46: (12)                        raise ValueError(f"Interval must have lower value first. Got
 {interval}")
 47: (4)                 graph = nx.Graph()
 48: (4)                 tupled_intervals = [tuple(interval) for interval in intervals]
 49: (4)                 graph.add_nodes_from(tupled_intervals)
 50: (4)                 while tupled_intervals:
 51: (8)                     min1, max1 = interval1 = tupled_intervals.pop()
 52: (8)                     for interval2 in tupled_intervals:
 53: (12)                        min2, max2 = interval2
 54: (12)                        if max1 >= min2 and max2 >= min1:
 55: (16)                            graph.add_edge(interval1, interval2)
 56: (4)                 return graph

 ----------------------------------------

 File 441 - joint_degree_seq.py:

 1: (0)              """Generate graphs with a given joint degree and directed joint degree"""
 2: (0)              import networkx as nx
 3: (0)              from networkx.utils import py_random_state
 4: (0)              __all__ = [
 5: (4)                  "is_valid_joint_degree",
 6: (4)                  "is_valid_directed_joint_degree",
 7: (4)                  "joint_degree_graph",
```

```
8: (4)                  "directed_joint_degree_graph",
9: (0)              ]
10: (0)             @nx._dispatchable(graphs=None)
11: (0)             def is_valid_joint_degree(joint_degrees):
12: (4)                 """Checks whether the given joint degree dictionary is realizable.
13: (4)                 A *joint degree dictionary* is a dictionary of dictionaries, in
14: (4)                 which entry ``joint_degrees[k][l]`` is an integer representing the
15: (4)                 number of edges joining nodes of degree *k* with nodes of degree
16: (4)                 *l*. Such a dictionary is realizable as a simple graph if and only
17: (4)                 if the following conditions are satisfied.
18: (4)                 - each entry must be an integer,
19: (4)                 - the total number of nodes of degree *k*, computed by
20: (6)                   ``sum(joint_degrees[k].values()) / k``, must be an integer,
21: (4)                 - the total number of edges joining nodes of degree *k* with
22: (6)                   nodes of degree *l* cannot exceed the total number of possible edges,
23: (4)                 - each diagonal entry ``joint_degrees[k][k]`` must be even (this is
24: (6)                   a convention assumed by the :func:`joint_degree_graph` function).
25: (4)                 Parameters
26: (4)                 ----------
27: (4)                 joint_degrees :  dictionary of dictionary of integers
28: (8)                     A joint degree dictionary in which entry ``joint_degrees[k][l]``
29: (8)                     is the number of edges joining nodes of degree *k* with nodes of
30: (8)                     degree *l*.
31: (4)                 Returns
32: (4)                 -------
33: (4)                 bool
34: (8)                     Whether the given joint degree dictionary is realizable as a
35: (8)                     simple graph.
36: (4)                 References
37: (4)                 ----------
38: (4)                 .. [1] M. Gjoka, M. Kurant, A. Markopoulou, "2.5K Graphs: from Sampling
39: (7)                    to Generation", IEEE Infocom, 2013.
40: (4)                 .. [2] I. Stanton, A. Pinar, "Constructing and sampling graphs with a
41: (7)                    prescribed joint degree distribution", Journal of Experimental
42: (7)                    Algorithmics, 2012.
43: (4)                 """
44: (4)                 degree_count = {}
45: (4)                 for k in joint_degrees:
46: (8)                     if k > 0:
47: (12)                        k_size = sum(joint_degrees[k].values()) / k
48: (12)                        if not k_size.is_integer():
49: (16)                            return False
50: (12)                        degree_count[k] = k_size
51: (4)                 for k in joint_degrees:
52: (8)                     for l in joint_degrees[k]:
53: (12)                        if not float(joint_degrees[k][l]).is_integer():
54: (16)                            return False
55: (12)                        if (k != l) and (joint_degrees[k][l] > degree_count[k] *
degree_count[l]):
56: (16)                            return False
57: (12)                        elif k == l:
58: (16)                            if joint_degrees[k][k] > degree_count[k] * (degree_count[k] -
1):
59: (20)                                return False
60: (16)                            if joint_degrees[k][k] % 2 != 0:
61: (20)                                return False
62: (4)                 return True
63: (0)             def _neighbor_switch(G, w, unsat, h_node_residual, avoid_node_id=None):
64: (4)                 """Releases one free stub for ``w``, while preserving joint degree in G.
65: (4)                 Parameters
66: (4)                 ----------
67: (4)                 G : NetworkX graph
68: (8)                     Graph in which the neighbor switch will take place.
69: (4)                 w : integer
70: (8)                     Node id for which we will execute this neighbor switch.
71: (4)                 unsat : set of integers
72: (8)                     Set of unsaturated node ids that have the same degree as w.
73: (4)                 h_node_residual: dictionary of integers
74: (8)                     Keeps track of the remaining stubs  for a given node.
```

```
(4)                 avoid_node_id: integer
(8)                     Node id to avoid when selecting w_prime.
(4)                 Notes
(4)                 -----
(4)                 First, it selects *w_prime*, an  unsaturated node that has the same degree
(4)                 as ``w``. Second, it selects *switch_node*, a neighbor node of ``w`` that
(4)                 is not  connected to *w_prime*. Then it executes an edge swap i.e. removes
(4)                 (``w``,*switch_node*) and adds (*w_prime*,*switch_node*). Gjoka et. al.
[1]
(4)                 prove that such an edge swap is always possible.
(4)                 References
(4)                 ----------
(4)                 .. [1] M. Gjoka, B. Tillman, A. Markopoulou, "Construction of Simple
(7)                    Graphs with a Target Joint Degree Matrix and Beyond", IEEE Infocom, '15
(4)                 """
(4)                 if (avoid_node_id is None) or (h_node_residual[avoid_node_id] > 1):
(8)                     w_prime = next(iter(unsat))
(4)                 else:
(8)                     iter_var = iter(unsat)
(8)                     while True:
(12)                        w_prime = next(iter_var)
(12)                        if w_prime != avoid_node_id:
(16)                            break
(4)                 w_prime_neighbs = G[w_prime]  # slightly faster declaring this variable
(4)                 for v in G[w]:
(8)                     if (v not in w_prime_neighbs) and (v != w_prime):
(12)                        switch_node = v
(12)                        break
(4)                 G.remove_edge(w, switch_node)
(4)                 G.add_edge(w_prime, switch_node)
(4)                 h_node_residual[w] += 1
(4)                 h_node_residual[w_prime] -= 1
(4)                 if h_node_residual[w_prime] == 0:
(8)                     unsat.remove(w_prime)
(0)             @py_random_state(1)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def joint_degree_graph(joint_degrees, seed=None):
(4)                 """Generates a random simple graph with the given joint degree dictionary.
(4)                 Parameters
(4)                 ----------
(4)                 joint_degrees :  dictionary of dictionary of integers
(8)                     A joint degree dictionary in which entry ``joint_degrees[k][l]`` is
the
(8)                     number of edges joining nodes of degree *k* with nodes of degree *l*.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 G : Graph
(8)                     A graph with the specified joint degree dictionary.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If *joint_degrees* dictionary is not realizable.
(4)                 Notes
(4)                 -----
(4)                 In each iteration of the "while loop" the algorithm picks two disconnected
(4)                 nodes *v* and *w*, of degree *k* and *l* correspondingly,  for which
(4)                 ``joint_degrees[k][l]`` has not reached its target yet. It then adds
(4)                 edge (*v*, *w*) and increases the number of edges in graph G by one.
(4)                 The intelligence of the algorithm lies in the fact that  it is always
(4)                 possible to add an edge between such disconnected nodes *v* and *w*,
(4)                 even if one or both nodes do not have free stubs. That is made possible by
(4)                 executing a "neighbor switch", an edge rewiring move that releases
(4)                 a free stub while keeping the joint degree of G the same.
(4)                 The algorithm continues for E (number of edges) iterations of
(4)                 the "while loop", at the which point all entries of the given
(4)                 ``joint_degrees[k][l]`` have reached their target values and the
```

```
(4)                 construction is complete.
(4)                 References
(4)                 ----------
(4)                 ..  [1] M. Gjoka, B. Tillman, A. Markopoulou, "Construction of Simple
(8)                     Graphs with a Target Joint Degree Matrix and Beyond", IEEE Infocom,
'15
(4)                 Examples
(4)                 --------
(4)                 >>> joint_degrees = {
(4)                 ...     1: {4: 1},
(4)                 ...     2: {2: 2, 3: 2, 4: 2},
(4)                 ...     3: {2: 2, 4: 1},
(4)                 ...     4: {1: 1, 2: 2, 3: 1},
(4)                 ... }
(4)                 >>> G = nx.joint_degree_graph(joint_degrees)
(4)                 >>>
(4)                 """
(4)                 if not is_valid_joint_degree(joint_degrees):
(8)                     msg = "Input joint degree dict not realizable as a simple graph"
(8)                     raise nx.NetworkXError(msg)
(4)                 degree_count = {k: sum(l.values()) // k for k, l in joint_degrees.items()
if k > 0}
(4)                 N = sum(degree_count.values())
(4)                 G = nx.empty_graph(N)
(4)                 h_degree_nodelist = {}
(4)                 h_node_residual = {}
(4)                 nodeid = 0
(4)                 for degree, num_nodes in degree_count.items():
(8)                     h_degree_nodelist[degree] = range(nodeid, nodeid + num_nodes)
(8)                     for v in h_degree_nodelist[degree]:
(12)                        h_node_residual[v] = degree
(8)                     nodeid += int(num_nodes)
(4)                 for k in joint_degrees:
(8)                     for l in joint_degrees[k]:
(12)                        n_edges_add = joint_degrees[k][l]
(12)                        if (n_edges_add > 0) and (k >= l):
(16)                            k_size = degree_count[k]
(16)                            l_size = degree_count[l]
(16)                            k_nodes = h_degree_nodelist[k]
(16)                            l_nodes = h_degree_nodelist[l]
(16)                            k_unsat = {v for v in k_nodes if h_node_residual[v] > 0}
(16)                            if k != l:
(20)                                l_unsat = {w for w in l_nodes if h_node_residual[w] > 0}
(16)                            else:
(20)                                l_unsat = k_unsat
(20)                                n_edges_add = joint_degrees[k][l] // 2
(16)                            while n_edges_add > 0:
(20)                                v = k_nodes[seed.randrange(k_size)]
(20)                                w = l_nodes[seed.randrange(l_size)]
(20)                                if not G.has_edge(v, w) and (v != w):
(24)                                    if h_node_residual[v] == 0:
(28)                                        _neighbor_switch(G, v, k_unsat, h_node_residual)
(24)                                    if h_node_residual[w] == 0:
(28)                                        if k != l:
(32)                                            _neighbor_switch(G, w, l_unsat,
h_node_residual)
(28)                                        else:
(32)                                            _neighbor_switch(
(36)                                                G, w, l_unsat, h_node_residual,
avoid_node_id=v
(32)                                            )
(24)                                    G.add_edge(v, w)
(24)                                    h_node_residual[v] -= 1
(24)                                    h_node_residual[w] -= 1
(24)                                    n_edges_add -= 1
(24)                                    if h_node_residual[v] == 0:
(28)                                        k_unsat.discard(v)
(24)                                    if h_node_residual[w] == 0:
(28)                                        l_unsat.discard(w)
```

```
    return G
@nx._dispatchable(graphs=None)
def is_valid_directed_joint_degree(in_degrees, out_degrees, nkk):
    """Checks whether the given directed joint degree input is realizable
    Parameters
    ----------
    in_degrees :  list of integers
        in degree sequence contains the in degrees of nodes.
    out_degrees : list of integers
        out degree sequence contains the out degrees of nodes.
    nkk  :  dictionary of dictionary of integers
        directed joint degree dictionary. for nodes of out degree k (first
        level of dict) and nodes of in degree l (second level of dict)
        describes the number of edges.
    Returns
    -------
    boolean
        returns true if given input is realizable, else returns false.
    Notes
    -----
    Here is the list of conditions that the inputs (in/out degree sequences,
    nkk) need to satisfy for simple directed graph realizability:
    - Condition 0: in_degrees and out_degrees have the same length
    - Condition 1: nkk[k][l]  is integer for all k,l
    - Condition 2: sum(nkk[k])/k = number of nodes with partition id k, is an
                   integer and matching degree sequence
    - Condition 3: number of edges and non-chords between k and l cannot
exceed
                   maximum possible number of edges
    References
    ----------
    [1] B. Tillman, A. Markopoulou, C. T. Butts & M. Gjoka,
        "Construction of Directed 2K Graphs". In Proc. of KDD 2017.
    """
    V = {}  # number of nodes with in/out degree.
    forbidden = {}
    if len(in_degrees) != len(out_degrees):
        return False
    for idx in range(len(in_degrees)):
        i = in_degrees[idx]
        o = out_degrees[idx]
        V[(i, 0)] = V.get((i, 0), 0) + 1
        V[(o, 1)] = V.get((o, 1), 0) + 1
        forbidden[(o, i)] = forbidden.get((o, i), 0) + 1
    S = {}  # number of edges going from in/out degree nodes.
    for k in nkk:
        for l in nkk[k]:
            val = nkk[k][l]
            if not float(val).is_integer():  # condition 1
                return False
            if val > 0:
                S[(k, 1)] = S.get((k, 1), 0) + val
                S[(l, 0)] = S.get((l, 0), 0) + val
                if val + forbidden.get((k, l), 0) > V[(k, 1)] * V[(l, 0)]:
                    return False
    return all(S[s] / s[0] == V[s] for s in S)
def _directed_neighbor_switch(
    G, w, unsat, h_node_residual_out, chords, h_partition_in, partition
):
    """Releases one free stub for node w, while preserving joint degree in G.
    Parameters
    ----------
    G : networkx directed graph
        graph within which the edge swap will take place.
    w : integer
        node id for which we need to perform a neighbor switch.
    unsat: set of integers
        set of node ids that have the same degree as w and are unsaturated.
    h_node_residual_out: dict of integers
```

```
275: (8)                    for a given node, keeps track of the remaining stubs to be added.
276: (4)                chords: set of tuples
277: (8)                    keeps track of available positions to add edges.
278: (4)                h_partition_in: dict of integers
279: (8)                    for a given node, keeps track of its partition id (in degree).
280: (4)                partition: integer
281: (8)                    partition id to check if chords have to be updated.
282: (4)                Notes
283: (4)                -----
284: (4)                First, it selects node w_prime that (1) has the same degree as w and
285: (4)                (2) is unsaturated. Then, it selects node v, a neighbor of w, that is
286: (4)                not connected to w_prime and does an edge swap i.e. removes (w,v) and
287: (4)                adds (w_prime,v). If neighbor switch is not possible for w using
288: (4)                w_prime and v, then return w_prime; in [1] it's proven that
289: (4)                such unsaturated nodes can be used.
290: (4)                References
291: (4)                ----------
292: (4)                [1] B. Tillman, A. Markopoulou, C. T. Butts & M. Gjoka,
293: (8)                    "Construction of Directed 2K Graphs". In Proc. of KDD 2017.
294: (4)                """
295: (4)                w_prime = unsat.pop()
296: (4)                unsat.add(w_prime)
297: (4)                w_neighbs = list(G.successors(w))
298: (4)                w_prime_neighbs = list(G.successors(w_prime))
299: (4)                for v in w_neighbs:
300: (8)                    if (v not in w_prime_neighbs) and w_prime != v:
301: (12)                       G.remove_edge(w, v)
302: (12)                       G.add_edge(w_prime, v)
303: (12)                       if h_partition_in[v] == partition:
304: (16)                           chords.add((w, v))
305: (16)                           chords.discard((w_prime, v))
306: (12)                       h_node_residual_out[w] += 1
307: (12)                       h_node_residual_out[w_prime] -= 1
308: (12)                       if h_node_residual_out[w_prime] == 0:
309: (16)                           unsat.remove(w_prime)
310: (12)                       return None
311: (4)                return w_prime
312: (0)            def _directed_neighbor_switch_rev(
313: (4)                G, w, unsat, h_node_residual_in, chords, h_partition_out, partition
314: (0)            ):
315: (4)                """The reverse of directed_neighbor_switch.
316: (4)                Parameters
317: (4)                ----------
318: (4)                G : networkx directed graph
319: (8)                    graph within which the edge swap will take place.
320: (4)                w : integer
321: (8)                    node id for which we need to perform a neighbor switch.
322: (4)                unsat: set of integers
323: (8)                    set of node ids that have the same degree as w and are unsaturated.
324: (4)                h_node_residual_in: dict of integers
325: (8)                    for a given node, keeps track of the remaining stubs to be added.
326: (4)                chords: set of tuples
327: (8)                    keeps track of available positions to add edges.
328: (4)                h_partition_out: dict of integers
329: (8)                    for a given node, keeps track of its partition id (out degree).
330: (4)                partition: integer
331: (8)                    partition id to check if chords have to be updated.
332: (4)                Notes
333: (4)                -----
334: (4)                Same operation as directed_neighbor_switch except it handles this
operation
335: (4)                for incoming edges instead of outgoing.
336: (4)                """
337: (4)                w_prime = unsat.pop()
338: (4)                unsat.add(w_prime)
339: (4)                w_neighbs = list(G.predecessors(w))
340: (4)                w_prime_neighbs = list(G.predecessors(w_prime))
341: (4)                for v in w_neighbs:
342: (8)                    if (v not in w_prime_neighbs) and w_prime != v:
```

```
343: (12)                        G.remove_edge(v, w)
344: (12)                        G.add_edge(v, w_prime)
345: (12)                        if h_partition_out[v] == partition:
346: (16)                            chords.add((v, w))
347: (16)                            chords.discard((v, w_prime))
348: (12)                        h_node_residual_in[w] += 1
349: (12)                        h_node_residual_in[w_prime] -= 1
350: (12)                        if h_node_residual_in[w_prime] == 0:
351: (16)                            unsat.remove(w_prime)
352: (12)                        return None
353: (4)                    return w_prime
354: (0)                @py_random_state(3)
355: (0)                @nx._dispatchable(graphs=None, returns_graph=True)
356: (0)                def directed_joint_degree_graph(in_degrees, out_degrees, nkk, seed=None):
357: (4)                    """Generates a random simple directed graph with the joint degree.
358: (4)                    Parameters
359: (4)                    ----------
360: (4)                    degree_seq :  list of tuples (of size 3)
361: (8)                        degree sequence contains tuples of nodes with node id, in degree and
362: (8)                        out degree.
363: (4)                    nkk  :  dictionary of dictionary of integers
364: (8)                        directed joint degree dictionary, for nodes of out degree k (first
365: (8)                        level of dict) and nodes of in degree l (second level of dict)
366: (8)                        describes the number of edges.
367: (4)                    seed : hashable object, optional
368: (8)                        Seed for random number generator.
369: (4)                    Returns
370: (4)                    -------
371: (4)                    G : Graph
372: (8)                        A directed graph with the specified inputs.
373: (4)                    Raises
374: (4)                    ------
375: (4)                    NetworkXError
376: (8)                        If degree_seq and nkk are not realizable as a simple directed graph.
377: (4)                    Notes
378: (4)                    -----
379: (4)                    Similarly to the undirected version:
380: (4)                    In each iteration of the "while loop" the algorithm picks two disconnected
381: (4)                    nodes v and w, of degree k and l correspondingly,  for which nkk[k][l] has
382: (4)                    not reached its target yet i.e. (for given k,l): n_edges_add < nkk[k][l].
383: (4)                    It then adds edge (v,w) and always increases the number of edges in graph
G
384: (4)                    by one.
385: (4)                    The intelligence of the algorithm lies in the fact that  it is always
386: (4)                    possible to add an edge between disconnected nodes v and w, for which
387: (4)                    nkk[degree(v)][degree(w)] has not reached its target, even if one or both
388: (4)                    nodes do not have free stubs. If either node v or w does not have a free
389: (4)                    stub, we perform a "neighbor switch", an edge rewiring move that releases
a
390: (4)                    free stub while keeping nkk the same.
391: (4)                    The difference for the directed version lies in the fact that neighbor
392: (4)                    switches might not be able to rewire, but in these cases unsaturated nodes
393: (4)                    can be reassigned to use instead, see [1] for detailed description and
394: (4)                    proofs.
395: (4)                    The algorithm continues for E (number of edges in the graph) iterations of
396: (4)                    the "while loop", at which point all entries of the given nkk[k][l] have
397: (4)                    reached their target values and the construction is complete.
398: (4)                    References
399: (4)                    ----------
400: (4)                    [1] B. Tillman, A. Markopoulou, C. T. Butts & M. Gjoka,
401: (8)                        "Construction of Directed 2K Graphs". In Proc. of KDD 2017.
402: (4)                    Examples
403: (4)                    --------
404: (4)                    >>> in_degrees = [0, 1, 1, 2]
405: (4)                    >>> out_degrees = [1, 1, 1, 1]
406: (4)                    >>> nkk = {1: {1: 2, 2: 2}}
407: (4)                    >>> G = nx.directed_joint_degree_graph(in_degrees, out_degrees, nkk)
408: (4)                    >>>
409: (4)                    """
```

```
(4)                 if not is_valid_directed_joint_degree(in_degrees, out_degrees, nkk):
(8)                     msg = "Input is not realizable as a simple graph"
(8)                     raise nx.NetworkXError(msg)
(4)                 G = nx.DiGraph()
(4)                 h_degree_nodelist_in = {}
(4)                 h_degree_nodelist_out = {}
(4)                 h_degree_nodelist_in_unsat = {}
(4)                 h_degree_nodelist_out_unsat = {}
(4)                 h_node_residual_out = {}
(4)                 h_node_residual_in = {}
(4)                 h_partition_out = {}
(4)                 h_partition_in = {}
(4)                 non_chords = {}
(4)                 for idx, i in enumerate(in_degrees):
(8)                     idx = int(idx)
(8)                     if i > 0:
(12)                        h_degree_nodelist_in.setdefault(i, [])
(12)                        h_degree_nodelist_in_unsat.setdefault(i, set())
(12)                        h_degree_nodelist_in[i].append(idx)
(12)                        h_degree_nodelist_in_unsat[i].add(idx)
(12)                        h_node_residual_in[idx] = i
(12)                        h_partition_in[idx] = i
(4)                 for idx, o in enumerate(out_degrees):
(8)                     o = out_degrees[idx]
(8)                     non_chords[(o, in_degrees[idx])] = non_chords.get((o,
in_degrees[idx]), 0) + 1
(8)                     idx = int(idx)
(8)                     if o > 0:
(12)                        h_degree_nodelist_out.setdefault(o, [])
(12)                        h_degree_nodelist_out_unsat.setdefault(o, set())
(12)                        h_degree_nodelist_out[o].append(idx)
(12)                        h_degree_nodelist_out_unsat[o].add(idx)
(12)                        h_node_residual_out[idx] = o
(12)                        h_partition_out[idx] = o
(8)                     G.add_node(idx)
(4)                 nk_in = {}
(4)                 nk_out = {}
(4)                 for p in h_degree_nodelist_in:
(8)                     nk_in[p] = len(h_degree_nodelist_in[p])
(4)                 for p in h_degree_nodelist_out:
(8)                     nk_out[p] = len(h_degree_nodelist_out[p])
(4)                 for k in nkk:
(8)                     for l in nkk[k]:
(12)                        n_edges_add = nkk[k][l]
(12)                        if n_edges_add > 0:
(16)                            chords = set()
(16)                            k_len = nk_out[k]
(16)                            l_len = nk_in[l]
(16)                            chords_sample = seed.sample(
(20)                                range(k_len * l_len), n_edges_add + non_chords.get((k, l),
0)
(16)                            )
(16)                            num = 0
(16)                            while len(chords) < n_edges_add:
(20)                                i = h_degree_nodelist_out[k][chords_sample[num] % k_len]
(20)                                j = h_degree_nodelist_in[l][chords_sample[num] // k_len]
(20)                                num += 1
(20)                                if i != j:
(24)                                    chords.add((i, j))
(16)                            k_unsat = h_degree_nodelist_out_unsat[k]
(16)                            l_unsat = h_degree_nodelist_in_unsat[l]
(16)                            while n_edges_add > 0:
(20)                                v, w = chords.pop()
(20)                                chords.add((v, w))
(20)                                if h_node_residual_out[v] == 0:
(24)                                    _v = _directed_neighbor_switch(
(28)                                        G,
(28)                                        v,
(28)                                        k_unsat,
```

```
477: (28)                                        h_node_residual_out,
478: (28)                                        chords,
479: (28)                                        h_partition_in,
480: (28)                                        l,
481: (24)                                    )
482: (24)                                    if _v is not None:
483: (28)                                        v = _v
484: (20)                                if h_node_residual_in[w] == 0:
485: (24)                                    _w = _directed_neighbor_switch_rev(
486: (28)                                        G,
487: (28)                                        w,
488: (28)                                        l_unsat,
489: (28)                                        h_node_residual_in,
490: (28)                                        chords,
491: (28)                                        h_partition_out,
492: (28)                                        k,
493: (24)                                    )
494: (24)                                    if _w is not None:
495: (28)                                        w = _w
496: (20)                                G.add_edge(v, w)
497: (20)                                h_node_residual_out[v] -= 1
498: (20)                                h_node_residual_in[w] -= 1
499: (20)                                n_edges_add -= 1
500: (20)                                chords.discard((v, w))
501: (20)                                if h_node_residual_out[v] == 0:
502: (24)                                    k_unsat.discard(v)
503: (20)                                if h_node_residual_in[w] == 0:
504: (24)                                    l_unsat.discard(w)
505: (4)                return G
```

File 442 - lattice.py:

```
1: (0)              """Functions for generating grid graphs and lattices
2: (0)              The :func:`grid_2d_graph`, :func:`triangular_lattice_graph`, and
3: (0)              :func:`hexagonal_lattice_graph` functions correspond to the three
4: (0)              `regular tilings of the plane`_, the square, triangular, and hexagonal
5: (0)              tilings, respectively. :func:`grid_graph` and :func:`hypercube_graph`
6: (0)              are similar for arbitrary dimensions. Useful relevant discussion can
7: (0)              be found about `Triangular Tiling`_, and `Square, Hex and Triangle Grids`_
8: (0)              .. _regular tilings of the plane:
https://en.wikipedia.org/wiki/List_of_regular_polytopes_and_compounds#Euclidean_tilings
9: (0)              .. _Square, Hex and Triangle Grids: http://www-cs-
students.stanford.edu/~amitp/game-programming/grids/
10: (0)             .. _Triangular Tiling: https://en.wikipedia.org/wiki/Triangular_tiling
11: (0)             """
12: (0)             from itertools import repeat
13: (0)             from math import sqrt
14: (0)             import networkx as nx
15: (0)             from networkx.classes import set_node_attributes
16: (0)             from networkx.exception import NetworkXError
17: (0)             from networkx.generators.classic import cycle_graph, empty_graph, path_graph
18: (0)             from networkx.relabel import relabel_nodes
19: (0)             from networkx.utils import flatten, nodes_or_number, pairwise
20: (0)             __all__ = [
21: (4)                 "grid_2d_graph",
22: (4)                 "grid_graph",
23: (4)                 "hypercube_graph",
24: (4)                 "triangular_lattice_graph",
25: (4)                 "hexagonal_lattice_graph",
26: (0)             ]
27: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
28: (0)             @nodes_or_number([0, 1])
29: (0)             def grid_2d_graph(m, n, periodic=False, create_using=None):
30: (4)                 """Returns the two-dimensional grid graph.
31: (4)                 The grid graph has each node connected to its four nearest neighbors.
32: (4)                 Parameters
33: (4)                 ----------
```

```
    m, n : int or iterable container of nodes
        If an integer, nodes are from `range(n)`.
        If a container, elements become the coordinate of the nodes.
    periodic : bool or iterable
        If `periodic` is True, both dimensions are periodic. If False, none
        are periodic.  If `periodic` is iterable, it should yield 2 bool
        values indicating whether the 1st and 2nd axes, respectively, are
        periodic.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
        Graph type to create. If graph instance, then cleared before
populated.
    Returns
    -------
    NetworkX graph
        The (possibly periodic) grid graph of the specified dimensions.
    """
    G = empty_graph(0, create_using)
    row_name, rows = m
    col_name, cols = n
    G.add_nodes_from((i, j) for i in rows for j in cols)
    G.add_edges_from(((i, j), (pi, j)) for pi, i in pairwise(rows) for j in
cols)
    G.add_edges_from(((i, j), (i, pj)) for i in rows for pj, j in
pairwise(cols))
    try:
        periodic_r, periodic_c = periodic
    except TypeError:
        periodic_r = periodic_c = periodic
    if periodic_r and len(rows) > 2:
        first = rows[0]
        last = rows[-1]
        G.add_edges_from(((first, j), (last, j)) for j in cols)
    if periodic_c and len(cols) > 2:
        first = cols[0]
        last = cols[-1]
        G.add_edges_from(((i, first), (i, last)) for i in rows)
    if G.is_directed():
        G.add_edges_from((v, u) for u, v in G.edges())
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def grid_graph(dim, periodic=False):
    """Returns the *n*-dimensional grid graph.
    The dimension *n* is the length of the list `dim` and the size in
    each dimension is the value of the corresponding list element.
    Parameters
    ----------
    dim : list or tuple of numbers or iterables of nodes
        'dim' is a tuple or list with, for each dimension, either a number
        that is the size of that dimension or an iterable of nodes for
        that dimension. The dimension of the grid_graph is the length
        of `dim`.
    periodic : bool or iterable
        If `periodic` is True, all dimensions are periodic. If False all
        dimensions are not periodic. If `periodic` is iterable, it should
        yield `dim` bool values each of which indicates whether the
        corresponding axis is periodic.
    Returns
    -------
    NetworkX graph
        The (possibly periodic) grid graph of the specified dimensions.
    Examples
    --------
    To produce a 2 by 3 by 4 grid graph, a graph on 24 nodes:
    >>> from networkx import grid_graph
    >>> G = grid_graph(dim=(2, 3, 4))
    >>> len(G)
    24
    >>> G = grid_graph(dim=(range(7, 9), range(3, 6)))
    >>> len(G)
```

```
100: (4)                6
101: (4)                """
102: (4)                from networkx.algorithms.operators.product import cartesian_product
103: (4)                if not dim:
104: (8)                    return empty_graph(0)
105: (4)                try:
106: (8)                    func = (cycle_graph if p else path_graph for p in periodic)
107: (4)                except TypeError:
108: (8)                    func = repeat(cycle_graph if periodic else path_graph)
109: (4)                G = next(func)(dim[0])
110: (4)                for current_dim in dim[1:]:
111: (8)                    Gnew = next(func)(current_dim)
112: (8)                    G = cartesian_product(Gnew, G)
113: (4)                H = relabel_nodes(G, flatten)
114: (4)                return H
115: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
116: (0)            def hypercube_graph(n):
117: (4)                """Returns the *n*-dimensional hypercube graph.
118: (4)                The nodes are the integers between 0 and ``2 ** n - 1``, inclusive.
119: (4)                For more information on the hypercube graph, see the Wikipedia
120: (4)                article `Hypercube graph`_.
121: (4)                .. _Hypercube graph: https://en.wikipedia.org/wiki/Hypercube_graph
122: (4)                Parameters
123: (4)                ----------
124: (4)                n : int
125: (8)                    The dimension of the hypercube.
126: (8)                    The number of nodes in the graph will be ``2 ** n``.
127: (4)                Returns
128: (4)                -------
129: (4)                NetworkX graph
130: (8)                    The hypercube graph of dimension *n*.
131: (4)                """
132: (4)                dim = n * [2]
133: (4)                G = grid_graph(dim)
134: (4)                return G
135: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
136: (0)            def triangular_lattice_graph(
137: (4)                m, n, periodic=False, with_positions=True, create_using=None
138: (0)            ):
139: (4)                r"""Returns the $m$ by $n$ triangular lattice graph.
140: (4)                The `triangular lattice graph`_ is a two-dimensional `grid graph`_ in
141: (4)                which each square unit has a diagonal edge (each grid unit has a chord).
142: (4)                The returned graph has $m$ rows and $n$ columns of triangles. Rows and
143: (4)                columns include both triangles pointing up and down. Rows form a strip
144: (4)                of constant height. Columns form a series of diamond shapes, staggered
145: (4)                with the columns on either side. Another way to state the size is that
146: (4)                the nodes form a grid of `m+1` rows and `(n + 1) // 2` columns.
147: (4)                The odd row nodes are shifted horizontally relative to the even rows.
148: (4)                Directed graph types have edges pointed up or right.
149: (4)                Positions of nodes are computed by default or `with_positions is True`.
150: (4)                The position of each node (embedded in a euclidean plane) is stored in
151: (4)                the graph using equilateral triangles with sidelength 1.
152: (4)                The height between rows of nodes is thus $\sqrt(3)/2$.
153: (4)                Nodes lie in the first quadrant with the node $(0, 0)$ at the origin.
154: (4)                .. _triangular lattice graph:
http://mathworld.wolfram.com/TriangularGrid.html
155: (4)                .. _grid graph: http://www-cs-students.stanford.edu/~amitp/game-
programming/grids/
156: (4)                .. _Triangular Tiling: https://en.wikipedia.org/wiki/Triangular_tiling
157: (4)                Parameters
158: (4)                ----------
159: (4)                m : int
160: (8)                    The number of rows in the lattice.
161: (4)                n : int
162: (8)                    The number of columns in the lattice.
163: (4)                periodic : bool (default: False)
164: (8)                    If True, join the boundary vertices of the grid using periodic
165: (8)                    boundary conditions. The join between boundaries is the final row
166: (8)                    and column of triangles. This means there is one row and one column
```

```
167: (8)                    fewer nodes for the periodic lattice. Periodic lattices require
168: (8)                    `m >= 3`, `n >= 5` and are allowed but misaligned if `m` or `n` are
odd
169: (4)                with_positions : bool (default: True)
170: (8)                    Store the coordinates of each node in the graph node attribute 'pos'.
171: (8)                    The coordinates provide a lattice with equilateral triangles.
172: (8)                    Periodic positions shift the nodes vertically in a nonlinear way so
173: (8)                    the edges don't overlap so much.
174: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
175: (8)                    Graph type to create. If graph instance, then cleared before
populated.
176: (4)                Returns
177: (4)                -------
178: (4)                NetworkX graph
179: (8)                    The *m* by *n* triangular lattice graph.
180: (4)                """
181: (4)                H = empty_graph(0, create_using)
182: (4)                if n == 0 or m == 0:
183: (8)                    return H
184: (4)                if periodic:
185: (8)                    if n < 5 or m < 3:
186: (12)                       msg = f"m > 2 and n > 4 required for periodic. m={m}, n={n}"
187: (12)                       raise NetworkXError(msg)
188: (4)                N = (n + 1) // 2  # number of nodes in row
189: (4)                rows = range(m + 1)
190: (4)                cols = range(N + 1)
191: (4)                H.add_edges_from(((i, j), (i + 1, j)) for j in rows for i in cols[:N])
192: (4)                H.add_edges_from(((i, j), (i, j + 1)) for j in rows[:m] for i in cols)
193: (4)                H.add_edges_from(((i, j), (i + 1, j + 1)) for j in rows[1:m:2] for i in
cols[:N])
194: (4)                H.add_edges_from(((i + 1, j), (i, j + 1)) for j in rows[:m:2] for i in
cols[:N])
195: (4)                from networkx.algorithms.minors import contracted_nodes
196: (4)                if periodic is True:
197: (8)                    for i in cols:
198: (12)                       H = contracted_nodes(H, (i, 0), (i, m))
199: (8)                    for j in rows[:m]:
200: (12)                       H = contracted_nodes(H, (0, j), (N, j))
201: (4)                elif n % 2:
202: (8)                    H.remove_nodes_from((N, j) for j in rows[1::2])
203: (4)                if with_positions:
204: (8)                    ii = (i for i in cols for j in rows)
205: (8)                    jj = (j for i in cols for j in rows)
206: (8)                    xx = (0.5 * (j % 2) + i for i in cols for j in rows)
207: (8)                    h = sqrt(3) / 2
208: (8)                    if periodic:
209: (12)                       yy = (h * j + 0.01 * i * i for i in cols for j in rows)
210: (8)                    else:
211: (12)                       yy = (h * j for i in cols for j in rows)
212: (8)                    pos = {(i, j): (x, y) for i, j, x, y in zip(ii, jj, xx, yy) if (i, j)
in H}
213: (8)                    set_node_attributes(H, pos, "pos")
214: (4)                return H
215: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
216: (0)            def hexagonal_lattice_graph(
217: (4)                m, n, periodic=False, with_positions=True, create_using=None
218: (0)            ):
219: (4)                """Returns an `m` by `n` hexagonal lattice graph.
220: (4)                The *hexagonal lattice graph* is a graph whose nodes and edges are
221: (4)                the `hexagonal tiling`_ of the plane.
222: (4)                The returned graph will have `m` rows and `n` columns of hexagons.
223: (4)                `Odd numbered columns`_ are shifted up relative to even numbered columns.
224: (4)                Positions of nodes are computed by default or `with_positions is True`.
225: (4)                Node positions creating the standard embedding in the plane
226: (4)                with sidelength 1 and are stored in the node attribute 'pos'.
227: (4)                `pos = nx.get_node_attributes(G, 'pos')` creates a dict ready for drawing.
228: (4)                .. _hexagonal tiling: https://en.wikipedia.org/wiki/Hexagonal_tiling
229: (4)                .. _Odd numbered columns: http://www-cs-students.stanford.edu/~amitp/game-
programming/grids/
```

```
    Parameters
    ----------
    m : int
        The number of rows of hexagons in the lattice.
    n : int
        The number of columns of hexagons in the lattice.
    periodic : bool
        Whether to make a periodic grid by joining the boundary vertices.
        For this to work `n` must be even and both `n > 1` and `m > 1`.
        The periodic connections create another row and column of hexagons
        so these graphs have fewer nodes as boundary nodes are identified.
    with_positions : bool (default: True)
        Store the coordinates of each node in the graph node attribute 'pos'.
        The coordinates provide a lattice with vertical columns of hexagons
        offset to interleave and cover the plane.
        Periodic positions shift the nodes vertically in a nonlinear way so
        the edges don't overlap so much.
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
        Graph type to create. If graph instance, then cleared before populated.

        If graph is directed, edges will point up or right.
    Returns
    -------
    NetworkX graph
        The *m* by *n* hexagonal lattice graph.
    """
    G = empty_graph(0, create_using)
    if m == 0 or n == 0:
        return G
    if periodic and (n % 2 == 1 or m < 2 or n < 2):
        msg = "periodic hexagonal lattice needs m > 1, n > 1 and even n"
        raise NetworkXError(msg)
    M = 2 * m  # twice as many nodes as hexagons vertically
    rows = range(M + 2)
    cols = range(n + 1)
    col_edges = (((i, j), (i, j + 1)) for i in cols for j in rows[: M + 1])
    row_edges = (((i, j), (i + 1, j)) for i in cols[:n] for j in rows if i % 2 == j % 2)
    G.add_edges_from(col_edges)
    G.add_edges_from(row_edges)
    G.remove_node((0, M + 1))
    G.remove_node((n, (M + 1) * (n % 2)))
    from networkx.algorithms.minors import contracted_nodes
    if periodic:
        for i in cols[:n]:
            G = contracted_nodes(G, (i, 0), (i, M))
        for i in cols[1:]:
            G = contracted_nodes(G, (i, 1), (i, M + 1))
        for j in rows[1:M]:
            G = contracted_nodes(G, (0, j), (n, j))
        G.remove_node((n, M))
    ii = (i for i in cols for j in rows)
    jj = (j for i in cols for j in rows)
    xx = (0.5 + i + i // 2 + (j % 2) * ((i % 2) - 0.5) for i in cols for j in rows)
    h = sqrt(3) / 2
    if periodic:
        yy = (h * j + 0.01 * i * i for i in cols for j in rows)
    else:
        yy = (h * j for i in cols for j in rows)
    pos = {(i, j): (x, y) for i, j, x, y in zip(ii, jj, xx, yy) if (i, j) in G}
    set_node_attributes(G, pos, "pos")
    return G

----------------------------------------

File 443 - line.py:
```

```
(0)  """Functions for generating line graphs."""
(0)  from collections import defaultdict
(0)  from functools import partial
(0)  from itertools import combinations
(0)  import networkx as nx
(0)  from networkx.utils import arbitrary_element
(0)  from networkx.utils.decorators import not_implemented_for
(0)  __all__ = ["line_graph", "inverse_line_graph"]
(0)  @nx._dispatchable(returns_graph=True)
(0)  def line_graph(G, create_using=None):
(4)      r"""Returns the line graph of the graph or digraph `G`.
(4)      The line graph of a graph `G` has a node for each edge in `G` and an
(4)      edge joining those nodes if the two edges in `G` share a common node. For
(4)      directed graphs, nodes are adjacent exactly when the edges they represent
(4)      form a directed path of length two.
(4)      The nodes of the line graph are 2-tuples of nodes in the original graph (or
(4)      3-tuples for multigraphs, with the key of the edge as the third element).
(4)      For information about self-loops and more discussion, see the **Notes**
(4)      section below.
(4)      Parameters
(4)      ----------
(4)      G : graph
(8)          A NetworkX Graph, DiGraph, MultiGraph, or MultiDigraph.
(4)      create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)         Graph type to create. If graph instance, then cleared before populated.
(4)      Returns
(4)      -------
(4)      L : graph
(8)          The line graph of G.
(4)      Examples
(4)      --------
(4)      >>> G = nx.star_graph(3)
(4)      >>> L = nx.line_graph(G)
(4)      >>> print(sorted(map(sorted, L.edges())))  # makes a 3-clique, K3
(4)      [[(0, 1), (0, 2)], [(0, 1), (0, 3)], [(0, 2), (0, 3)]]
(4)      Edge attributes from `G` are not copied over as node attributes in `L`, but
(4)      attributes can be copied manually:
(4)      >>> G = nx.path_graph(4)
(4)      >>> G.add_edges_from((u, v, {"tot": u + v}) for u, v in G.edges)
(4)      >>> G.edges(data=True)
(4)      EdgeDataView([(0, 1, {'tot': 1}), (1, 2, {'tot': 3}), (2, 3, {'tot': 5})])
(4)      >>> H = nx.line_graph(G)
(4)      >>> H.add_nodes_from((node, G.edges[node]) for node in H)
(4)      >>> H.nodes(data=True)
(4)      NodeDataView({(0, 1): {'tot': 1}, (2, 3): {'tot': 5}, (1, 2): {'tot': 3}})
(4)      Notes
(4)      -----
(4)      Graph, node, and edge data are not propagated to the new graph. For
(4)      undirected graphs, the nodes in G must be sortable, otherwise the
(4)      constructed line graph may not be correct.
(4)      *Self-loops in undirected graphs*
(4)      For an undirected graph `G` without multiple edges, each edge can be
(4)      written as a set `\{u, v\}`.  Its line graph `L` has the edges of `G` as
(4)      its nodes. If `x` and `y` are two nodes in `L`, then `\{x, y\}` is an edge
(4)      in `L` if and only if the intersection of `x` and `y` is nonempty. Thus,
(4)      the set of all edges is determined by the set of all pairwise intersections
(4)      of edges in `G`.
(4)      Trivially, every edge in G would have a nonzero intersection with itself,
(4)      and so every node in `L` should have a self-loop. This is not so
(4)      interesting, and the original context of line graphs was with simple
(4)      graphs, which had no self-loops or multiple edges. The line graph was also
(4)      meant to be a simple graph and thus, self-loops in `L` are not part of the
(4)      standard definition of a line graph. In a pairwise intersection matrix,
(4)      this is analogous to excluding the diagonal entries from the line graph
(4)      definition.
(4)      Self-loops and multiple edges in `G` add nodes to `L` in a natural way,
```

```
and
67: (4)                 do not require any fundamental changes to the definition. It might be
68: (4)                 argued that the self-loops we excluded before should now be included.
69: (4)                 However, the self-loops are still "trivial" in some sense and thus, are
70: (4)                 usually excluded.
71: (4)                 *Self-loops in directed graphs*
72: (4)                 For a directed graph `G` without multiple edges, each edge can be written
73: (4)                 as a tuple `(u, v)`. Its line graph `L` has the edges of `G` as its
74: (4)                 nodes. If `x` and `y` are two nodes in `L`, then `(x, y)` is an edge in
`L`
75: (4)                 if and only if the tail of `x` matches the head of `y`, for example, if `x
76: (4)                 = (a, b)` and `y = (b, c)` for some vertices `a`, `b`, and `c` in `G`.
77: (4)                 Due to the directed nature of the edges, it is no longer the case that
78: (4)                 every edge in `G` should have a self-loop in `L`. Now, the only time
79: (4)                 self-loops arise is if a node in `G` itself has a self-loop.  So such
80: (4)                 self-loops are no longer "trivial" but instead, represent essential
81: (4)                 features of the topology of `G`. For this reason, the historical
82: (4)                 development of line digraphs is such that self-loops are included. When
the
83: (4)                 graph `G` has multiple edges, once again only superficial changes are
84: (4)                 required to the definition.
85: (4)                 References
86: (4)                 ----------
87: (4)                 * Harary, Frank, and Norman, Robert Z., "Some properties of line
digraphs",
88: (6)                   Rend. Circ. Mat. Palermo, II. Ser. 9 (1960), 161--168.
89: (4)                 * Hemminger, R. L.; Beineke, L. W. (1978), "Line graphs and line
digraphs",
90: (6)                   in Beineke, L. W.; Wilson, R. J., Selected Topics in Graph Theory,
91: (6)                   Academic Press Inc., pp. 271--305.
92: (4)                 """
93: (4)                 if G.is_directed():
94: (8)                     L = _lg_directed(G, create_using=create_using)
95: (4)                 else:
96: (8)                     L = _lg_undirected(G, selfloops=False, create_using=create_using)
97: (4)                 return L
98: (0)             def _lg_directed(G, create_using=None):
99: (4)                 """Returns the line graph L of the (multi)digraph G.
100: (4)                Edges in G appear as nodes in L, represented as tuples of the form (u,v)
101: (4)                or (u,v,key) if G is a multidigraph. A node in L corresponding to the edge
102: (4)                (u,v) is connected to every node corresponding to an edge (v,w).
103: (4)                Parameters
104: (4)                ----------
105: (4)                G : digraph
106: (8)                    A directed graph or directed multigraph.
107: (4)                create_using : NetworkX graph constructor, optional
108: (7)                   Graph type to create. If graph instance, then cleared before populated.
109: (7)                   Default is to use the same graph class as `G`.
110: (4)                """
111: (4)                L = nx.empty_graph(0, create_using, default=G.__class__)
112: (4)                get_edges = partial(G.edges, keys=True) if G.is_multigraph() else G.edges
113: (4)                for from_node in get_edges():
114: (8)                    L.add_node(from_node)
115: (8)                    for to_node in get_edges(from_node[1]):
116: (12)                       L.add_edge(from_node, to_node)
117: (4)                return L
118: (0)            def _lg_undirected(G, selfloops=False, create_using=None):
119: (4)                """Returns the line graph L of the (multi)graph G.
120: (4)                Edges in G appear as nodes in L, represented as sorted tuples of the form
121: (4)                (u,v), or (u,v,key) if G is a multigraph. A node in L corresponding to
122: (4)                the edge {u,v} is connected to every node corresponding to an edge that
123: (4)                involves u or v.
124: (4)                Parameters
125: (4)                ----------
126: (4)                G : graph
127: (8)                    An undirected graph or multigraph.
128: (4)                selfloops : bool
129: (8)                    If `True`, then self-loops are included in the line graph. If `False`,
130: (8)                    they are excluded.
```

```
131: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
132: (7)                   Graph type to create. If graph instance, then cleared before populated.
133: (4)                Notes
134: (4)                -----
135: (4)                The standard algorithm for line graphs of undirected graphs does not
136: (4)                produce self-loops.
137: (4)                """
138: (4)                L = nx.empty_graph(0, create_using, default=G.__class__)
139: (4)                get_edges = partial(G.edges, keys=True) if G.is_multigraph() else G.edges
140: (4)                shift = 0 if selfloops else 1
141: (4)                node_index = {n: i for i, n in enumerate(G)}
142: (4)                edge_key_function = lambda edge: (node_index[edge[0]],
node_index[edge[1]])
143: (4)                edges = set()
144: (4)                for u in G:
145: (8)                    nodes = [tuple(sorted(x[:2], key=node_index.get)) + x[2:] for x in
get_edges(u)]
146: (8)                    if len(nodes) == 1:
147: (12)                       L.add_node(nodes[0])
148: (8)                    for i, a in enumerate(nodes):
149: (12)                       edges.update(
150: (16)                           [
151: (20)                               tuple(sorted((a, b), key=edge_key_function))
152: (20)                               for b in nodes[i + shift :]
153: (16)                           ]
154: (12)                       )
155: (4)                L.add_edges_from(edges)
156: (4)                return L
157: (0)            @not_implemented_for("directed")
158: (0)            @not_implemented_for("multigraph")
159: (0)            @nx._dispatchable(returns_graph=True)
160: (0)            def inverse_line_graph(G):
161: (4)                """Returns the inverse line graph of graph G.
162: (4)                If H is a graph, and G is the line graph of H, such that G = L(H).
163: (4)                Then H is the inverse line graph of G.
164: (4)                Not all graphs are line graphs and these do not have an inverse line
graph.
165: (4)                In these cases this function raises a NetworkXError.
166: (4)                Parameters
167: (4)                ----------
168: (4)                G : graph
169: (8)                    A NetworkX Graph
170: (4)                Returns
171: (4)                -------
172: (4)                H : graph
173: (8)                    The inverse line graph of G.
174: (4)                Raises
175: (4)                ------
176: (4)                NetworkXNotImplemented
177: (8)                    If G is directed or a multigraph
178: (4)                NetworkXError
179: (8)                    If G is not a line graph
180: (4)                Notes
181: (4)                -----
182: (4)                This is an implementation of the Roussopoulos algorithm[1]_.
183: (4)                If G consists of multiple components, then the algorithm doesn't work.
184: (4)                You should invert every component separately:
185: (4)                >>> K5 = nx.complete_graph(5)
186: (4)                >>> P4 = nx.Graph([("a", "b"), ("b", "c"), ("c", "d")])
187: (4)                >>> G = nx.union(K5, P4)
188: (4)                >>> root_graphs = []
189: (4)                >>> for comp in nx.connected_components(G):
190: (4)                ...     root_graphs.append(nx.inverse_line_graph(G.subgraph(comp)))
191: (4)                >>> len(root_graphs)
192: (4)                2
193: (4)                References
194: (4)                ----------
195: (4)                .. [1] Roussopoulos, N.D. , "A max {m, n} algorithm for determining the
graph H from
```

```
(7)                 its line graph G", Information Processing Letters 2, (1973), 108--112, ISSN 0020-0190,
(7)                 `DOI link <https://doi.org/10.1016/0020-0190(73)90029-X>`_
(4)             """
(4)             if G.number_of_nodes() == 0:
(8)                 return nx.empty_graph(1)
(4)             elif G.number_of_nodes() == 1:
(8)                 v = arbitrary_element(G)
(8)                 a = (v, 0)
(8)                 b = (v, 1)
(8)                 H = nx.Graph([(a, b)])
(8)                 return H
(4)             elif G.number_of_nodes() > 1 and G.number_of_edges() == 0:
(8)                 msg = (
(12)                    "inverse_line_graph() doesn't work on an edgeless graph. "
(12)                    "Please use this function on each component separately."
(8)                 )
(8)                 raise nx.NetworkXError(msg)
(4)             if nx.number_of_selfloops(G) != 0:
(8)                 msg = (
(12)                    "A line graph as generated by NetworkX has no selfloops, so G has no "
(12)                    "inverse line graph. Please remove the selfloops from G and try again."
(8)                 )
(8)                 raise nx.NetworkXError(msg)
(4)             starting_cell = _select_starting_cell(G)
(4)             P = _find_partition(G, starting_cell)
(4)             P_count = {u: 0 for u in G.nodes}
(4)             for p in P:
(8)                 for u in p:
(12)                    P_count[u] += 1
(4)             if max(P_count.values()) > 2:
(8)                 msg = "G is not a line graph (vertex found in more than two partition cells)"
(8)                 raise nx.NetworkXError(msg)
(4)             W = tuple((u,) for u in P_count if P_count[u] == 1)
(4)             H = nx.Graph()
(4)             H.add_nodes_from(P)
(4)             H.add_nodes_from(W)
(4)             for a, b in combinations(H.nodes, 2):
(8)                 if any(a_bit in b for a_bit in a):
(12)                    H.add_edge(a, b)
(4)             return H
(0)         def _triangles(G, e):
(4)             """Return list of all triangles containing edge e"""
(4)             u, v = e
(4)             if u not in G:
(8)                 raise nx.NetworkXError(f"Vertex {u} not in graph")
(4)             if v not in G[u]:
(8)                 raise nx.NetworkXError(f"Edge ({u}, {v}) not in graph")
(4)             triangle_list = []
(4)             for x in G[u]:
(8)                 if x in G[v]:
(12)                    triangle_list.append((u, v, x))
(4)             return triangle_list
(0)         def _odd_triangle(G, T):
(4)             """Test whether T is an odd triangle in G
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(4)             T : 3-tuple of vertices forming triangle in G
(4)             Returns
(4)             -------
(4)             True is T is an odd triangle
(4)             False otherwise
(4)             Raises
(4)             ------
(4)             NetworkXError
```

```
(8)                 T is not a triangle in G
(4)             Notes
(4)             -----
(4)             An odd triangle is one in which there exists another vertex in G which is
(4)             adjacent to either exactly one or exactly all three of the vertices in the
(4)             triangle.
(4)             """
(4)             for u in T:
(8)                 if u not in G.nodes():
(12)                    raise nx.NetworkXError(f"Vertex {u} not in graph")
(4)             for e in list(combinations(T, 2)):
(8)                 if e[0] not in G[e[1]]:
(12)                    raise nx.NetworkXError(f"Edge ({e[0]}, {e[1]}) not in graph")
(4)             T_nbrs = defaultdict(int)
(4)             for t in T:
(8)                 for v in G[t]:
(12)                    if v not in T:
(16)                        T_nbrs[v] += 1
(4)             return any(T_nbrs[v] in [1, 3] for v in T_nbrs)
(0)         def _find_partition(G, starting_cell):
(4)             """Find a partition of the vertices of G into cells of complete graphs
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(4)             starting_cell : tuple of vertices in G which form a cell
(4)             Returns
(4)             -------
(4)             List of tuples of vertices of G
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If a cell is not a complete subgraph then G is not a line graph
(4)             """
(4)             G_partition = G.copy()
(4)             P = [starting_cell]  # partition set
(4)             G_partition.remove_edges_from(list(combinations(starting_cell, 2)))
(4)             partitioned_vertices = list(starting_cell)
(4)             while G_partition.number_of_edges() > 0:
(8)                 u = partitioned_vertices.pop()
(8)                 deg_u = len(G_partition[u])
(8)                 if deg_u != 0:
(12)                    new_cell = [u] + list(G_partition[u])
(12)                    for u in new_cell:
(16)                        for v in new_cell:
(20)                            if (u != v) and (v not in G_partition[u]):
(24)                                msg = (
(28)                                    "G is not a line graph "
(28)                                    "(partition cell not a complete subgraph)"
(24)                                )
(24)                                raise nx.NetworkXError(msg)
(12)                    P.append(tuple(new_cell))
(12)                    G_partition.remove_edges_from(list(combinations(new_cell, 2)))
(12)                    partitioned_vertices += new_cell
(4)             return P
(0)         def _select_starting_cell(G, starting_edge=None):
(4)             """Select a cell to initiate _find_partition
(4)             Parameters
(4)             ----------
(4)             G : NetworkX Graph
(4)             starting_edge: an edge to build the starting cell from
(4)             Returns
(4)             -------
(4)             Tuple of vertices in G
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If it is determined that G is not a line graph
(4)             Notes
(4)             -----
```

```
330: (4)                If starting edge not specified then pick an arbitrary edge - doesn't
331: (4)                matter which. However, this function may call itself requiring a
332: (4)                specific starting edge. Note that the r, s notation for counting
333: (4)                triangles is the same as in the Roussopoulos paper cited above.
334: (4)                """
335: (4)                if starting_edge is None:
336: (8)                    e = arbitrary_element(G.edges())
337: (4)                else:
338: (8)                    e = starting_edge
339: (8)                    if e[0] not in G.nodes():
340: (12)                       raise nx.NetworkXError(f"Vertex {e[0]} not in graph")
341: (8)                    if e[1] not in G[e[0]]:
342: (12)                       msg = f"starting_edge ({e[0]}, {e[1]}) is not in the Graph"
343: (12)                       raise nx.NetworkXError(msg)
344: (4)                e_triangles = _triangles(G, e)
345: (4)                r = len(e_triangles)
346: (4)                if r == 0:
347: (8)                    starting_cell = e
348: (4)                elif r == 1:
349: (8)                    T = e_triangles[0]
350: (8)                    a, b, c = T
351: (8)                    ac_edges = len(_triangles(G, (a, c)))
352: (8)                    bc_edges = len(_triangles(G, (b, c)))
353: (8)                    if ac_edges == 1:
354: (12)                       if bc_edges == 1:
355: (16)                           starting_cell = T
356: (12)                       else:
357: (16)                           return _select_starting_cell(G, starting_edge=(b, c))
358: (8)                    else:
359: (12)                       return _select_starting_cell(G, starting_edge=(a, c))
360: (4)                else:
361: (8)                    s = 0
362: (8)                    odd_triangles = []
363: (8)                    for T in e_triangles:
364: (12)                       if _odd_triangle(G, T):
365: (16)                           s += 1
366: (16)                           odd_triangles.append(T)
367: (8)                    if r == 2 and s == 0:
368: (12)                       starting_cell = T
369: (8)                    elif r - 1 <= s <= r:
370: (12)                       triangle_nodes = set()
371: (12)                       for T in odd_triangles:
372: (16)                           for x in T:
373: (20)                               triangle_nodes.add(x)
374: (12)                       for u in triangle_nodes:
375: (16)                           for v in triangle_nodes:
376: (20)                               if u != v and (v not in G[u]):
377: (24)                                   msg = (
378: (28)                                       "G is not a line graph (odd triangles "
379: (28)                                       "do not form complete subgraph)"
380: (24)                                   )
381: (24)                                   raise nx.NetworkXError(msg)
382: (12)                       starting_cell = tuple(triangle_nodes)
383: (8)                    else:
384: (12)                       msg = (
385: (16)                           "G is not a line graph (incorrect number of "
386: (16)                           "odd triangles around starting edge)"
387: (12)                       )
388: (12)                       raise nx.NetworkXError(msg)
389: (4)                return starting_cell
```

File 444 - mycielski.py:

```
1: (0)              """Functions related to the Mycielski Operation and the Mycielskian family
2: (0)              of graphs.
3: (0)              """
4: (0)              import networkx as nx
```

```
from networkx.utils import not_implemented_for
__all__ = ["mycielskian", "mycielski_graph"]
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@nx._dispatchable(returns_graph=True)
def mycielskian(G, iterations=1):
    r"""Returns the Mycielskian of a simple, undirected graph G
    The Mycielskian of graph preserves a graph's triangle free
    property while increasing the chromatic number by 1.
    The Mycielski Operation on a graph, :math:`G=(V, E)`, constructs a new
    graph with :math:`2|V| + 1` nodes and :math:`3|E| + |V|` edges.
    The construction is as follows:
    Let :math:`V = {0, ..., n-1}`. Construct another vertex set
    :math:`U = {n, ..., 2n}` and a vertex, `w`.
    Construct a new graph, `M`, with vertices :math:`U \bigcup V \bigcup w`.
    For edges, :math:`(u, v) \in E` add edges :math:`(u, v), (u, v + n)`, and
    :math:`(u + n, v)` to M. Finally, for all vertices :math:`u \in U`, add
    edge :math:`(u, w)` to M.
    The Mycielski Operation can be done multiple times by repeating the above
    process iteratively.
    More information can be found at https://en.wikipedia.org/wiki/Mycielskian
    Parameters
    ----------
    G : graph
        A simple, undirected NetworkX graph
    iterations : int
        The number of iterations of the Mycielski operation to
        perform on G. Defaults to 1. Must be a non-negative integer.
    Returns
    -------
    M : graph
        The Mycielskian of G after the specified number of iterations.
    Notes
    -----
    Graph, node, and edge data are not necessarily propagated to the new graph.
    """
    M = nx.convert_node_labels_to_integers(G)
    for i in range(iterations):
        n = M.number_of_nodes()
        M.add_nodes_from(range(n, 2 * n))
        old_edges = list(M.edges())
        M.add_edges_from((u, v + n) for u, v in old_edges)
        M.add_edges_from((u + n, v) for u, v in old_edges)
        M.add_node(2 * n)
        M.add_edges_from((u + n, 2 * n) for u in range(n))
    return M
@nx._dispatchable(graphs=None, returns_graph=True)
def mycielski_graph(n):
    """Generator for the n_th Mycielski Graph.
    The Mycielski family of graphs is an infinite set of graphs.
    :math:`M_1` is the singleton graph, :math:`M_2` is two vertices with an
    edge, and, for :math:`i > 2`, :math:`M_i` is the Mycielskian of
    :math:`M_{i-1}`.
    More information can be found at
    http://mathworld.wolfram.com/MycielskiGraph.html
    Parameters
    ----------
    n : int
        The desired Mycielski Graph.
    Returns
    -------
    M : graph
        The n_th Mycielski Graph
    Notes
    -----
    The first graph in the Mycielski sequence is the singleton graph.
    The Mycielskian of this graph is not the :math:`P_2` graph, but rather the
    :math:`P_2` graph with an extra, isolated vertex. The second Mycielski
```

```
73: (4)                 graph is the :math:`P_2` graph, so the first two are hard coded.
74: (4)                 The remaining graphs are generated using the Mycielski operation.
75: (4)                 """
76: (4)                 if n < 1:
77: (8)                     raise nx.NetworkXError("must satisfy n >= 1")
78: (4)                 if n == 1:
79: (8)                     return nx.empty_graph(1)
80: (4)                 else:
81: (8)                     return mycielskian(nx.path_graph(2), n - 2)

----------------------------------------

File 445 - nonisomorphic_trees.py:

1: (0)              """
2: (0)              Implementation of the Wright, Richmond, Odlyzko and McKay (WROM)
3: (0)              algorithm for the enumeration of all non-isomorphic free trees of a
4: (0)              given order.  Rooted trees are represented by level sequences, i.e.,
5: (0)              lists in which the i-th element specifies the distance of vertex i to
6: (0)              the root.
7: (0)              """
8: (0)              __all__ = ["nonisomorphic_trees", "number_of_nonisomorphic_trees"]
9: (0)              import networkx as nx
10: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
11: (0)             def nonisomorphic_trees(order, create="graph"):
12: (4)                 """Generates lists of nonisomorphic trees
13: (4)                 Parameters
14: (4)                 ----------
15: (4)                 order : int
16: (7)                    order of the desired tree(s)
17: (4)                 create : one of {"graph", "matrix"} (default="graph")
18: (7)                    If ``"graph"`` is selected a list of ``Graph`` instances will be
returned,
19: (7)                    if matrix is selected a list of adjacency matrices will be returned.
20: (7)                    .. deprecated:: 3.3
21: (10)                      The `create` argument is deprecated and will be removed in NetworkX
22: (10)                      version 3.5. In the future, `nonisomorphic_trees` will yield graph
23: (10)                      instances by default. To generate adjacency matrices, call
24: (10)                      ``nx.to_numpy_array`` on the output, e.g.::
25: (13)                         [nx.to_numpy_array(G) for G in nx.nonisomorphic_trees(N)]
26: (4)                 Yields
27: (4)                 ------
28: (4)                 list
29: (7)                    A list of nonisomorphic trees, in one of two formats depending on the
30: (7)                    value of the `create` parameter:
31: (7)                    - ``create="graph"``: yields a list of `networkx.Graph` instances
32: (7)                    - ``create="matrix"``: yields a list of list-of-lists representing
adjacency matrices
33: (4)                 """
34: (4)                 if order < 2:
35: (8)                     raise ValueError
36: (4)                 layout = list(range(order // 2 + 1)) + list(range(1, (order + 1) // 2))
37: (4)                 while layout is not None:
38: (8)                     layout = _next_tree(layout)
39: (8)                     if layout is not None:
40: (12)                        if create == "graph":
41: (16)                            yield _layout_to_graph(layout)
42: (12)                        elif create == "matrix":
43: (16)                            import warnings
44: (16)                            warnings.warn(
45: (20)                                (
46: (24)                                    "\n\nThe 'create=matrix' argument of
nonisomorphic_trees\n"
47: (24)                                    "is deprecated and will be removed in version 3.5.\n"
48: (24)                                    "Use ``nx.to_numpy_array`` to convert graphs to
adjacency "
49: (24)                                    "matrices, e.g.::\n\n"
50: (24)                                    "   [nx.to_numpy_array(G) for G in
nx.nonisomorphic_trees(N)]"
```

```
(20)                                ),
(20)                                category=DeprecationWarning,
(20)                                stacklevel=2,
(16)                            )
(16)                            yield _layout_to_matrix(layout)
(12)                        layout = _next_rooted_tree(layout)
(0)             @nx._dispatchable(graphs=None)
(0)             def number_of_nonisomorphic_trees(order):
(4)                 """Returns the number of nonisomorphic trees
(4)                 Parameters
(4)                 ----------
(4)                 order : int
(6)                   order of the desired tree(s)
(4)                 Returns
(4)                 -------
(4)                 length : Number of nonisomorphic graphs for the given order
(4)                 References
(4)                 ----------
(4)                 """
(4)                 return sum(1 for _ in nonisomorphic_trees(order))
(0)             def _next_rooted_tree(predecessor, p=None):
(4)                 """One iteration of the Beyer-Hedetniemi algorithm."""
(4)                 if p is None:
(8)                     p = len(predecessor) - 1
(8)                     while predecessor[p] == 1:
(12)                        p -= 1
(4)                 if p == 0:
(8)                     return None
(4)                 q = p - 1
(4)                 while predecessor[q] != predecessor[p] - 1:
(8)                     q -= 1
(4)                 result = list(predecessor)
(4)                 for i in range(p, len(result)):
(8)                     result[i] = result[i - p + q]
(4)                 return result
(0)             def _next_tree(candidate):
(4)                 """One iteration of the Wright, Richmond, Odlyzko and McKay
(4)                 algorithm."""
(4)                 left, rest = _split_tree(candidate)
(4)                 left_height = max(left)
(4)                 rest_height = max(rest)
(4)                 valid = rest_height >= left_height
(4)                 if valid and rest_height == left_height:
(8)                     if len(left) > len(rest):
(12)                        valid = False
(8)                     elif len(left) == len(rest) and left > rest:
(12)                        valid = False
(4)                 if valid:
(8)                     return candidate
(4)                 else:
(8)                     p = len(left)
(8)                     new_candidate = _next_rooted_tree(candidate, p)
(8)                     if candidate[p] > 2:
(12)                        new_left, new_rest = _split_tree(new_candidate)
(12)                        new_left_height = max(new_left)
(12)                        suffix = range(1, new_left_height + 2)
(12)                        new_candidate[-len(suffix) :] = suffix
(8)                     return new_candidate
(0)             def _split_tree(layout):
(4)                 """Returns a tuple of two layouts, one containing the left
(4)                 subtree of the root vertex, and one containing the original tree
(4)                 with the left subtree removed."""
(4)                 one_found = False
(4)                 m = None
(4)                 for i in range(len(layout)):
(8)                     if layout[i] == 1:
(12)                        if one_found:
(16)                            m = i
(16)                            break
```

```
120: (12)                        else:
121: (16)                            one_found = True
122: (4)                if m is None:
123: (8)                    m = len(layout)
124: (4)                left = [layout[i] - 1 for i in range(1, m)]
125: (4)                rest = [0] + [layout[i] for i in range(m, len(layout))]
126: (4)                return (left, rest)
127: (0)            def _layout_to_matrix(layout):
128: (4)                """Create the adjacency matrix for the tree specified by the
129: (4)                given layout (level sequence)."""
130: (4)                result = [[0] * len(layout) for i in range(len(layout))]
131: (4)                stack = []
132: (4)                for i in range(len(layout)):
133: (8)                    i_level = layout[i]
134: (8)                    if stack:
135: (12)                       j = stack[-1]
136: (12)                       j_level = layout[j]
137: (12)                       while j_level >= i_level:
138: (16)                           stack.pop()
139: (16)                           j = stack[-1]
140: (16)                           j_level = layout[j]
141: (12)                       result[i][j] = result[j][i] = 1
142: (8)                    stack.append(i)
143: (4)                return result
144: (0)            def _layout_to_graph(layout):
145: (4)                """Create a NetworkX Graph for the tree specified by the
146: (4)                given layout(level sequence)"""
147: (4)                G = nx.Graph()
148: (4)                stack = []
149: (4)                for i in range(len(layout)):
150: (8)                    i_level = layout[i]
151: (8)                    if stack:
152: (12)                       j = stack[-1]
153: (12)                       j_level = layout[j]
154: (12)                       while j_level >= i_level:
155: (16)                           stack.pop()
156: (16)                           j = stack[-1]
157: (16)                           j_level = layout[j]
158: (12)                       G.add_edge(i, j)
159: (8)                    stack.append(i)
160: (4)                return G
```

File 446 - random_graphs.py:

```
1: (0)              """
2: (0)              Generators for random graphs.
3: (0)              """
4: (0)              import itertools
5: (0)              import math
6: (0)              from collections import defaultdict
7: (0)              import networkx as nx
8: (0)              from networkx.utils import py_random_state
9: (0)              from .classic import complete_graph, empty_graph, path_graph, star_graph
10: (0)             from .degree_seq import degree_sequence_tree
11: (0)             __all__ = [
12: (4)                 "fast_gnp_random_graph",
13: (4)                 "gnp_random_graph",
14: (4)                 "dense_gnm_random_graph",
15: (4)                 "gnm_random_graph",
16: (4)                 "erdos_renyi_graph",
17: (4)                 "binomial_graph",
18: (4)                 "newman_watts_strogatz_graph",
19: (4)                 "watts_strogatz_graph",
20: (4)                 "connected_watts_strogatz_graph",
21: (4)                 "random_regular_graph",
22: (4)                 "barabasi_albert_graph",
23: (4)                 "dual_barabasi_albert_graph",
```

```
24: (4)                 "extended_barabasi_albert_graph",
25: (4)                 "powerlaw_cluster_graph",
26: (4)                 "random_lobster",
27: (4)                 "random_shell_graph",
28: (4)                 "random_powerlaw_tree",
29: (4)                 "random_powerlaw_tree_sequence",
30: (4)                 "random_kernel_graph",
31: (0)             ]
32: (0)             @py_random_state(2)
33: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
34: (0)             def fast_gnp_random_graph(n, p, seed=None, directed=False):
35: (4)                 """Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or
36: (4)                 a binomial graph.
37: (4)                 Parameters
38: (4)                 ----------
39: (4)                 n : int
40: (8)                     The number of nodes.
41: (4)                 p : float
42: (8)                     Probability for edge creation.
43: (4)                 seed : integer, random_state, or None (default)
44: (8)                     Indicator of random number generation state.
45: (8)                     See :ref:`Randomness<randomness>`.
46: (4)                 directed : bool, optional (default=False)
47: (8)                     If True, this function returns a directed graph.
48: (4)                 Notes
49: (4)                 -----
50: (4)                 The $G_{n,p}$ graph algorithm chooses each of the $[n (n - 1)] / 2$
51: (4)                 (undirected) or $n (n - 1)$ (directed) possible edges with probability
$p$.
52: (4)                 This algorithm [1]_ runs in $O(n + m)$ time, where `m` is the expected
number of
53: (4)                 edges, which equals $p n (n - 1) / 2$. This should be faster than
54: (4)                 :func:`gnp_random_graph` when $p$ is small and the expected number of
edges
55: (4)                 is small (that is, the graph is sparse).
56: (4)                 See Also
57: (4)                 --------
58: (4)                 gnp_random_graph
59: (4)                 References
60: (4)                 ----------
61: (4)                 .. [1] Vladimir Batagelj and Ulrik Brandes,
62: (7)                    "Efficient generation of large random networks",
63: (7)                    Phys. Rev. E, 71, 036113, 2005.
64: (4)                 """
65: (4)                 G = empty_graph(n)
66: (4)                 if p <= 0 or p >= 1:
67: (8)                     return nx.gnp_random_graph(n, p, seed=seed, directed=directed)
68: (4)                 lp = math.log(1.0 - p)
69: (4)                 if directed:
70: (8)                     G = nx.DiGraph(G)
71: (8)                     v = 1
72: (8)                     w = -1
73: (8)                     while v < n:
74: (12)                        lr = math.log(1.0 - seed.random())
75: (12)                        w = w + 1 + int(lr / lp)
76: (12)                        while w >= v and v < n:
77: (16)                            w = w - v
78: (16)                            v = v + 1
79: (12)                        if v < n:
80: (16)                            G.add_edge(w, v)
81: (4)                 v = 1
82: (4)                 w = -1
83: (4)                 while v < n:
84: (8)                     lr = math.log(1.0 - seed.random())
85: (8)                     w = w + 1 + int(lr / lp)
86: (8)                     while w >= v and v < n:
87: (12)                        w = w - v
88: (12)                        v = v + 1
89: (8)                     if v < n:
```

```
90: (12)                        G.add_edge(v, w)
91: (4)                 return G
92: (0)             @py_random_state(2)
93: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
94: (0)             def gnp_random_graph(n, p, seed=None, directed=False):
95: (4)                 """Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph
96: (4)                 or a binomial graph.
97: (4)                 The $G_{n,p}$ model chooses each of the possible edges with probability
$p$.
98: (4)                 Parameters
99: (4)                 ----------
100: (4)                n : int
101: (8)                    The number of nodes.
102: (4)                p : float
103: (8)                    Probability for edge creation.
104: (4)                seed : integer, random_state, or None (default)
105: (8)                    Indicator of random number generation state.
106: (8)                    See :ref:`Randomness<randomness>`.
107: (4)                directed : bool, optional (default=False)
108: (8)                    If True, this function returns a directed graph.
109: (4)                See Also
110: (4)                --------
111: (4)                fast_gnp_random_graph
112: (4)                Notes
113: (4)                -----
114: (4)                This algorithm [2]_ runs in $O(n^2)$ time.  For sparse graphs (that is,
for
115: (4)                small values of $p$), :func:`fast_gnp_random_graph` is a faster algorithm.
116: (4)                :func:`binomial_graph` and :func:`erdos_renyi_graph` are
117: (4)                aliases for :func:`gnp_random_graph`.
118: (4)                >>> nx.binomial_graph is nx.gnp_random_graph
119: (4)                True
120: (4)                >>> nx.erdos_renyi_graph is nx.gnp_random_graph
121: (4)                True
122: (4)                References
123: (4)                ----------
124: (4)                .. [1] P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).
125: (4)                .. [2] E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).
126: (4)                """
127: (4)                if directed:
128: (8)                    edges = itertools.permutations(range(n), 2)
129: (8)                    G = nx.DiGraph()
130: (4)                else:
131: (8)                    edges = itertools.combinations(range(n), 2)
132: (8)                    G = nx.Graph()
133: (4)                G.add_nodes_from(range(n))
134: (4)                if p <= 0:
135: (8)                    return G
136: (4)                if p >= 1:
137: (8)                    return complete_graph(n, create_using=G)
138: (4)                for e in edges:
139: (8)                    if seed.random() < p:
140: (12)                       G.add_edge(*e)
141: (4)                return G
142: (0)            binomial_graph = gnp_random_graph
143: (0)            erdos_renyi_graph = gnp_random_graph
144: (0)            @py_random_state(2)
145: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
146: (0)            def dense_gnm_random_graph(n, m, seed=None):
147: (4)                """Returns a $G_{n,m}$ random graph.
148: (4)                In the $G_{n,m}$ model, a graph is chosen uniformly at random from the set
149: (4)                of all graphs with $n$ nodes and $m$ edges.
150: (4)                This algorithm should be faster than :func:`gnm_random_graph` for dense
151: (4)                graphs.
152: (4)                Parameters
153: (4)                ----------
154: (4)                n : int
155: (8)                    The number of nodes.
156: (4)                m : int
```

```
157: (8)                    The number of edges.
158: (4)                seed : integer, random_state, or None (default)
159: (8)                    Indicator of random number generation state.
160: (8)                    See :ref:`Randomness<randomness>`.
161: (4)                See Also
162: (4)                --------
163: (4)                gnm_random_graph
164: (4)                Notes
165: (4)                -----
166: (4)                Algorithm by Keith M. Briggs Mar 31, 2006.
167: (4)                Inspired by Knuth's Algorithm S (Selection sampling technique),
168: (4)                in section 3.4.2 of [1]_.
169: (4)                References
170: (4)                ----------
171: (4)                .. [1] Donald E. Knuth, The Art of Computer Programming,
172: (8)                    Volume 2/Seminumerical algorithms, Third Edition, Addison-Wesley,
1997.
173: (4)                """
174: (4)                mmax = n * (n - 1) // 2
175: (4)                if m >= mmax:
176: (8)                    G = complete_graph(n)
177: (4)                else:
178: (8)                    G = empty_graph(n)
179: (4)                if n == 1 or m >= mmax:
180: (8)                    return G
181: (4)                u = 0
182: (4)                v = 1
183: (4)                t = 0
184: (4)                k = 0
185: (4)                while True:
186: (8)                    if seed.randrange(mmax - t) < m - k:
187: (12)                       G.add_edge(u, v)
188: (12)                       k += 1
189: (12)                       if k == m:
190: (16)                           return G
191: (8)                    t += 1
192: (8)                    v += 1
193: (8)                    if v == n:  # go to next row of adjacency matrix
194: (12)                       u += 1
195: (12)                       v = u + 1
196: (0)            @py_random_state(2)
197: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
198: (0)            def gnm_random_graph(n, m, seed=None, directed=False):
199: (4)                """Returns a $G_{n,m}$ random graph.
200: (4)                In the $G_{n,m}$ model, a graph is chosen uniformly at random from the set
201: (4)                of all graphs with $n$ nodes and $m$ edges.
202: (4)                This algorithm should be faster than :func:`dense_gnm_random_graph` for
203: (4)                sparse graphs.
204: (4)                Parameters
205: (4)                ----------
206: (4)                n : int
207: (8)                    The number of nodes.
208: (4)                m : int
209: (8)                    The number of edges.
210: (4)                seed : integer, random_state, or None (default)
211: (8)                    Indicator of random number generation state.
212: (8)                    See :ref:`Randomness<randomness>`.
213: (4)                directed : bool, optional (default=False)
214: (8)                    If True return a directed graph
215: (4)                See also
216: (4)                --------
217: (4)                dense_gnm_random_graph
218: (4)                """
219: (4)                if directed:
220: (8)                    G = nx.DiGraph()
221: (4)                else:
222: (8)                    G = nx.Graph()
223: (4)                G.add_nodes_from(range(n))
224: (4)                if n == 1:
```

```
(8)                 return G
(4)             max_edges = n * (n - 1)
(4)             if not directed:
(8)                 max_edges /= 2.0
(4)             if m >= max_edges:
(8)                 return complete_graph(n, create_using=G)
(4)             nlist = list(G)
(4)             edge_count = 0
(4)             while edge_count < m:
(8)                 u = seed.choice(nlist)
(8)                 v = seed.choice(nlist)
(8)                 if u == v or G.has_edge(u, v):
(12)                    continue
(8)                 else:
(12)                    G.add_edge(u, v)
(12)                    edge_count = edge_count + 1
(4)             return G
(0)         @py_random_state(3)
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def newman_watts_strogatz_graph(n, k, p, seed=None):
(4)             """Returns a Newman–Watts–Strogatz small-world graph.
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 The number of nodes.
(4)             k : int
(8)                 Each node is joined with its `k` nearest neighbors in a ring
(8)                 topology.
(4)             p : float
(8)                 The probability of adding a new edge for each edge.
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Notes
(4)             -----
(4)             First create a ring over $n$ nodes [1]_.  Then each node in the ring is
(4)             connected with its $k$ nearest neighbors (or $k - 1$ neighbors if $k$
(4)             is odd).  Then shortcuts are created by adding new edges as follows: for
(4)             each edge $(u, v)$ in the underlying "$n$-ring with $k$ nearest
(4)             neighbors" with probability $p$ add a new edge $(u, w)$ with
(4)             randomly-chosen existing node $w$.  In contrast with
(4)             :func:`watts_strogatz_graph`, no edges are removed.
(4)             See Also
(4)             --------
(4)             watts_strogatz_graph
(4)             References
(4)             ----------
(4)             .. [1] M. E. J. Newman and D. J. Watts,
(7)                Renormalization group analysis of the small-world network model,
(7)                Physics Letters A, 263, 341, 1999.
(7)                https://doi.org/10.1016/S0375-9601(99)00757-4
(4)             """
(4)             if k > n:
(8)                 raise nx.NetworkXError("k>=n, choose smaller k or larger n")
(4)             if k == n:
(8)                 return nx.complete_graph(n)
(4)             G = empty_graph(n)
(4)             nlist = list(G.nodes())
(4)             fromv = nlist
(4)             for j in range(1, k // 2 + 1):
(8)                 tov = fromv[j:] + fromv[0:j]  # the first j are now last
(8)                 for i in range(len(fromv)):
(12)                    G.add_edge(fromv[i], tov[i])
(4)             e = list(G.edges())
(4)             for u, v in e:
(8)                 if seed.random() < p:
(12)                    w = seed.choice(nlist)
(12)                    while w == u or G.has_edge(u, w):
(16)                        w = seed.choice(nlist)
```

```
(16)                            if G.degree(u) >= n - 1:
(20)                                break  # skip this rewiring
(12)                        else:
(16)                            G.add_edge(u, w)
(4)                 return G
(0)             @py_random_state(3)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def watts_strogatz_graph(n, k, p, seed=None):
(4)                 """Returns a Watts–Strogatz small-world graph.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes
(4)                 k : int
(8)                     Each node is joined with its `k` nearest neighbors in a ring
(8)                     topology.
(4)                 p : float
(8)                     The probability of rewiring each edge
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 See Also
(4)                 --------
(4)                 newman_watts_strogatz_graph
(4)                 connected_watts_strogatz_graph
(4)                 Notes
(4)                 -----
(4)                 First create a ring over $n$ nodes [1]_.  Then each node in the ring is
joined
(4)                 to its $k$ nearest neighbors (or $k - 1$ neighbors if $k$ is odd).
(4)                 Then shortcuts are created by replacing some edges as follows: for each
(4)                 edge $(u, v)$ in the underlying "$n$-ring with $k$ nearest neighbors"
(4)                 with probability $p$ replace it with a new edge $(u, w)$ with uniformly
(4)                 random choice of existing node $w$.
(4)                 In contrast with :func:`newman_watts_strogatz_graph`, the random rewiring
(4)                 does not increase the number of edges. The rewired graph is not guaranteed
(4)                 to be connected as in :func:`connected_watts_strogatz_graph`.
(4)                 References
(4)                 ----------
(4)                 .. [1] Duncan J. Watts and Steven H. Strogatz,
(7)                    Collective dynamics of small-world networks,
(7)                    Nature, 393, pp. 440--442, 1998.
(4)                 """
(4)                 if k > n:
(8)                     raise nx.NetworkXError("k>n, choose smaller k or larger n")
(4)                 if k == n:
(8)                     return nx.complete_graph(n)
(4)                 G = nx.Graph()
(4)                 nodes = list(range(n))  # nodes are labeled 0 to n-1
(4)                 for j in range(1, k // 2 + 1):
(8)                     targets = nodes[j:] + nodes[0:j]  # first j nodes are now last in list
(8)                     G.add_edges_from(zip(nodes, targets))
(4)                 for j in range(1, k // 2 + 1):  # outer loop is neighbors
(8)                     targets = nodes[j:] + nodes[0:j]  # first j nodes are now last in list
(8)                     for u, v in zip(nodes, targets):
(12)                        if seed.random() < p:
(16)                            w = seed.choice(nodes)
(16)                            while w == u or G.has_edge(u, w):
(20)                                w = seed.choice(nodes)
(20)                                if G.degree(u) >= n - 1:
(24)                                    break  # skip this rewiring
(16)                            else:
(20)                                G.remove_edge(u, v)
(20)                                G.add_edge(u, w)
(4)                 return G
(0)             @py_random_state(4)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def connected_watts_strogatz_graph(n, k, p, tries=100, seed=None):
(4)                 """Returns a connected Watts–Strogatz small-world graph.
```

```
    Attempts to generate a connected graph by repeated generation of
    Watts–Strogatz small-world graphs.  An exception is raised if the maximum
    number of tries is exceeded.
    Parameters
    ----------
    n : int
        The number of nodes
    k : int
        Each node is joined with its `k` nearest neighbors in a ring
        topology.
    p : float
        The probability of rewiring each edge
    tries : int
        Number of attempts to generate a connected graph.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    Notes
    -----
    First create a ring over $n$ nodes [1]_.  Then each node in the ring is joined
    to its $k$ nearest neighbors (or $k - 1$ neighbors if $k$ is odd).
    Then shortcuts are created by replacing some edges as follows: for each
    edge $(u, v)$ in the underlying "$n$-ring with $k$ nearest neighbors"
    with probability $p$ replace it with a new edge $(u, w)$ with uniformly
    random choice of existing node $w$.
    The entire process is repeated until a connected graph results.
    See Also
    --------
    newman_watts_strogatz_graph
    watts_strogatz_graph
    References
    ----------
    .. [1] Duncan J. Watts and Steven H. Strogatz,
       Collective dynamics of small-world networks,
       Nature, 393, pp. 440--442, 1998.
    """
    for i in range(tries):
        G = watts_strogatz_graph(n, k, p, seed)
        if nx.is_connected(G):
            return G
    raise nx.NetworkXError("Maximum number of tries exceeded")
@py_random_state(2)
@nx._dispatchable(graphs=None, returns_graph=True)
def random_regular_graph(d, n, seed=None):
    r"""Returns a random $d$-regular graph on $n$ nodes.
    A regular graph is a graph where each node has the same number of neighbors.
    The resulting graph has no self-loops or parallel edges.
    Parameters
    ----------
    d : int
      The degree of each node.
    n : integer
      The number of nodes. The value of $n \times d$ must be even.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    Notes
    -----
    The nodes are numbered from $0$ to $n - 1$.
    Kim and Vu's paper [2]_ shows that this algorithm samples in an
    asymptotically uniform way from the space of random graphs when
    $d = O(n^{1 / 3 - \epsilon})$.
    Raises
    ------
    NetworkXError
        If $n \times d$ is odd or $d$ is greater than or equal to $n$.
    References
```

```
(4)                 ----------
(4)                 .. [1] A. Steger and N. Wormald,
(7)                    Generating random regular graphs quickly,
(7)                    Probability and Computing 8 (1999), 377-396, 1999.
(7)                    https://doi.org/10.1017/S0963548399003867
(4)                 .. [2] Jeong Han Kim and Van H. Vu,
(7)                    Generating random regular graphs,
(7)                    Proceedings of the thirty-fifth ACM symposium on Theory of computing,
(7)                    San Diego, CA, USA, pp 213--222, 2003.
(7)                    http://portal.acm.org/citation.cfm?id=780542.780576
(4)                 """
(4)                 if (n * d) % 2 != 0:
(8)                     raise nx.NetworkXError("n * d must be even")
(4)                 if not 0 <= d < n:
(8)                     raise nx.NetworkXError("the 0 <= d < n inequality must be satisfied")
(4)                 if d == 0:
(8)                     return empty_graph(n)
(4)                 def _suitable(edges, potential_edges):
(8)                     if not potential_edges:
(12)                        return True
(8)                     for s1 in potential_edges:
(12)                        for s2 in potential_edges:
(16)                            if s1 == s2:
(20)                                break
(16)                            if s1 > s2:
(20)                                s1, s2 = s2, s1
(16)                            if (s1, s2) not in edges:
(20)                                return True
(8)                     return False
(4)                 def _try_creation():
(8)                     edges = set()
(8)                     stubs = list(range(n)) * d
(8)                     while stubs:
(12)                        potential_edges = defaultdict(lambda: 0)
(12)                        seed.shuffle(stubs)
(12)                        stubiter = iter(stubs)
(12)                        for s1, s2 in zip(stubiter, stubiter):
(16)                            if s1 > s2:
(20)                                s1, s2 = s2, s1
(16)                            if s1 != s2 and ((s1, s2) not in edges):
(20)                                edges.add((s1, s2))
(16)                            else:
(20)                                potential_edges[s1] += 1
(20)                                potential_edges[s2] += 1
(12)                        if not _suitable(edges, potential_edges):
(16)                            return None  # failed to find suitable edge set
(12)                        stubs = [
(16)                            node
(16)                            for node, potential in potential_edges.items()
(16)                            for _ in range(potential)
(12)                        ]
(8)                     return edges
(4)                 edges = _try_creation()
(4)                 while edges is None:
(8)                     edges = _try_creation()
(4)                 G = nx.Graph()
(4)                 G.add_edges_from(edges)
(4)                 return G
(0)             def _random_subset(seq, m, rng):
(4)                 """Return m unique elements from seq.
(4)                 This differs from random.sample which can return repeated
(4)                 elements if seq holds repeated elements.
(4)                 Note: rng is a random.Random or numpy.random.RandomState instance.
(4)                 """
(4)                 targets = set()
(4)                 while len(targets) < m:
(8)                     x = rng.choice(seq)
(8)                     targets.add(x)
(4)                 return targets
```

```
@py_random_state(2)
@nx._dispatchable(graphs=None, returns_graph=True)
def barabasi_albert_graph(n, m, seed=None, initial_graph=None):
    """Returns a random graph using Barabási–Albert preferential attachment
    A graph of $n$ nodes is grown by attaching new nodes each with $m$
    edges that are preferentially attached to existing nodes with high degree.
    Parameters
    ----------
    n : int
        Number of nodes
    m : int
        Number of edges to attach from a new node to existing nodes
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
    initial_graph : Graph or None (default)
        Initial network for Barabási–Albert algorithm.
        It should be a connected graph for most use cases.
        A copy of `initial_graph` is used.
        If None, starts from a star graph on (m+1) nodes.
    Returns
    -------
    G : Graph
    Raises
    ------
    NetworkXError
        If `m` does not satisfy ``1 <= m < n``, or
        the initial graph number of nodes m0 does not satisfy ``m <= m0 <= n``.
    References
    ----------
    .. [1] A. L. Barabási and R. Albert "Emergence of scaling in
       random networks", Science 286, pp 509-512, 1999.
    """
    if m < 1 or m >= n:
        raise nx.NetworkXError(
            f"Barabási–Albert network must have m >= 1 and m < n, m = {m}, n = {n}"
        )
    if initial_graph is None:
        G = star_graph(m)
    else:
        if len(initial_graph) < m or len(initial_graph) > n:
            raise nx.NetworkXError(
                f"Barabási–Albert initial graph needs between m={m} and n={n} nodes"
            )
        G = initial_graph.copy()
    repeated_nodes = [n for n, d in G.degree() for _ in range(d)]
    source = len(G)
    while source < n:
        targets = _random_subset(repeated_nodes, m, seed)
        G.add_edges_from(zip([source] * m, targets))
        repeated_nodes.extend(targets)
        repeated_nodes.extend([source] * m)
        source += 1
    return G
@py_random_state(4)
@nx._dispatchable(graphs=None, returns_graph=True)
def dual_barabasi_albert_graph(n, m1, m2, p, seed=None, initial_graph=None):
    """Returns a random graph using dual Barabási–Albert preferential attachment
    A graph of $n$ nodes is grown by attaching new nodes each with either $m_1$
    edges (with probability $p$) or $m_2$ edges (with probability $1-p$) that
    are preferentially attached to existing nodes with high degree.
    Parameters
    ----------
    n : int
```

```
562: (8)                    Number of nodes
563: (4)                m1 : int
564: (8)                    Number of edges to link each new node to existing nodes with
probability $p$
565: (4)                m2 : int
566: (8)                    Number of edges to link each new node to existing nodes with
probability $1-p$
567: (4)                p : float
568: (8)                    The probability of attaching $m_1$ edges (as opposed to $m_2$ edges)
569: (4)                seed : integer, random_state, or None (default)
570: (8)                    Indicator of random number generation state.
571: (8)                    See :ref:`Randomness<randomness>`.
572: (4)                initial_graph : Graph or None (default)
573: (8)                    Initial network for Barabási–Albert algorithm.
574: (8)                    A copy of `initial_graph` is used.
575: (8)                    It should be connected for most use cases.
576: (8)                    If None, starts from an star graph on max(m1, m2) + 1 nodes.
577: (4)                Returns
578: (4)                -------
579: (4)                G : Graph
580: (4)                Raises
581: (4)                ------
582: (4)                NetworkXError
583: (8)                    If `m1` and `m2` do not satisfy ``1 <= m1,m2 < n``, or
584: (8)                    `p` does not satisfy ``0 <= p <= 1``, or
585: (8)                    the initial graph number of nodes m0 does not satisfy m1, m2 <= m0 <=
n.
586: (4)                References
587: (4)                ----------
588: (4)                .. [1] N. Moshiri "The dual-Barabasi-Albert model", arXiv:1810.10538.
589: (4)                """
590: (4)                if m1 < 1 or m1 >= n:
591: (8)                    raise nx.NetworkXError(
592: (12)                       f"Dual Barabási–Albert must have m1 >= 1 and m1 < n, m1 = {m1}, n
= {n}"
593: (8)                    )
594: (4)                if m2 < 1 or m2 >= n:
595: (8)                    raise nx.NetworkXError(
596: (12)                       f"Dual Barabási–Albert must have m2 >= 1 and m2 < n, m2 = {m2}, n
= {n}"
597: (8)                    )
598: (4)                if p < 0 or p > 1:
599: (8)                    raise nx.NetworkXError(
600: (12)                       f"Dual Barabási–Albert network must have 0 <= p <= 1, p = {p}"
601: (8)                    )
602: (4)                if p == 1:
603: (8)                    return barabasi_albert_graph(n, m1, seed)
604: (4)                elif p == 0:
605: (8)                    return barabasi_albert_graph(n, m2, seed)
606: (4)                if initial_graph is None:
607: (8)                    G = star_graph(max(m1, m2))
608: (4)                else:
609: (8)                    if len(initial_graph) < max(m1, m2) or len(initial_graph) > n:
610: (12)                       raise nx.NetworkXError(
611: (16)                           f"Barabási–Albert initial graph must have between "
612: (16)                           f"max(m1, m2) = {max(m1, m2)} and n = {n} nodes"
613: (12)                       )
614: (8)                    G = initial_graph.copy()
615: (4)                targets = list(G)
616: (4)                repeated_nodes = [n for n, d in G.degree() for _ in range(d)]
617: (4)                source = len(G)
618: (4)                while source < n:
619: (8)                    if seed.random() < p:
620: (12)                       m = m1
621: (8)                    else:
622: (12)                       m = m2
623: (8)                    targets = _random_subset(repeated_nodes, m, seed)
624: (8)                    G.add_edges_from(zip([source] * m, targets))
625: (8)                    repeated_nodes.extend(targets)
```

```
(8)                 repeated_nodes.extend([source] * m)
(8)                 source += 1
(4)             return G
(0)         @py_random_state(4)
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def extended_barabasi_albert_graph(n, m, p, q, seed=None):
(4)             """Returns an extended Barabási–Albert model graph.
(4)             An extended Barabási–Albert model graph is a random graph constructed
(4)             using preferential attachment. The extended model allows new edges,
(4)             rewired edges or new nodes. Based on the probabilities $p$ and $q$
(4)             with $p + q < 1$, the growing behavior of the graph is determined as:
(4)             1) With $p$ probability, $m$ new edges are added to the graph,
(4)             starting from randomly chosen existing nodes and attached preferentially
at the other end.
(4)             2) With $q$ probability, $m$ existing edges are rewired
(4)             by randomly choosing an edge and rewiring one end to a preferentially
chosen node.
(4)             3) With $(1 - p - q)$ probability, $m$ new nodes are added to the graph
(4)             with edges attached preferentially.
(4)             When $p = q = 0$, the model behaves just like the Barabási–Alber model.
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 Number of nodes
(4)             m : int
(8)                 Number of edges with which a new node attaches to existing nodes
(4)             p : float
(8)                 Probability value for adding an edge between existing nodes. p + q < 1
(4)             q : float
(8)                 Probability value of rewiring of existing edges. p + q < 1
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             G : Graph
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If `m` does not satisfy ``1 <= m < n`` or ``1 >= p + q``
(4)             References
(4)             ----------
(4)             .. [1] Albert, R., & Barabási, A. L. (2000)
(7)                Topology of evolving networks: local events and universality
(7)                Physical review letters, 85(24), 5234.
(4)             """
(4)             if m < 1 or m >= n:
(8)                 msg = f"Extended Barabasi-Albert network needs m>=1 and m<n, m={m}, n=
{n}"
(8)                 raise nx.NetworkXError(msg)
(4)             if p + q >= 1:
(8)                 msg = f"Extended Barabasi-Albert network needs p + q <= 1, p={p}, q=
{q}"
(8)                 raise nx.NetworkXError(msg)
(4)             G = empty_graph(m)
(4)             attachment_preference = []
(4)             attachment_preference.extend(range(m))
(4)             new_node = m
(4)             while new_node < n:
(8)                 a_probability = seed.random()
(8)                 clique_degree = len(G) - 1
(8)                 clique_size = (len(G) * clique_degree) / 2
(8)                 if a_probability < p and G.size() <= clique_size - m:
(12)                    eligible_nodes = [nd for nd, deg in G.degree() if deg <
clique_degree]
(12)                    for i in range(m):
(16)                        src_node = seed.choice(eligible_nodes)
(16)                        prohibited_nodes = list(G[src_node])
(16)                        prohibited_nodes.append(src_node)
```

```
690: (16)                            dest_node = seed.choice(
691: (20)                                [nd for nd in attachment_preference if nd not in
prohibited_nodes]
692: (16)                            )
693: (16)                            G.add_edge(src_node, dest_node)
694: (16)                            attachment_preference.append(src_node)
695: (16)                            attachment_preference.append(dest_node)
696: (16)                            if G.degree(src_node) == clique_degree:
697: (20)                                eligible_nodes.remove(src_node)
698: (16)                            if G.degree(dest_node) == clique_degree and dest_node in
eligible_nodes:
699: (20)                                eligible_nodes.remove(dest_node)
700: (8)                    elif p <= a_probability < (p + q) and m <= G.size() < clique_size:
701: (12)                       eligible_nodes = [nd for nd, deg in G.degree() if 0 < deg <
clique_degree]
702: (12)                       for i in range(m):
703: (16)                           node = seed.choice(eligible_nodes)
704: (16)                           nbr_nodes = list(G[node])
705: (16)                           src_node = seed.choice(nbr_nodes)
706: (16)                           nbr_nodes.append(node)
707: (16)                           dest_node = seed.choice(
708: (20)                               [nd for nd in attachment_preference if nd not in
nbr_nodes]
709: (16)                           )
710: (16)                           G.remove_edge(node, src_node)
711: (16)                           G.add_edge(node, dest_node)
712: (16)                           attachment_preference.remove(src_node)
713: (16)                           attachment_preference.append(dest_node)
714: (16)                           if G.degree(src_node) == 0 and src_node in eligible_nodes:
715: (20)                               eligible_nodes.remove(src_node)
716: (16)                           if dest_node in eligible_nodes:
717: (20)                               if G.degree(dest_node) == clique_degree:
718: (24)                                   eligible_nodes.remove(dest_node)
719: (16)                           else:
720: (20)                               if G.degree(dest_node) == 1:
721: (24)                                   eligible_nodes.append(dest_node)
722: (8)                    else:
723: (12)                       targets = _random_subset(attachment_preference, m, seed)
724: (12)                       G.add_edges_from(zip([new_node] * m, targets))
725: (12)                       attachment_preference.extend(targets)
726: (12)                       attachment_preference.extend([new_node] * (m + 1))
727: (12)                       new_node += 1
728: (4)                return G
729: (0)            @py_random_state(3)
730: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
731: (0)            def powerlaw_cluster_graph(n, m, p, seed=None):
732: (4)                """Holme and Kim algorithm for growing graphs with powerlaw
733: (4)                degree distribution and approximate average clustering.
734: (4)                Parameters
735: (4)                ----------
736: (4)                n : int
737: (8)                    the number of nodes
738: (4)                m : int
739: (8)                    the number of random edges to add for each new node
740: (4)                p : float,
741: (8)                    Probability of adding a triangle after adding a random edge
742: (4)                seed : integer, random_state, or None (default)
743: (8)                    Indicator of random number generation state.
744: (8)                    See :ref:`Randomness<randomness>`.
745: (4)                Notes
746: (4)                -----
747: (4)                The average clustering has a hard time getting above a certain
748: (4)                cutoff that depends on `m`.  This cutoff is often quite low.  The
749: (4)                transitivity (fraction of triangles to possible triangles) seems to
750: (4)                decrease with network size.
751: (4)                It is essentially the Barabási–Albert (BA) growth model with an
752: (4)                extra step that each random edge is followed by a chance of
753: (4)                making an edge to one of its neighbors too (and thus a triangle).
754: (4)                This algorithm improves on BA in the sense that it enables a
```

```
    higher average clustering to be attained if desired.
    It seems possible to have a disconnected graph with this algorithm
    since the initial `m` nodes may not be all linked to a new node
    on the first iteration like the BA model.
    Raises
    ------
    NetworkXError
        If `m` does not satisfy ``1 <= m <= n`` or `p` does not
        satisfy ``0 <= p <= 1``.
    References
    ----------
    .. [1] P. Holme and B. J. Kim,
       "Growing scale-free networks with tunable clustering",
       Phys. Rev. E, 65, 026107, 2002.
    """
    if m < 1 or n < m:
        raise nx.NetworkXError(f"NetworkXError must have m>1 and m<n, m={m},n={n}")
    if p > 1 or p < 0:
        raise nx.NetworkXError(f"NetworkXError p must be in [0,1], p={p}")
    G = empty_graph(m)  # add m initial nodes (m0 in barabasi-speak)
    repeated_nodes = list(G.nodes())  # list of existing nodes to sample from
    source = m  # next node is m
    while source < n:  # Now add the other n-1 nodes
        possible_targets = _random_subset(repeated_nodes, m, seed)
        target = possible_targets.pop()
        G.add_edge(source, target)
        repeated_nodes.append(target)  # add one node to list for each new link
        count = 1
        while count < m:  # add m-1 more new links
            if seed.random() < p:  # clustering step: add triangle
                neighborhood = [
                    nbr
                    for nbr in G.neighbors(target)
                    if not G.has_edge(source, nbr) and nbr != source
                ]
                if neighborhood:  # if there is a neighbor without a link
                    nbr = seed.choice(neighborhood)
                    G.add_edge(source, nbr)  # add triangle
                    repeated_nodes.append(nbr)
                    count = count + 1
                    continue  # go to top of while loop
            target = possible_targets.pop()
            G.add_edge(source, target)
            repeated_nodes.append(target)
            count = count + 1
        repeated_nodes.extend([source] * m)  # add source node to list m times
        source += 1
    return G
@py_random_state(3)
@nx._dispatchable(graphs=None, returns_graph=True)
def random_lobster(n, p1, p2, seed=None):
    """Returns a random lobster graph.
    A lobster is a tree that reduces to a caterpillar when pruning all
    leaf nodes. A caterpillar is a tree that reduces to a path graph
    when pruning all leaf nodes; setting `p2` to zero produces a caterpillar.
    This implementation iterates on the probabilities `p1` and `p2` to add
    edges at levels 1 and 2, respectively. Graphs are therefore constructed
    iteratively with uniform randomness at each level rather than being selected
    uniformly at random from the set of all possible lobsters.
    Parameters
    ----------
    n : int
        The expected number of nodes in the backbone
    p1 : float
        Probability of adding an edge to the backbone
    p2 : float
```

```
821: (8)                    Probability of adding an edge one level beyond backbone
822: (4)                seed : integer, random_state, or None (default)
823: (8)                    Indicator of random number generation state.
824: (8)                    See :ref:`Randomness<randomness>`.
825: (4)                Raises
826: (4)                ------
827: (4)                NetworkXError
828: (8)                    If `p1` or `p2` parameters are >= 1 because the while loops would
never finish.
829: (4)                """
830: (4)                p1, p2 = abs(p1), abs(p2)
831: (4)                if any(p >= 1 for p in [p1, p2]):
832: (8)                    raise nx.NetworkXError("Probability values for `p1` and `p2` must both
be < 1.")
833: (4)                llen = int(2 * seed.random() * n + 0.5)
834: (4)                L = path_graph(llen)
835: (4)                current_node = llen - 1
836: (4)                for n in range(llen):
837: (8)                    while seed.random() < p1:  # add fuzzy caterpillar parts
838: (12)                       current_node += 1
839: (12)                       L.add_edge(n, current_node)
840: (12)                       cat_node = current_node
841: (12)                       while seed.random() < p2:  # add crunchy lobster bits
842: (16)                           current_node += 1
843: (16)                           L.add_edge(cat_node, current_node)
844: (4)                return L  # voila, un lobster!
845: (0)            @py_random_state(1)
846: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
847: (0)            def random_shell_graph(constructor, seed=None):
848: (4)                """Returns a random shell graph for the constructor given.
849: (4)                Parameters
850: (4)                ----------
851: (4)                constructor : list of three-tuples
852: (8)                    Represents the parameters for a shell, starting at the center
853: (8)                    shell.  Each element of the list must be of the form `(n, m,
854: (8)                    d)`, where `n` is the number of nodes in the shell, `m` is
855: (8)                    the number of edges in the shell, and `d` is the ratio of
856: (8)                    inter-shell (next) edges to intra-shell edges. If `d` is zero,
857: (8)                    there will be no intra-shell edges, and if `d` is one there
858: (8)                    will be all possible intra-shell edges.
859: (4)                seed : integer, random_state, or None (default)
860: (8)                    Indicator of random number generation state.
861: (8)                    See :ref:`Randomness<randomness>`.
862: (4)                Examples
863: (4)                --------
864: (4)                >>> constructor = [(10, 20, 0.8), (20, 40, 0.8)]
865: (4)                >>> G = nx.random_shell_graph(constructor)
866: (4)                """
867: (4)                G = empty_graph(0)
868: (4)                glist = []
869: (4)                intra_edges = []
870: (4)                nnodes = 0
871: (4)                for n, m, d in constructor:
872: (8)                    inter_edges = int(m * d)
873: (8)                    intra_edges.append(m - inter_edges)
874: (8)                    g = nx.convert_node_labels_to_integers(
875: (12)                       gnm_random_graph(n, inter_edges, seed=seed), first_label=nnodes
876: (8)                    )
877: (8)                    glist.append(g)
878: (8)                    nnodes += n
879: (8)                    G = nx.operators.union(G, g)
880: (4)                for gi in range(len(glist) - 1):
881: (8)                    nlist1 = list(glist[gi])
882: (8)                    nlist2 = list(glist[gi + 1])
883: (8)                    total_edges = intra_edges[gi]
884: (8)                    edge_count = 0
885: (8)                    while edge_count < total_edges:
886: (12)                       u = seed.choice(nlist1)
887: (12)                       v = seed.choice(nlist2)
```

```
(12)                        if u == v or G.has_edge(u, v):
(16)                            continue
(12)                        else:
(16)                            G.add_edge(u, v)
(16)                            edge_count = edge_count + 1
(4)                 return G
(0)             @py_random_state(2)
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_powerlaw_tree(n, gamma=3, seed=None, tries=100):
(4)                 """Returns a tree with a power law degree distribution.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes.
(4)                 gamma : float
(8)                     Exponent of the power law.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 tries : int
(8)                     Number of attempts to adjust the sequence to make it a tree.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If no valid sequence is found within the maximum number of
(8)                     attempts.
(4)                 Notes
(4)                 -----
(4)                 A trial power law degree sequence is chosen and then elements are
(4)                 swapped with new elements from a powerlaw distribution until the
(4)                 sequence makes a tree (by checking, for example, that the number of
(4)                 edges is one smaller than the number of nodes).
(4)                 """
(4)                 seq = random_powerlaw_tree_sequence(n, gamma=gamma, seed=seed,
tries=tries)
(4)                 G = degree_sequence_tree(seq)
(4)                 return G
(0)             @py_random_state(2)
(0)             @nx._dispatchable(graphs=None)
(0)             def random_powerlaw_tree_sequence(n, gamma=3, seed=None, tries=100):
(4)                 """Returns a degree sequence for a tree with a power law distribution.
(4)                 Parameters
(4)                 ----------
(4)                 n : int,
(8)                     The number of nodes.
(4)                 gamma : float
(8)                     Exponent of the power law.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 tries : int
(8)                     Number of attempts to adjust the sequence to make it a tree.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If no valid sequence is found within the maximum number of
(8)                     attempts.
(4)                 Notes
(4)                 -----
(4)                 A trial power law degree sequence is chosen and then elements are
(4)                 swapped with new elements from a power law distribution until
(4)                 the sequence makes a tree (by checking, for example, that the number of
(4)                 edges is one smaller than the number of nodes).
(4)                 """
(4)                 z = nx.utils.powerlaw_sequence(n, exponent=gamma, seed=seed)
(4)                 zseq = [min(n, max(round(s), 0)) for s in z]
(4)                 z = nx.utils.powerlaw_sequence(tries, exponent=gamma, seed=seed)
(4)                 swap = [min(n, max(round(s), 0)) for s in z]
(4)                 for deg in swap:
```

```
956: (8)                    if 2 * n - sum(zseq) == 2:
957: (12)                       return zseq
958: (8)                    index = seed.randint(0, n - 1)
959: (8)                    zseq[index] = swap.pop()
960: (4)                raise nx.NetworkXError(
961: (8)                    f"Exceeded max ({tries}) attempts for a valid tree sequence."
962: (4)                )
963: (0)            @py_random_state(3)
964: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
965: (0)            def random_kernel_graph(n, kernel_integral, kernel_root=None, seed=None):
966: (4)                r"""Returns an random graph based on the specified kernel.
967: (4)                The algorithm chooses each of the $[n(n-1)]/2$ possible edges with
968: (4)                probability specified by a kernel $\kappa(x,y)$ [1]_.  The kernel
969: (4)                $\kappa(x,y)$ must be a symmetric (in $x,y$), non-negative,
970: (4)                bounded function.
971: (4)                Parameters
972: (4)                ----------
973: (4)                n : int
974: (8)                    The number of nodes
975: (4)                kernel_integral : function
976: (8)                    Function that returns the definite integral of the kernel
$\kappa(x,y)$,
977: (8)                    $F(y,a,b) := \int_a^b \kappa(x,y)dx$
978: (4)                kernel_root: function (optional)
979: (8)                    Function that returns the root $b$ of the equation $F(y,a,b) = r$.
980: (8)                    If None, the root is found using :func:`scipy.optimize.brentq`
981: (8)                    (this requires SciPy).
982: (4)                seed : integer, random_state, or None (default)
983: (8)                    Indicator of random number generation state.
984: (8)                    See :ref:`Randomness<randomness>`.
985: (4)                Notes
986: (4)                -----
987: (4)                The kernel is specified through its definite integral which must be
988: (4)                provided as one of the arguments. If the integral and root of the
989: (4)                kernel integral can be found in $O(1)$ time then this algorithm runs in
990: (4)                time $O(n+m)$ where m is the expected number of edges [2]_.
991: (4)                The nodes are set to integers from $0$ to $n-1$.
992: (4)                Examples
993: (4)                --------
994: (4)                Generate an Erdős–Rényi random graph $G(n,c/n)$, with kernel
995: (4)                $\kappa(x,y)=c$ where $c$ is the mean expected degree.
996: (4)                >>> def integral(u, w, z):
997: (4)                ...     return c * (z - w)
998: (4)                >>> def root(u, w, r):
999: (4)                ...     return r / c + w
1000: (4)               >>> c = 1
1001: (4)               >>> graph = nx.random_kernel_graph(1000, integral, root)
1002: (4)               See Also
1003: (4)               --------
1004: (4)               gnp_random_graph
1005: (4)               expected_degree_graph
1006: (4)               References
1007: (4)               ----------
1008: (4)               .. [1] Bollobás, Béla,  Janson, S. and Riordan, O.
1009: (7)                  "The phase transition in inhomogeneous random graphs",
1010: (7)                  *Random Structures Algorithms*, 31, 3--122, 2007.
1011: (4)               .. [2] Hagberg A, Lemons N (2015),
1012: (7)                  "Fast Generation of Sparse Random Kernel Graphs".
1013: (7)                  PLoS ONE 10(9): e0135177, 2015. doi:10.1371/journal.pone.0135177
1014: (4)               """
1015: (4)               if kernel_root is None:
1016: (8)                   import scipy as sp
1017: (8)                   def kernel_root(y, a, r):
1018: (12)                      def my_function(b):
1019: (16)                          return kernel_integral(y, a, b) - r
1020: (12)                      return sp.optimize.brentq(my_function, a, 1)
1021: (4)               graph = nx.Graph()
1022: (4)               graph.add_nodes_from(range(n))
1023: (4)               (i, j) = (1, 1)
```

```
1024: (4)                while i < n:
1025: (8)                    r = -math.log(1 - seed.random())  # (1-seed.random()) in (0, 1]
1026: (8)                    if kernel_integral(i / n, j / n, 1) <= r:
1027: (12)                       i, j = i + 1, i + 1
1028: (8)                    else:
1029: (12)                       j = math.ceil(n * kernel_root(i / n, j / n, r))
1030: (12)                       graph.add_edge(i - 1, j - 1)
1031: (4)                return graph

----------------------------------------

File 447 - random_clustered.py:

1: (0)               """Generate graphs with given degree and triangle sequence.
2: (0)               """
3: (0)               import networkx as nx
4: (0)               from networkx.utils import py_random_state
5: (0)               __all__ = ["random_clustered_graph"]
6: (0)               @py_random_state(2)
7: (0)               @nx._dispatchable(graphs=None, returns_graph=True)
8: (0)               def random_clustered_graph(joint_degree_sequence, create_using=None,
seed=None):
9: (4)                   r"""Generate a random graph with the given joint independent edge degree
and
10: (4)                  triangle degree sequence.
11: (4)                  This uses a configuration model-like approach to generate a random graph
12: (4)                  (with parallel edges and self-loops) by randomly assigning edges to match
13: (4)                  the given joint degree sequence.
14: (4)                  The joint degree sequence is a list of pairs of integers of the form
15: (4)                  $[(d_{1,i}, d_{1,t}), \dotsc, (d_{n,i}, d_{n,t})]$. According to this
list,
16: (4)                  vertex $u$ is a member of $d_{u,t}$ triangles and has $d_{u, i}$ other
17: (4)                  edges. The number $d_{u,t}$ is the *triangle degree* of $u$ and the number
18: (4)                  $d_{u,i}$ is the *independent edge degree*.
19: (4)                  Parameters
20: (4)                  ----------
21: (4)                  joint_degree_sequence : list of integer pairs
22: (8)                      Each list entry corresponds to the independent edge degree and
23: (8)                      triangle degree of a node.
24: (4)                  create_using : NetworkX graph constructor, optional (default MultiGraph)
25: (7)                     Graph type to create. If graph instance, then cleared before populated.
26: (4)                  seed : integer, random_state, or None (default)
27: (8)                      Indicator of random number generation state.
28: (8)                      See :ref:`Randomness<randomness>`.
29: (4)                  Returns
30: (4)                  -------
31: (4)                  G : MultiGraph
32: (8)                      A graph with the specified degree sequence. Nodes are labeled
33: (8)                      starting at 0 with an index corresponding to the position in
34: (8)                      deg_sequence.
35: (4)                  Raises
36: (4)                  ------
37: (4)                  NetworkXError
38: (8)                      If the independent edge degree sequence sum is not even
39: (8)                      or the triangle degree sequence sum is not divisible by 3.
40: (4)                  Notes
41: (4)                  -----
42: (4)                  As described by Miller [1]_ (see also Newman [2]_ for an equivalent
43: (4)                  description).
44: (4)                  A non-graphical degree sequence (not realizable by some simple
45: (4)                  graph) is allowed since this function returns graphs with self
46: (4)                  loops and parallel edges.  An exception is raised if the
47: (4)                  independent degree sequence does not have an even sum or the
48: (4)                  triangle degree sequence sum is not divisible by 3.
49: (4)                  This configuration model-like construction process can lead to
50: (4)                  duplicate edges and loops.  You can remove the self-loops and
51: (4)                  parallel edges (see below) which will likely result in a graph
52: (4)                  that doesn't have the exact degree sequence specified.  This
53: (4)                  "finite-size effect" decreases as the size of the graph increases.
```

```
54: (4)                 References
55: (4)                 ----------
56: (4)                 .. [1] Joel C. Miller. "Percolation and epidemics in random clustered
57: (11)                       networks". In: Physical review. E, Statistical, nonlinear, and soft
58: (11)                       matter physics 80 (2 Part 1 August 2009).
59: (4)                 .. [2] M. E. J. Newman. "Random Graphs with Clustering".
60: (11)                       In: Physical Review Letters 103 (5 July 2009)
61: (4)                 Examples
62: (4)                 --------
63: (4)                 >>> deg = [(1, 0), (1, 0), (1, 0), (2, 0), (1, 0), (2, 1), (0, 1), (0, 1)]
64: (4)                 >>> G = nx.random_clustered_graph(deg)
65: (4)                 To remove parallel edges:
66: (4)                 >>> G = nx.Graph(G)
67: (4)                 To remove self loops:
68: (4)                 >>> G.remove_edges_from(nx.selfloop_edges(G))
69: (4)                 """
70: (4)                 joint_degree_sequence = list(joint_degree_sequence)
71: (4)                 N = len(joint_degree_sequence)
72: (4)                 G = nx.empty_graph(N, create_using, default=nx.MultiGraph)
73: (4)                 if G.is_directed():
74: (8)                     raise nx.NetworkXError("Directed Graph not supported")
75: (4)                 ilist = []
76: (4)                 tlist = []
77: (4)                 for n in G:
78: (8)                     degrees = joint_degree_sequence[n]
79: (8)                     for icount in range(degrees[0]):
80: (12)                        ilist.append(n)
81: (8)                     for tcount in range(degrees[1]):
82: (12)                        tlist.append(n)
83: (4)                 if len(ilist) % 2 != 0 or len(tlist) % 3 != 0:
84: (8)                     raise nx.NetworkXError("Invalid degree sequence")
85: (4)                 seed.shuffle(ilist)
86: (4)                 seed.shuffle(tlist)
87: (4)                 while ilist:
88: (8)                     G.add_edge(ilist.pop(), ilist.pop())
89: (4)                 while tlist:
90: (8)                     n1 = tlist.pop()
91: (8)                     n2 = tlist.pop()
92: (8)                     n3 = tlist.pop()
93: (8)                     G.add_edges_from([(n1, n2), (n1, n3), (n2, n3)])
94: (4)                 return G

----------------------------------------

File 448 - small.py:

1: (0)              """
2: (0)              Various small and named graphs, together with some compact generators.
3: (0)              """
4: (0)              __all__ = [
5: (4)                  "LCF_graph",
6: (4)                  "bull_graph",
7: (4)                  "chvatal_graph",
8: (4)                  "cubical_graph",
9: (4)                  "desargues_graph",
10: (4)                 "diamond_graph",
11: (4)                 "dodecahedral_graph",
12: (4)                 "frucht_graph",
13: (4)                 "heawood_graph",
14: (4)                 "hoffman_singleton_graph",
15: (4)                 "house_graph",
16: (4)                 "house_x_graph",
17: (4)                 "icosahedral_graph",
18: (4)                 "krackhardt_kite_graph",
19: (4)                 "moebius_kantor_graph",
20: (4)                 "octahedral_graph",
21: (4)                 "pappus_graph",
22: (4)                 "petersen_graph",
23: (4)                 "sedgewick_maze_graph",
```

```
24: (4)                 "tetrahedral_graph",
25: (4)                 "truncated_cube_graph",
26: (4)                 "truncated_tetrahedron_graph",
27: (4)                 "tutte_graph",
28: (0)             ]
29: (0)             from functools import wraps
30: (0)             import networkx as nx
31: (0)             from networkx.exception import NetworkXError
32: (0)             from networkx.generators.classic import (
33: (4)                 complete_graph,
34: (4)                 cycle_graph,
35: (4)                 empty_graph,
36: (4)                 path_graph,
37: (0)             )
38: (0)             def _raise_on_directed(func):
39: (4)                 """
40: (4)                 A decorator which inspects the `create_using` argument and raises a
41: (4)                 NetworkX exception when `create_using` is a DiGraph (class or instance)
for
42: (4)                 graph generators that do not support directed outputs.
43: (4)                 """
44: (4)                 @wraps(func)
45: (4)                 def wrapper(*args, **kwargs):
46: (8)                     if kwargs.get("create_using") is not None:
47: (12)                        G = nx.empty_graph(create_using=kwargs["create_using"])
48: (12)                        if G.is_directed():
49: (16)                            raise NetworkXError("Directed Graph not supported")
50: (8)                     return func(*args, **kwargs)
51: (4)                 return wrapper
52: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
53: (0)             def LCF_graph(n, shift_list, repeats, create_using=None):
54: (4)                 """
55: (4)                 Return the cubic graph specified in LCF notation.
56: (4)                 LCF (Lederberg-Coxeter-Fruchte) notation[1]_ is a compressed
57: (4)                 notation used in the generation of various cubic Hamiltonian
58: (4)                 graphs of high symmetry. See, for example, `dodecahedral_graph`,
59: (4)                 `desargues_graph`, `heawood_graph` and `pappus_graph`.
60: (4)                 Nodes are drawn from ``range(n)``. Each node ``n_i`` is connected with
61: (4)                 node ``n_i + shift % n`` where ``shift`` is given by cycling through
62: (4)                 the input `shift_list` `repeat` s times.
63: (4)                 Parameters
64: (4)                 ----------
65: (4)                 n : int
66: (7)                    The starting graph is the `n`-cycle with nodes ``0, ..., n-1``.
67: (7)                    The null graph is returned if `n` < 1.
68: (4)                 shift_list : list
69: (7)                    A list of integer shifts mod `n`, ``[s1, s2, .., sk]``
70: (4)                 repeats : int
71: (7)                    Integer specifying the number of times that shifts in `shift_list`
72: (7)                    are successively applied to each current node in the n-cycle
73: (7)                    to generate an edge between ``n_current`` and ``n_current + shift mod
n``.
74: (4)                 Returns
75: (4)                 -------
76: (4)                 G : Graph
77: (7)                    A graph instance created from the specified LCF notation.
78: (4)                 Examples
79: (4)                 --------
80: (4)                 The utility graph $K_{3,3}$
81: (4)                 >>> G = nx.LCF_graph(6, [3, -3], 3)
82: (4)                 >>> G.edges()
83: (4)                 EdgeView([(0, 1), (0, 5), (0, 3), (1, 2), (1, 4), (2, 3), (2, 5), (3, 4),
(4, 5)])
84: (4)                 The Heawood graph:
85: (4)                 >>> G = nx.LCF_graph(14, [5, -5], 7)
86: (4)                 >>> nx.is_isomorphic(G, nx.heawood_graph())
87: (4)                 True
88: (4)                 References
89: (4)                 ----------
```

```
 90: (4)                .. [1] https://en.wikipedia.org/wiki/LCF_notation
 91: (4)                """
 92: (4)                if n <= 0:
 93: (8)                    return empty_graph(0, create_using)
 94: (4)                G = cycle_graph(n, create_using)
 95: (4)                if G.is_directed():
 96: (8)                    raise NetworkXError("Directed Graph not supported")
 97: (4)                G.name = "LCF_graph"
 98: (4)                nodes = sorted(G)
 99: (4)                n_extra_edges = repeats * len(shift_list)
 100: (4)               if n_extra_edges < 1:
 101: (8)                   return G
 102: (4)               for i in range(n_extra_edges):
 103: (8)                   shift = shift_list[i % len(shift_list)]  # cycle through shift_list
 104: (8)                   v1 = nodes[i % n]  # cycle repeatedly through nodes
 105: (8)                   v2 = nodes[(i + shift) % n]
 106: (8)                   G.add_edge(v1, v2)
 107: (4)               return G
 108: (0)           @_raise_on_directed
 109: (0)           @nx._dispatchable(graphs=None, returns_graph=True)
 110: (0)           def bull_graph(create_using=None):
 111: (4)               """
 112: (4)               Returns the Bull Graph
 113: (4)               The Bull Graph has 5 nodes and 5 edges. It is a planar undirected
 114: (4)               graph in the form of a triangle with two disjoint pendant edges [1]_
 115: (4)               The name comes from the triangle and pendant edges representing
 116: (4)               respectively the body and legs of a bull.
 117: (4)               Parameters
 118: (4)               ----------
 119: (4)               create_using : NetworkX graph constructor, optional (default=nx.Graph)
 120: (7)                  Graph type to create. If graph instance, then cleared before populated.
 121: (4)               Returns
 122: (4)               -------
 123: (4)               G : networkx Graph
 124: (8)                   A bull graph with 5 nodes
 125: (4)               References
 126: (4)               ----------
 127: (4)               .. [1] https://en.wikipedia.org/wiki/Bull_graph.
 128: (4)               """
 129: (4)               G = nx.from_dict_of_lists(
 130: (8)                   {0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 4], 3: [1], 4: [2]},
 131: (8)                   create_using=create_using,
 132: (4)               )
 133: (4)               G.name = "Bull Graph"
 134: (4)               return G
 135: (0)           @_raise_on_directed
 136: (0)           @nx._dispatchable(graphs=None, returns_graph=True)
 137: (0)           def chvatal_graph(create_using=None):
 138: (4)               """
 139: (4)               Returns the Chvátal Graph
 140: (4)               The Chvátal Graph is an undirected graph with 12 nodes and 24 edges [1]_.
 141: (4)               It has 370 distinct (directed) Hamiltonian cycles, giving a unique
generalized
 142: (4)               LCF notation of order 4, two of order 6 , and 43 of order 1 [2]_.
 143: (4)               Parameters
 144: (4)               ----------
 145: (4)               create_using : NetworkX graph constructor, optional (default=nx.Graph)
 146: (7)                  Graph type to create. If graph instance, then cleared before populated.
 147: (4)               Returns
 148: (4)               -------
 149: (4)               G : networkx Graph
 150: (8)                   The Chvátal graph with 12 nodes and 24 edges
 151: (4)               References
 152: (4)               ----------
 153: (4)               .. [1] https://en.wikipedia.org/wiki/Chv%C3%A1tal_graph
 154: (4)               .. [2] https://mathworld.wolfram.com/ChvatalGraph.html
 155: (4)               """
 156: (4)               G = nx.from_dict_of_lists(
 157: (8)                   {
```

```
158: (12)                        0: [1, 4, 6, 9],
159: (12)                        1: [2, 5, 7],
160: (12)                        2: [3, 6, 8],
161: (12)                        3: [4, 7, 9],
162: (12)                        4: [5, 8],
163: (12)                        5: [10, 11],
164: (12)                        6: [10, 11],
165: (12)                        7: [8, 11],
166: (12)                        8: [10],
167: (12)                        9: [10, 11],
168: (8)                     },
169: (8)                     create_using=create_using,
170: (4)                 )
171: (4)                 G.name = "Chvatal Graph"
172: (4)                 return G
173: (0)             @_raise_on_directed
174: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
175: (0)             def cubical_graph(create_using=None):
176: (4)                 """
177: (4)                 Returns the 3-regular Platonic Cubical Graph
178: (4)                 The skeleton of the cube (the nodes and edges) form a graph, with 8
179: (4)                 nodes, and 12 edges. It is a special case of the hypercube graph.
180: (4)                 It is one of 5 Platonic graphs, each a skeleton of its
181: (4)                 Platonic solid [1]_.
182: (4)                 Such graphs arise in parallel processing in computers.
183: (4)                 Parameters
184: (4)                 ----------
185: (4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
186: (7)                    Graph type to create. If graph instance, then cleared before populated.
187: (4)                 Returns
188: (4)                 -------
189: (4)                 G : networkx Graph
190: (8)                     A cubical graph with 8 nodes and 12 edges
191: (4)                 References
192: (4)                 ----------
193: (4)                 .. [1] https://en.wikipedia.org/wiki/Cube#Cubical_graph
194: (4)                 """
195: (4)                 G = nx.from_dict_of_lists(
196: (8)                     {
197: (12)                        0: [1, 3, 4],
198: (12)                        1: [0, 2, 7],
199: (12)                        2: [1, 3, 6],
200: (12)                        3: [0, 2, 5],
201: (12)                        4: [0, 5, 7],
202: (12)                        5: [3, 4, 6],
203: (12)                        6: [2, 5, 7],
204: (12)                        7: [1, 4, 6],
205: (8)                     },
206: (8)                     create_using=create_using,
207: (4)                 )
208: (4)                 G.name = "Platonic Cubical Graph"
209: (4)                 return G
210: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
211: (0)             def desargues_graph(create_using=None):
212: (4)                 """
213: (4)                 Returns the Desargues Graph
214: (4)                 The Desargues Graph is a non-planar, distance-transitive cubic graph
215: (4)                 with 20 nodes and 30 edges [1]_.
216: (4)                 It is a symmetric graph. It can be represented in LCF notation
217: (4)                 as [5,-5,9,-9]^5 [2]_.
218: (4)                 Parameters
219: (4)                 ----------
220: (4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
221: (7)                    Graph type to create. If graph instance, then cleared before populated.
222: (4)                 Returns
223: (4)                 -------
224: (4)                 G : networkx Graph
225: (8)                     Desargues Graph with 20 nodes and 30 edges
226: (4)                 References
```

```
(4)                 ----------
(4)                 .. [1] https://en.wikipedia.org/wiki/Desargues_graph
(4)                 .. [2] https://mathworld.wolfram.com/DesarguesGraph.html
(4)                 """
(4)                 G = LCF_graph(20, [5, -5, 9, -9], 5, create_using)
(4)                 G.name = "Desargues Graph"
(4)                 return G
(0)             @_raise_on_directed
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def diamond_graph(create_using=None):
(4)                 """
(4)                 Returns the Diamond graph
(4)                 The Diamond Graph is  planar undirected graph with 4 nodes and 5 edges.
(4)                 It is also sometimes known as the double triangle graph or kite graph [1]_.
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 G : networkx Graph
(8)                     Diamond Graph with 4 nodes and 5 edges
(4)                 References
(4)                 ----------
(4)                 .. [1] https://mathworld.wolfram.com/DiamondGraph.html
(4)                 """
(4)                 G = nx.from_dict_of_lists(
(8)                     {0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3], 3: [1, 2]}, create_using=create_using
(4)                 )
(4)                 G.name = "Diamond Graph"
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def dodecahedral_graph(create_using=None):
(4)                 """
(4)                 Returns the Platonic Dodecahedral graph.
(4)                 The dodecahedral graph has 20 nodes and 30 edges. The skeleton of the
(4)                 dodecahedron forms a graph. It is one of 5 Platonic graphs [1]_.
(4)                 It can be described in LCF notation as:
(4)                 ``[10, 7, 4, -4, -7, 10, -4, 7, -7, 4]^2`` [2]_.
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 G : networkx Graph
(8)                     Dodecahedral Graph with 20 nodes and 30 edges
(4)                 References
(4)                 ----------
(4)                 .. [1] https://en.wikipedia.org/wiki/Regular_dodecahedron#Dodecahedral_graph
(4)                 .. [2] https://mathworld.wolfram.com/DodecahedralGraph.html
(4)                 """
(4)                 G = LCF_graph(20, [10, 7, 4, -4, -7, 10, -4, 7, -7, 4], 2, create_using)
(4)                 G.name = "Dodecahedral Graph"
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def frucht_graph(create_using=None):
(4)                 """
(4)                 Returns the Frucht Graph.
(4)                 The Frucht Graph is the smallest cubical graph whose
(4)                 automorphism group consists only of the identity element [1]_.
(4)                 It has 12 nodes and 18 edges and no nontrivial symmetries.
(4)                 It is planar and Hamiltonian [2]_.
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
```

```
(7)                 Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Frucht Graph with 12 nodes and 18 edges
(4)             References
(4)             ----------
(4)             .. [1] https://en.wikipedia.org/wiki/Frucht_graph
(4)             .. [2] https://mathworld.wolfram.com/FruchtGraph.html
(4)             """
(4)             G = cycle_graph(7, create_using)
(4)             G.add_edges_from(
(8)                 [
(12)                    [0, 7],
(12)                    [1, 7],
(12)                    [2, 8],
(12)                    [3, 9],
(12)                    [4, 9],
(12)                    [5, 10],
(12)                    [6, 10],
(12)                    [7, 11],
(12)                    [8, 11],
(12)                    [8, 9],
(12)                    [10, 11],
(8)                 ]
(4)             )
(4)             G.name = "Frucht Graph"
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def heawood_graph(create_using=None):
(4)             """
(4)             Returns the Heawood Graph, a (3,6) cage.
(4)             The Heawood Graph is an undirected graph with 14 nodes and 21 edges,
(4)             named after Percy John Heawood [1]_.
(4)             It is cubic symmetric, nonplanar, Hamiltonian, and can be represented
(4)             in LCF notation as ``[5,-5]^7`` [2]_.
(4)             It is the unique (3,6)-cage: the regular cubic graph of girth 6 with
(4)             minimal number of vertices [3]_.
(4)             Parameters
(4)             ----------
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                 Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Heawood Graph with 14 nodes and 21 edges
(4)             References
(4)             ----------
(4)             .. [1] https://en.wikipedia.org/wiki/Heawood_graph
(4)             .. [2] https://mathworld.wolfram.com/HeawoodGraph.html
(4)             .. [3] https://www.win.tue.nl/~aeb/graphs/Heawood.html
(4)             """
(4)             G = LCF_graph(14, [5, -5], 7, create_using)
(4)             G.name = "Heawood Graph"
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def hoffman_singleton_graph():
(4)             """
(4)             Returns the Hoffman-Singleton Graph.
(4)             The Hoffman–Singleton graph is a symmetrical undirected graph
(4)             with 50 nodes and 175 edges.
(4)             All indices lie in ``Z % 5``: that is, the integers mod 5 [1]_.
(4)             It is the only regular graph of vertex degree 7, diameter 2, and girth 5.
(4)             It is the unique (7,5)-cage graph and Moore graph, and contains many
(4)             copies of the Petersen graph [2]_.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Hoffman–Singleton Graph with 50 nodes and 175 edges
```

```
    Notes
    -----
    Constructed from pentagon and pentagram as follows: Take five pentagons $P_h$
    and five pentagrams $Q_i$ . Join vertex $j$ of $P_h$ to vertex $h·i+j$ of $Q_i$ [3]_.
    References
    ----------
    .. [1] https://blogs.ams.org/visualinsight/2016/02/01/hoffman-singleton-graph/
    .. [2] https://mathworld.wolfram.com/Hoffman-SingletonGraph.html
    .. [3] https://en.wikipedia.org/wiki/Hoffman%E2%80%93Singleton_graph
    """
    G = nx.Graph()
    for i in range(5):
        for j in range(5):
            G.add_edge(("pentagon", i, j), ("pentagon", i, (j - 1) % 5))
            G.add_edge(("pentagon", i, j), ("pentagon", i, (j + 1) % 5))
            G.add_edge(("pentagram", i, j), ("pentagram", i, (j - 2) % 5))
            G.add_edge(("pentagram", i, j), ("pentagram", i, (j + 2) % 5))
            for k in range(5):
                G.add_edge(("pentagon", i, j), ("pentagram", k, (i * k + j) % 5))
    G = nx.convert_node_labels_to_integers(G)
    G.name = "Hoffman-Singleton Graph"
    return G
@_raise_on_directed
@nx._dispatchable(graphs=None, returns_graph=True)
def house_graph(create_using=None):
    """
    Returns the House graph (square with triangle on top)
    The house graph is a simple undirected graph with
    5 nodes and 6 edges [1]_.
    Parameters
    ----------
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : networkx Graph
        House graph in the form of a square with a triangle on top
    References
    ----------
    .. [1] https://mathworld.wolfram.com/HouseGraph.html
    """
    G = nx.from_dict_of_lists(
        {0: [1, 2], 1: [0, 3], 2: [0, 3, 4], 3: [1, 2, 4], 4: [2, 3]},
        create_using=create_using,
    )
    G.name = "House Graph"
    return G
@_raise_on_directed
@nx._dispatchable(graphs=None, returns_graph=True)
def house_x_graph(create_using=None):
    """
    Returns the House graph with a cross inside the house square.
    The House X-graph is the House graph plus the two edges connecting diagonally
    opposite vertices of the square base. It is also one of the two graphs
    obtained by removing two edges from the pentatope graph [1]_.
    Parameters
    ----------
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : networkx Graph
        House graph with diagonal vertices connected
    References
```

```
    ----------
    .. [1] https://mathworld.wolfram.com/HouseGraph.html
    """
    G = house_graph(create_using)
    G.add_edges_from([(0, 3), (1, 2)])
    G.name = "House-with-X-inside Graph"
    return G
@_raise_on_directed
@nx._dispatchable(graphs=None, returns_graph=True)
def icosahedral_graph(create_using=None):
    """
    Returns the Platonic Icosahedral graph.
    The icosahedral graph has 12 nodes and 30 edges. It is a Platonic graph
    whose nodes have the connectivity of the icosahedron. It is undirected,
    regular and Hamiltonian [1]_.
    Parameters
    ----------
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : networkx Graph
        Icosahedral graph with 12 nodes and 30 edges.
    References
    ----------
    .. [1] https://mathworld.wolfram.com/IcosahedralGraph.html
    """
    G = nx.from_dict_of_lists(
        {
            0: [1, 5, 7, 8, 11],
            1: [2, 5, 6, 8],
            2: [3, 6, 8, 9],
            3: [4, 6, 9, 10],
            4: [5, 6, 10, 11],
            5: [6, 11],
            7: [8, 9, 10, 11],
            8: [9],
            9: [10],
            10: [11],
        },
        create_using=create_using,
    )
    G.name = "Platonic Icosahedral Graph"
    return G
@_raise_on_directed
@nx._dispatchable(graphs=None, returns_graph=True)
def krackhardt_kite_graph(create_using=None):
    """
    Returns the Krackhardt Kite Social Network.
    A 10 actor social network introduced by David Krackhardt
    to illustrate different centrality measures [1]_.
    Parameters
    ----------
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
       Graph type to create. If graph instance, then cleared before populated.
    Returns
    -------
    G : networkx Graph
        Krackhardt Kite graph with 10 nodes and 18 edges
    Notes
    -----
    The traditional labeling is:
    Andre=1, Beverley=2, Carol=3, Diane=4,
    Ed=5, Fernando=6, Garth=7, Heather=8, Ike=9, Jane=10.
    References
    ----------
    .. [1] Krackhardt, David. "Assessing the Political Landscape: Structure,
       Cognition, and Power in Organizations". Administrative Science
Quarterly.
```

```
(7)                 35 (2): 342–369. doi:10.2307/2393394. JSTOR 2393394. June 1990.
(4)             """
(4)             G = nx.from_dict_of_lists(
(8)                 {
(12)                    0: [1, 2, 3, 5],
(12)                    1: [0, 3, 4, 6],
(12)                    2: [0, 3, 5],
(12)                    3: [0, 1, 2, 4, 5, 6],
(12)                    4: [1, 3, 6],
(12)                    5: [0, 2, 3, 6, 7],
(12)                    6: [1, 3, 4, 5, 7],
(12)                    7: [5, 6, 8],
(12)                    8: [7, 9],
(12)                    9: [8],
(8)                 },
(8)                 create_using=create_using,
(4)             )
(4)             G.name = "Krackhardt Kite Social Network"
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def moebius_kantor_graph(create_using=None):
(4)             """
(4)             Returns the Moebius-Kantor graph.
(4)             The Möbius-Kantor graph is the cubic symmetric graph on 16 nodes.
(4)             Its LCF notation is [5,-5]^8, and it is isomorphic to the generalized
(4)             Petersen graph [1]_.
(4)             Parameters
(4)             ----------
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Moebius-Kantor graph
(4)             References
(4)             ----------
(4)             .. [1] https://en.wikipedia.org/wiki/M%C3%B6bius%E2%80%93Kantor_graph
(4)             """
(4)             G = LCF_graph(16, [5, -5], 8, create_using)
(4)             G.name = "Moebius-Kantor Graph"
(4)             return G
(0)         @_raise_on_directed
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def octahedral_graph(create_using=None):
(4)             """
(4)             Returns the Platonic Octahedral graph.
(4)             The octahedral graph is the 6-node 12-edge Platonic graph having the
(4)             connectivity of the octahedron [1]_. If 6 couples go to a party,
(4)             and each person shakes hands with every person except his or her partner,
(4)             then this graph describes the set of handshakes that take place;
(4)             for this reason it is also called the cocktail party graph [2]_.
(4)             Parameters
(4)             ----------
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Octahedral graph
(4)             References
(4)             ----------
(4)             .. [1] https://mathworld.wolfram.com/OctahedralGraph.html
(4)             .. [2] https://en.wikipedia.org/wiki/Tur%C3%A1n_graph#Special_cases
(4)             """
(4)             G = nx.from_dict_of_lists(
(8)                 {0: [1, 2, 3, 4], 1: [2, 3, 5], 2: [4, 5], 3: [4, 5], 4: [5]},
(8)                 create_using=create_using,
(4)             )
(4)             G.name = "Platonic Octahedral Graph"
```

```
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def pappus_graph():
(4)                 """
(4)                 Returns the Pappus graph.
(4)                 The Pappus graph is a cubic symmetric distance-regular graph with 18 nodes
(4)                 and 27 edges. It is Hamiltonian and can be represented in LCF notation as
(4)                 [5,7,-7,7,-7,-5]^3 [1]_.
(4)                 Returns
(4)                 -------
(4)                 G : networkx Graph
(8)                     Pappus graph
(4)                 References
(4)                 ----------
(4)                 .. [1] https://en.wikipedia.org/wiki/Pappus_graph
(4)                 """
(4)                 G = LCF_graph(18, [5, 7, -7, 7, -7, -5], 3)
(4)                 G.name = "Pappus Graph"
(4)                 return G
(0)             @_raise_on_directed
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def petersen_graph(create_using=None):
(4)                 """
(4)                 Returns the Petersen graph.
(4)                 The Peterson graph is a cubic, undirected graph with 10 nodes and 15 edges
[1]_.
(4)                 Julius Petersen constructed the graph as the smallest counterexample
(4)                 against the claim that a connected bridgeless cubic graph
(4)                 has an edge colouring with three colours [2]_.
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 G : networkx Graph
(8)                     Petersen graph
(4)                 References
(4)                 ----------
(4)                 .. [1] https://en.wikipedia.org/wiki/Petersen_graph
(4)                 .. [2] https://www.win.tue.nl/~aeb/drg/graphs/Petersen.html
(4)                 """
(4)                 G = nx.from_dict_of_lists(
(8)                     {
(12)                        0: [1, 4, 5],
(12)                        1: [0, 2, 6],
(12)                        2: [1, 3, 7],
(12)                        3: [2, 4, 8],
(12)                        4: [3, 0, 9],
(12)                        5: [0, 7, 8],
(12)                        6: [1, 8, 9],
(12)                        7: [2, 5, 9],
(12)                        8: [3, 5, 6],
(12)                        9: [4, 6, 7],
(8)                     },
(8)                     create_using=create_using,
(4)                 )
(4)                 G.name = "Petersen Graph"
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def sedgewick_maze_graph(create_using=None):
(4)                 """
(4)                 Return a small maze with a cycle.
(4)                 This is the maze used in Sedgewick, 3rd Edition, Part 5, Graph
(4)                 Algorithms, Chapter 18, e.g. Figure 18.2 and following [1]_.
(4)                 Nodes are numbered 0,..,7
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
```

```
(7)                 Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Small maze with a cycle
(4)             References
(4)             ----------
(4)             .. [1] Figure 18.2, Chapter 18, Graph Algorithms (3rd Ed), Sedgewick
(4)             """
(4)             G = empty_graph(0, create_using)
(4)             G.add_nodes_from(range(8))
(4)             G.add_edges_from([[0, 2], [0, 7], [0, 5]])
(4)             G.add_edges_from([[1, 7], [2, 6]])
(4)             G.add_edges_from([[3, 4], [3, 5]])
(4)             G.add_edges_from([[4, 5], [4, 7], [4, 6]])
(4)             G.name = "Sedgewick Maze"
(4)             return G
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def tetrahedral_graph(create_using=None):
(4)             """
(4)             Returns the 3-regular Platonic Tetrahedral graph.
(4)             Tetrahedral graph has 4 nodes and 6 edges. It is a
(4)             special case of the complete graph, K4, and wheel graph, W4.
(4)             It is one of the 5 platonic graphs [1]_.
(4)             Parameters
(4)             ----------
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                 Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Tetrahedral Graph
(4)             References
(4)             ----------
(4)             .. [1] https://en.wikipedia.org/wiki/Tetrahedron#Tetrahedral_graph
(4)             """
(4)             G = complete_graph(4, create_using)
(4)             G.name = "Platonic Tetrahedral Graph"
(4)             return G
(0)         @_raise_on_directed
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def truncated_cube_graph(create_using=None):
(4)             """
(4)             Returns the skeleton of the truncated cube.
(4)             The truncated cube is an Archimedean solid with 14 regular
(4)             faces (6 octagonal and 8 triangular), 36 edges and 24 nodes [1]_.
(4)             The truncated cube is created by truncating (cutting off) the tips
(4)             of the cube one third of the way into each edge [2]_.
(4)             Parameters
(4)             ----------
(4)             create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                 Graph type to create. If graph instance, then cleared before populated.
(4)             Returns
(4)             -------
(4)             G : networkx Graph
(8)                 Skeleton of the truncated cube
(4)             References
(4)             ----------
(4)             .. [1] https://en.wikipedia.org/wiki/Truncated_cube
(4)             .. [2] https://www.coolmath.com/reference/polyhedra-truncated-cube
(4)             """
(4)             G = nx.from_dict_of_lists(
(8)                 {
(12)                    0: [1, 2, 4],
(12)                    1: [11, 14],
(12)                    2: [3, 4],
(12)                    3: [6, 8],
(12)                    4: [5],
(12)                    5: [16, 18],
```

```
(12)                        6: [7, 8],
(12)                        7: [10, 12],
(12)                        8: [9],
(12)                        9: [17, 20],
(12)                        10: [11, 12],
(12)                        11: [14],
(12)                        12: [13],
(12)                        13: [21, 22],
(12)                        14: [15],
(12)                        15: [19, 23],
(12)                        16: [17, 18],
(12)                        17: [20],
(12)                        18: [19],
(12)                        19: [23],
(12)                        20: [21],
(12)                        21: [22],
(12)                        22: [23],
(8)                     },
(8)                     create_using=create_using,
(4)                 )
(4)                 G.name = "Truncated Cube Graph"
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def truncated_tetrahedron_graph(create_using=None):
(4)                 """
(4)                 Returns the skeleton of the truncated Platonic tetrahedron.
(4)                 The truncated tetrahedron is an Archimedean solid with 4 regular hexagonal faces,
(4)                 4 equilateral triangle faces, 12 nodes and 18 edges. It can be constructed by truncating
(4)                 all 4 vertices of a regular tetrahedron at one third of the original edge length [1]_.
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 G : networkx Graph
(8)                     Skeleton of the truncated tetrahedron
(4)                 References
(4)                 ----------
(4)                 .. [1] https://en.wikipedia.org/wiki/Truncated_tetrahedron
(4)                 """
(4)                 G = path_graph(12, create_using)
(4)                 G.add_edges_from([(0, 2), (0, 9), (1, 6), (3, 11), (4, 11), (5, 7), (8, 10)])
(4)                 G.name = "Truncated Tetrahedron Graph"
(4)                 return G
(0)             @_raise_on_directed
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def tutte_graph(create_using=None):
(4)                 """
(4)                 Returns the Tutte graph.
(4)                 The Tutte graph is a cubic polyhedral, non-Hamiltonian graph. It has
(4)                 46 nodes and 69 edges.
(4)                 It is a counterexample to Tait's conjecture that every 3-regular polyhedron
(4)                 has a Hamiltonian cycle.
(4)                 It can be realized geometrically from a tetrahedron by multiply truncating
(4)                 three of its vertices [1]_.
(4)                 Parameters
(4)                 ----------
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 Returns
(4)                 -------
(4)                 G : networkx Graph
(8)                     Tutte graph
```

```
764: (4)                References
765: (4)                ----------
766: (4)                .. [1] https://en.wikipedia.org/wiki/Tutte_graph
767: (4)                """
768: (4)                G = nx.from_dict_of_lists(
769: (8)                    {
770: (12)                       0: [1, 2, 3],
771: (12)                       1: [4, 26],
772: (12)                       2: [10, 11],
773: (12)                       3: [18, 19],
774: (12)                       4: [5, 33],
775: (12)                       5: [6, 29],
776: (12)                       6: [7, 27],
777: (12)                       7: [8, 14],
778: (12)                       8: [9, 38],
779: (12)                       9: [10, 37],
780: (12)                       10: [39],
781: (12)                       11: [12, 39],
782: (12)                       12: [13, 35],
783: (12)                       13: [14, 15],
784: (12)                       14: [34],
785: (12)                       15: [16, 22],
786: (12)                       16: [17, 44],
787: (12)                       17: [18, 43],
788: (12)                       18: [45],
789: (12)                       19: [20, 45],
790: (12)                       20: [21, 41],
791: (12)                       21: [22, 23],
792: (12)                       22: [40],
793: (12)                       23: [24, 27],
794: (12)                       24: [25, 32],
795: (12)                       25: [26, 31],
796: (12)                       26: [33],
797: (12)                       27: [28],
798: (12)                       28: [29, 32],
799: (12)                       29: [30],
800: (12)                       30: [31, 33],
801: (12)                       31: [32],
802: (12)                       34: [35, 38],
803: (12)                       35: [36],
804: (12)                       36: [37, 39],
805: (12)                       37: [38],
806: (12)                       40: [41, 44],
807: (12)                       41: [42],
808: (12)                       42: [43, 45],
809: (12)                       43: [44],
810: (8)                    },
811: (8)                    create_using=create_using,
812: (4)                )
813: (4)                G.name = "Tutte's Graph"
814: (4)                return G

----------------------------------------

File 449 - social.py:

1: (0)              """
2: (0)              Famous social networks.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "karate_club_graph",
7: (4)                  "davis_southern_women_graph",
8: (4)                  "florentine_families_graph",
9: (4)                  "les_miserables_graph",
10: (0)             ]
11: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
12: (0)             def karate_club_graph():
13: (4)                 """Returns Zachary's Karate Club graph.
```

```
    Each node in the returned graph has a node attribute 'club' that
    indicates the name of the club to which the member represented by that
node
    belongs, either 'Mr. Hi' or 'Officer'. Each edge has a weight based on the
    number of contexts in which that edge's incident node members interacted.
    Examples
    --------
    To get the name of the club to which a node belongs::
        >>> G = nx.karate_club_graph()
        >>> G.nodes[5]["club"]
        'Mr. Hi'
        >>> G.nodes[9]["club"]
        'Officer'
    References
    ----------
    .. [1] Zachary, Wayne W.
       "An Information Flow Model for Conflict and Fission in Small Groups."
       *Journal of Anthropological Research*, 33, 452--473, (1977).
    """
    all_members = set(range(34))
    club1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 16, 17, 19, 21}
    G = nx.Graph()
    G.add_nodes_from(all_members)
    G.name = "Zachary's Karate Club"
    zacharydat = """\
0 4 5 3 3 3 3 2 2 0 2 3 2 3 0 0 0 2 0 2 0 2 0 0 0 0 0 0 0 0 0 2 0 0
4 0 6 3 0 0 0 4 0 0 0 0 0 5 0 0 0 1 0 2 0 2 0 0 0 0 0 0 0 0 2 0 0 0
5 6 0 3 0 0 0 4 5 1 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 3 0
3 3 3 0 0 0 0 3 0 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 2 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 5 0 0 0 3 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 2 5 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 4 4 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 4 3
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2
2 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 5 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 4
0 0 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2
2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 1
2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 4 0 2 0 0 5 4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 3 0 0 0 2 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 2 0 0 0 0 0 0 7 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 2
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 3 0 0 0 0 0 0 0 0 4
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 4 0 0 0 0 0 3 2
0 2 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 7 0 0 2 0 0 0 4 4
0 0 2 0 0 0 0 0 3 0 0 0 0 0 3 3 0 0 1 0 3 0 2 5 0 0 0 0 0 4 3 4 0 5
0 0 0 0 0 0 0 0 4 2 0 0 0 3 2 4 0 0 2 1 1 0 3 4 0 0 2 4 2 2 3 4 5 0"""
    for row, line in enumerate(zacharydat.split("\n")):
        thisrow = [int(b) for b in line.split()]
        for col, entry in enumerate(thisrow):
            if entry >= 1:
                G.add_edge(row, col, weight=entry)
    for v in G:
        G.nodes[v]["club"] = "Mr. Hi" if v in club1 else "Officer"
    return G
@nx._dispatchable(graphs=None, returns_graph=True)
def davis_southern_women_graph():
```

```
    """Returns Davis Southern women social network.
    This is a bipartite graph.
    References
    ----------
    .. [1] A. Davis, Gardner, B. B., Gardner, M. R., 1941. Deep South.
        University of Chicago Press, Chicago, IL.
    """
    G = nx.Graph()
    women = [
        "Evelyn Jefferson",
        "Laura Mandeville",
        "Theresa Anderson",
        "Brenda Rogers",
        "Charlotte McDowd",
        "Frances Anderson",
        "Eleanor Nye",
        "Pearl Oglethorpe",
        "Ruth DeSand",
        "Verne Sanderson",
        "Myra Liddel",
        "Katherina Rogers",
        "Sylvia Avondale",
        "Nora Fayette",
        "Helen Lloyd",
        "Dorothy Murchison",
        "Olivia Carleton",
        "Flora Price",
    ]
    G.add_nodes_from(women, bipartite=0)
    events = [
        "E1",
        "E2",
        "E3",
        "E4",
        "E5",
        "E6",
        "E7",
        "E8",
        "E9",
        "E10",
        "E11",
        "E12",
        "E13",
        "E14",
    ]
    G.add_nodes_from(events, bipartite=1)
    G.add_edges_from(
        [
            ("Evelyn Jefferson", "E1"),
            ("Evelyn Jefferson", "E2"),
            ("Evelyn Jefferson", "E3"),
            ("Evelyn Jefferson", "E4"),
            ("Evelyn Jefferson", "E5"),
            ("Evelyn Jefferson", "E6"),
            ("Evelyn Jefferson", "E8"),
            ("Evelyn Jefferson", "E9"),
            ("Laura Mandeville", "E1"),
            ("Laura Mandeville", "E2"),
            ("Laura Mandeville", "E3"),
            ("Laura Mandeville", "E5"),
            ("Laura Mandeville", "E6"),
            ("Laura Mandeville", "E7"),
            ("Laura Mandeville", "E8"),
            ("Theresa Anderson", "E2"),
            ("Theresa Anderson", "E3"),
            ("Theresa Anderson", "E4"),
            ("Theresa Anderson", "E5"),
            ("Theresa Anderson", "E6"),
            ("Theresa Anderson", "E7"),
```

```
(12)                        ("Theresa Anderson", "E8"),
(12)                        ("Theresa Anderson", "E9"),
(12)                        ("Brenda Rogers", "E1"),
(12)                        ("Brenda Rogers", "E3"),
(12)                        ("Brenda Rogers", "E4"),
(12)                        ("Brenda Rogers", "E5"),
(12)                        ("Brenda Rogers", "E6"),
(12)                        ("Brenda Rogers", "E7"),
(12)                        ("Brenda Rogers", "E8"),
(12)                        ("Charlotte McDowd", "E3"),
(12)                        ("Charlotte McDowd", "E4"),
(12)                        ("Charlotte McDowd", "E5"),
(12)                        ("Charlotte McDowd", "E7"),
(12)                        ("Frances Anderson", "E3"),
(12)                        ("Frances Anderson", "E5"),
(12)                        ("Frances Anderson", "E6"),
(12)                        ("Frances Anderson", "E8"),
(12)                        ("Eleanor Nye", "E5"),
(12)                        ("Eleanor Nye", "E6"),
(12)                        ("Eleanor Nye", "E7"),
(12)                        ("Eleanor Nye", "E8"),
(12)                        ("Pearl Oglethorpe", "E6"),
(12)                        ("Pearl Oglethorpe", "E8"),
(12)                        ("Pearl Oglethorpe", "E9"),
(12)                        ("Ruth DeSand", "E5"),
(12)                        ("Ruth DeSand", "E7"),
(12)                        ("Ruth DeSand", "E8"),
(12)                        ("Ruth DeSand", "E9"),
(12)                        ("Verne Sanderson", "E7"),
(12)                        ("Verne Sanderson", "E8"),
(12)                        ("Verne Sanderson", "E9"),
(12)                        ("Verne Sanderson", "E12"),
(12)                        ("Myra Liddel", "E8"),
(12)                        ("Myra Liddel", "E9"),
(12)                        ("Myra Liddel", "E10"),
(12)                        ("Myra Liddel", "E12"),
(12)                        ("Katherina Rogers", "E8"),
(12)                        ("Katherina Rogers", "E9"),
(12)                        ("Katherina Rogers", "E10"),
(12)                        ("Katherina Rogers", "E12"),
(12)                        ("Katherina Rogers", "E13"),
(12)                        ("Katherina Rogers", "E14"),
(12)                        ("Sylvia Avondale", "E7"),
(12)                        ("Sylvia Avondale", "E8"),
(12)                        ("Sylvia Avondale", "E9"),
(12)                        ("Sylvia Avondale", "E10"),
(12)                        ("Sylvia Avondale", "E12"),
(12)                        ("Sylvia Avondale", "E13"),
(12)                        ("Sylvia Avondale", "E14"),
(12)                        ("Nora Fayette", "E6"),
(12)                        ("Nora Fayette", "E7"),
(12)                        ("Nora Fayette", "E9"),
(12)                        ("Nora Fayette", "E10"),
(12)                        ("Nora Fayette", "E11"),
(12)                        ("Nora Fayette", "E12"),
(12)                        ("Nora Fayette", "E13"),
(12)                        ("Nora Fayette", "E14"),
(12)                        ("Helen Lloyd", "E7"),
(12)                        ("Helen Lloyd", "E8"),
(12)                        ("Helen Lloyd", "E10"),
(12)                        ("Helen Lloyd", "E11"),
(12)                        ("Helen Lloyd", "E12"),
(12)                        ("Dorothy Murchison", "E8"),
(12)                        ("Dorothy Murchison", "E9"),
(12)                        ("Olivia Carleton", "E9"),
(12)                        ("Olivia Carleton", "E11"),
(12)                        ("Flora Price", "E9"),
(12)                        ("Flora Price", "E11"),
(8)                     ]
```

```
220: (4)                )
221: (4)                G.graph["top"] = women
222: (4)                G.graph["bottom"] = events
223: (4)                return G
224: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
225: (0)            def florentine_families_graph():
226: (4)                """Returns Florentine families graph.
227: (4)                References
228: (4)                ----------
229: (4)                .. [1] Ronald L. Breiger and Philippa E. Pattison
230: (7)                   Cumulated social roles: The duality of persons and their algebras,1
231: (7)                   Social Networks, Volume 8, Issue 3, September 1986, Pages 215-256
232: (4)                """
233: (4)                G = nx.Graph()
234: (4)                G.add_edge("Acciaiuoli", "Medici")
235: (4)                G.add_edge("Castellani", "Peruzzi")
236: (4)                G.add_edge("Castellani", "Strozzi")
237: (4)                G.add_edge("Castellani", "Barbadori")
238: (4)                G.add_edge("Medici", "Barbadori")
239: (4)                G.add_edge("Medici", "Ridolfi")
240: (4)                G.add_edge("Medici", "Tornabuoni")
241: (4)                G.add_edge("Medici", "Albizzi")
242: (4)                G.add_edge("Medici", "Salviati")
243: (4)                G.add_edge("Salviati", "Pazzi")
244: (4)                G.add_edge("Peruzzi", "Strozzi")
245: (4)                G.add_edge("Peruzzi", "Bischeri")
246: (4)                G.add_edge("Strozzi", "Ridolfi")
247: (4)                G.add_edge("Strozzi", "Bischeri")
248: (4)                G.add_edge("Ridolfi", "Tornabuoni")
249: (4)                G.add_edge("Tornabuoni", "Guadagni")
250: (4)                G.add_edge("Albizzi", "Ginori")
251: (4)                G.add_edge("Albizzi", "Guadagni")
252: (4)                G.add_edge("Bischeri", "Guadagni")
253: (4)                G.add_edge("Guadagni", "Lamberteschi")
254: (4)                return G
255: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
256: (0)            def les_miserables_graph():
257: (4)                """Returns coappearance network of characters in the novel Les Miserables.
258: (4)                References
259: (4)                ----------
260: (4)                .. [1] D. E. Knuth, 1993.
261: (7)                   The Stanford GraphBase: a platform for combinatorial computing,
262: (7)                   pp. 74-87. New York: AcM Press.
263: (4)                """
264: (4)                G = nx.Graph()
265: (4)                G.add_edge("Napoleon", "Myriel", weight=1)
266: (4)                G.add_edge("MlleBaptistine", "Myriel", weight=8)
267: (4)                G.add_edge("MmeMagloire", "Myriel", weight=10)
268: (4)                G.add_edge("MmeMagloire", "MlleBaptistine", weight=6)
269: (4)                G.add_edge("CountessDeLo", "Myriel", weight=1)
270: (4)                G.add_edge("Geborand", "Myriel", weight=1)
271: (4)                G.add_edge("Champtercier", "Myriel", weight=1)
272: (4)                G.add_edge("Cravatte", "Myriel", weight=1)
273: (4)                G.add_edge("Count", "Myriel", weight=2)
274: (4)                G.add_edge("OldMan", "Myriel", weight=1)
275: (4)                G.add_edge("Valjean", "Labarre", weight=1)
276: (4)                G.add_edge("Valjean", "MmeMagloire", weight=3)
277: (4)                G.add_edge("Valjean", "MlleBaptistine", weight=3)
278: (4)                G.add_edge("Valjean", "Myriel", weight=5)
279: (4)                G.add_edge("Marguerite", "Valjean", weight=1)
280: (4)                G.add_edge("MmeDeR", "Valjean", weight=1)
281: (4)                G.add_edge("Isabeau", "Valjean", weight=1)
282: (4)                G.add_edge("Gervais", "Valjean", weight=1)
283: (4)                G.add_edge("Listolier", "Tholomyes", weight=4)
284: (4)                G.add_edge("Fameuil", "Tholomyes", weight=4)
285: (4)                G.add_edge("Fameuil", "Listolier", weight=4)
286: (4)                G.add_edge("Blacheville", "Tholomyes", weight=4)
287: (4)                G.add_edge("Blacheville", "Listolier", weight=4)
288: (4)                G.add_edge("Blacheville", "Fameuil", weight=4)
```

```
(4)                 G.add_edge("Favourite", "Tholomyes", weight=3)
(4)                 G.add_edge("Favourite", "Listolier", weight=3)
(4)                 G.add_edge("Favourite", "Fameuil", weight=3)
(4)                 G.add_edge("Favourite", "Blacheville", weight=4)
(4)                 G.add_edge("Dahlia", "Tholomyes", weight=3)
(4)                 G.add_edge("Dahlia", "Listolier", weight=3)
(4)                 G.add_edge("Dahlia", "Fameuil", weight=3)
(4)                 G.add_edge("Dahlia", "Blacheville", weight=3)
(4)                 G.add_edge("Dahlia", "Favourite", weight=5)
(4)                 G.add_edge("Zephine", "Tholomyes", weight=3)
(4)                 G.add_edge("Zephine", "Listolier", weight=3)
(4)                 G.add_edge("Zephine", "Fameuil", weight=3)
(4)                 G.add_edge("Zephine", "Blacheville", weight=3)
(4)                 G.add_edge("Zephine", "Favourite", weight=4)
(4)                 G.add_edge("Zephine", "Dahlia", weight=4)
(4)                 G.add_edge("Fantine", "Tholomyes", weight=3)
(4)                 G.add_edge("Fantine", "Listolier", weight=3)
(4)                 G.add_edge("Fantine", "Fameuil", weight=3)
(4)                 G.add_edge("Fantine", "Blacheville", weight=3)
(4)                 G.add_edge("Fantine", "Favourite", weight=4)
(4)                 G.add_edge("Fantine", "Dahlia", weight=4)
(4)                 G.add_edge("Fantine", "Zephine", weight=4)
(4)                 G.add_edge("Fantine", "Marguerite", weight=2)
(4)                 G.add_edge("Fantine", "Valjean", weight=9)
(4)                 G.add_edge("MmeThenardier", "Fantine", weight=2)
(4)                 G.add_edge("MmeThenardier", "Valjean", weight=7)
(4)                 G.add_edge("Thenardier", "MmeThenardier", weight=13)
(4)                 G.add_edge("Thenardier", "Fantine", weight=1)
(4)                 G.add_edge("Thenardier", "Valjean", weight=12)
(4)                 G.add_edge("Cosette", "MmeThenardier", weight=4)
(4)                 G.add_edge("Cosette", "Valjean", weight=31)
(4)                 G.add_edge("Cosette", "Tholomyes", weight=1)
(4)                 G.add_edge("Cosette", "Thenardier", weight=1)
(4)                 G.add_edge("Javert", "Valjean", weight=17)
(4)                 G.add_edge("Javert", "Fantine", weight=5)
(4)                 G.add_edge("Javert", "Thenardier", weight=5)
(4)                 G.add_edge("Javert", "MmeThenardier", weight=1)
(4)                 G.add_edge("Javert", "Cosette", weight=1)
(4)                 G.add_edge("Fauchelevent", "Valjean", weight=8)
(4)                 G.add_edge("Fauchelevent", "Javert", weight=1)
(4)                 G.add_edge("Bamatabois", "Fantine", weight=1)
(4)                 G.add_edge("Bamatabois", "Javert", weight=1)
(4)                 G.add_edge("Bamatabois", "Valjean", weight=2)
(4)                 G.add_edge("Perpetue", "Fantine", weight=1)
(4)                 G.add_edge("Simplice", "Perpetue", weight=2)
(4)                 G.add_edge("Simplice", "Valjean", weight=3)
(4)                 G.add_edge("Simplice", "Fantine", weight=2)
(4)                 G.add_edge("Simplice", "Javert", weight=1)
(4)                 G.add_edge("Scaufflaire", "Valjean", weight=1)
(4)                 G.add_edge("Woman1", "Valjean", weight=2)
(4)                 G.add_edge("Woman1", "Javert", weight=1)
(4)                 G.add_edge("Judge", "Valjean", weight=3)
(4)                 G.add_edge("Judge", "Bamatabois", weight=2)
(4)                 G.add_edge("Champmathieu", "Valjean", weight=3)
(4)                 G.add_edge("Champmathieu", "Judge", weight=3)
(4)                 G.add_edge("Champmathieu", "Bamatabois", weight=2)
(4)                 G.add_edge("Brevet", "Judge", weight=2)
(4)                 G.add_edge("Brevet", "Champmathieu", weight=2)
(4)                 G.add_edge("Brevet", "Valjean", weight=2)
(4)                 G.add_edge("Brevet", "Bamatabois", weight=1)
(4)                 G.add_edge("Chenildieu", "Judge", weight=2)
(4)                 G.add_edge("Chenildieu", "Champmathieu", weight=2)
(4)                 G.add_edge("Chenildieu", "Brevet", weight=2)
(4)                 G.add_edge("Chenildieu", "Valjean", weight=2)
(4)                 G.add_edge("Chenildieu", "Bamatabois", weight=1)
(4)                 G.add_edge("Cochepaille", "Judge", weight=2)
(4)                 G.add_edge("Cochepaille", "Champmathieu", weight=2)
(4)                 G.add_edge("Cochepaille", "Brevet", weight=2)
(4)                 G.add_edge("Cochepaille", "Chenildieu", weight=2)
```

```
(4)                 G.add_edge("Cochepaille", "Valjean", weight=2)
(4)                 G.add_edge("Cochepaille", "Bamatabois", weight=1)
(4)                 G.add_edge("Pontmercy", "Thenardier", weight=1)
(4)                 G.add_edge("Boulatruelle", "Thenardier", weight=1)
(4)                 G.add_edge("Eponine", "MmeThenardier", weight=2)
(4)                 G.add_edge("Eponine", "Thenardier", weight=3)
(4)                 G.add_edge("Anzelma", "Eponine", weight=2)
(4)                 G.add_edge("Anzelma", "Thenardier", weight=2)
(4)                 G.add_edge("Anzelma", "MmeThenardier", weight=1)
(4)                 G.add_edge("Woman2", "Valjean", weight=3)
(4)                 G.add_edge("Woman2", "Cosette", weight=1)
(4)                 G.add_edge("Woman2", "Javert", weight=1)
(4)                 G.add_edge("MotherInnocent", "Fauchelevent", weight=3)
(4)                 G.add_edge("MotherInnocent", "Valjean", weight=1)
(4)                 G.add_edge("Gribier", "Fauchelevent", weight=2)
(4)                 G.add_edge("MmeBurgon", "Jondrette", weight=1)
(4)                 G.add_edge("Gavroche", "MmeBurgon", weight=2)
(4)                 G.add_edge("Gavroche", "Thenardier", weight=1)
(4)                 G.add_edge("Gavroche", "Javert", weight=1)
(4)                 G.add_edge("Gavroche", "Valjean", weight=1)
(4)                 G.add_edge("Gillenormand", "Cosette", weight=3)
(4)                 G.add_edge("Gillenormand", "Valjean", weight=2)
(4)                 G.add_edge("Magnon", "Gillenormand", weight=1)
(4)                 G.add_edge("Magnon", "MmeThenardier", weight=1)
(4)                 G.add_edge("MlleGillenormand", "Gillenormand", weight=9)
(4)                 G.add_edge("MlleGillenormand", "Cosette", weight=2)
(4)                 G.add_edge("MlleGillenormand", "Valjean", weight=2)
(4)                 G.add_edge("MmePontmercy", "MlleGillenormand", weight=1)
(4)                 G.add_edge("MmePontmercy", "Pontmercy", weight=1)
(4)                 G.add_edge("MlleVaubois", "MlleGillenormand", weight=1)
(4)                 G.add_edge("LtGillenormand", "MlleGillenormand", weight=2)
(4)                 G.add_edge("LtGillenormand", "Gillenormand", weight=1)
(4)                 G.add_edge("LtGillenormand", "Cosette", weight=1)
(4)                 G.add_edge("Marius", "MlleGillenormand", weight=6)
(4)                 G.add_edge("Marius", "Gillenormand", weight=12)
(4)                 G.add_edge("Marius", "Pontmercy", weight=1)
(4)                 G.add_edge("Marius", "LtGillenormand", weight=1)
(4)                 G.add_edge("Marius", "Cosette", weight=21)
(4)                 G.add_edge("Marius", "Valjean", weight=19)
(4)                 G.add_edge("Marius", "Tholomyes", weight=1)
(4)                 G.add_edge("Marius", "Thenardier", weight=2)
(4)                 G.add_edge("Marius", "Eponine", weight=5)
(4)                 G.add_edge("Marius", "Gavroche", weight=4)
(4)                 G.add_edge("BaronessT", "Gillenormand", weight=1)
(4)                 G.add_edge("BaronessT", "Marius", weight=1)
(4)                 G.add_edge("Mabeuf", "Marius", weight=1)
(4)                 G.add_edge("Mabeuf", "Eponine", weight=1)
(4)                 G.add_edge("Mabeuf", "Gavroche", weight=1)
(4)                 G.add_edge("Enjolras", "Marius", weight=7)
(4)                 G.add_edge("Enjolras", "Gavroche", weight=7)
(4)                 G.add_edge("Enjolras", "Javert", weight=6)
(4)                 G.add_edge("Enjolras", "Mabeuf", weight=1)
(4)                 G.add_edge("Enjolras", "Valjean", weight=4)
(4)                 G.add_edge("Combeferre", "Enjolras", weight=15)
(4)                 G.add_edge("Combeferre", "Marius", weight=5)
(4)                 G.add_edge("Combeferre", "Gavroche", weight=6)
(4)                 G.add_edge("Combeferre", "Mabeuf", weight=2)
(4)                 G.add_edge("Prouvaire", "Gavroche", weight=1)
(4)                 G.add_edge("Prouvaire", "Enjolras", weight=4)
(4)                 G.add_edge("Prouvaire", "Combeferre", weight=2)
(4)                 G.add_edge("Feuilly", "Gavroche", weight=2)
(4)                 G.add_edge("Feuilly", "Enjolras", weight=6)
(4)                 G.add_edge("Feuilly", "Prouvaire", weight=2)
(4)                 G.add_edge("Feuilly", "Combeferre", weight=5)
(4)                 G.add_edge("Feuilly", "Mabeuf", weight=1)
(4)                 G.add_edge("Feuilly", "Marius", weight=1)
(4)                 G.add_edge("Courfeyrac", "Marius", weight=9)
(4)                 G.add_edge("Courfeyrac", "Enjolras", weight=17)
(4)                 G.add_edge("Courfeyrac", "Combeferre", weight=13)
```

```
(4)                 G.add_edge("Courfeyrac", "Gavroche", weight=7)
(4)                 G.add_edge("Courfeyrac", "Mabeuf", weight=2)
(4)                 G.add_edge("Courfeyrac", "Eponine", weight=1)
(4)                 G.add_edge("Courfeyrac", "Feuilly", weight=6)
(4)                 G.add_edge("Courfeyrac", "Prouvaire", weight=3)
(4)                 G.add_edge("Bahorel", "Combeferre", weight=5)
(4)                 G.add_edge("Bahorel", "Gavroche", weight=5)
(4)                 G.add_edge("Bahorel", "Courfeyrac", weight=6)
(4)                 G.add_edge("Bahorel", "Mabeuf", weight=2)
(4)                 G.add_edge("Bahorel", "Enjolras", weight=4)
(4)                 G.add_edge("Bahorel", "Feuilly", weight=3)
(4)                 G.add_edge("Bahorel", "Prouvaire", weight=2)
(4)                 G.add_edge("Bahorel", "Marius", weight=1)
(4)                 G.add_edge("Bossuet", "Marius", weight=5)
(4)                 G.add_edge("Bossuet", "Courfeyrac", weight=12)
(4)                 G.add_edge("Bossuet", "Gavroche", weight=5)
(4)                 G.add_edge("Bossuet", "Bahorel", weight=4)
(4)                 G.add_edge("Bossuet", "Enjolras", weight=10)
(4)                 G.add_edge("Bossuet", "Feuilly", weight=6)
(4)                 G.add_edge("Bossuet", "Prouvaire", weight=2)
(4)                 G.add_edge("Bossuet", "Combeferre", weight=9)
(4)                 G.add_edge("Bossuet", "Mabeuf", weight=1)
(4)                 G.add_edge("Bossuet", "Valjean", weight=1)
(4)                 G.add_edge("Joly", "Bahorel", weight=5)
(4)                 G.add_edge("Joly", "Bossuet", weight=7)
(4)                 G.add_edge("Joly", "Gavroche", weight=3)
(4)                 G.add_edge("Joly", "Courfeyrac", weight=5)
(4)                 G.add_edge("Joly", "Enjolras", weight=5)
(4)                 G.add_edge("Joly", "Feuilly", weight=5)
(4)                 G.add_edge("Joly", "Prouvaire", weight=2)
(4)                 G.add_edge("Joly", "Combeferre", weight=5)
(4)                 G.add_edge("Joly", "Mabeuf", weight=1)
(4)                 G.add_edge("Joly", "Marius", weight=2)
(4)                 G.add_edge("Grantaire", "Bossuet", weight=3)
(4)                 G.add_edge("Grantaire", "Enjolras", weight=3)
(4)                 G.add_edge("Grantaire", "Combeferre", weight=1)
(4)                 G.add_edge("Grantaire", "Courfeyrac", weight=2)
(4)                 G.add_edge("Grantaire", "Joly", weight=2)
(4)                 G.add_edge("Grantaire", "Gavroche", weight=1)
(4)                 G.add_edge("Grantaire", "Bahorel", weight=1)
(4)                 G.add_edge("Grantaire", "Feuilly", weight=1)
(4)                 G.add_edge("Grantaire", "Prouvaire", weight=1)
(4)                 G.add_edge("MotherPlutarch", "Mabeuf", weight=3)
(4)                 G.add_edge("Gueulemer", "Thenardier", weight=5)
(4)                 G.add_edge("Gueulemer", "Valjean", weight=1)
(4)                 G.add_edge("Gueulemer", "MmeThenardier", weight=1)
(4)                 G.add_edge("Gueulemer", "Javert", weight=1)
(4)                 G.add_edge("Gueulemer", "Gavroche", weight=1)
(4)                 G.add_edge("Gueulemer", "Eponine", weight=1)
(4)                 G.add_edge("Babet", "Thenardier", weight=6)
(4)                 G.add_edge("Babet", "Gueulemer", weight=6)
(4)                 G.add_edge("Babet", "Valjean", weight=1)
(4)                 G.add_edge("Babet", "MmeThenardier", weight=1)
(4)                 G.add_edge("Babet", "Javert", weight=2)
(4)                 G.add_edge("Babet", "Gavroche", weight=1)
(4)                 G.add_edge("Babet", "Eponine", weight=1)
(4)                 G.add_edge("Claquesous", "Thenardier", weight=4)
(4)                 G.add_edge("Claquesous", "Babet", weight=4)
(4)                 G.add_edge("Claquesous", "Gueulemer", weight=4)
(4)                 G.add_edge("Claquesous", "Valjean", weight=1)
(4)                 G.add_edge("Claquesous", "MmeThenardier", weight=1)
(4)                 G.add_edge("Claquesous", "Javert", weight=1)
(4)                 G.add_edge("Claquesous", "Eponine", weight=1)
(4)                 G.add_edge("Claquesous", "Enjolras", weight=1)
(4)                 G.add_edge("Montparnasse", "Javert", weight=1)
(4)                 G.add_edge("Montparnasse", "Babet", weight=2)
(4)                 G.add_edge("Montparnasse", "Gueulemer", weight=2)
(4)                 G.add_edge("Montparnasse", "Claquesous", weight=2)
(4)                 G.add_edge("Montparnasse", "Valjean", weight=1)
```

```
496: (4)                G.add_edge("Montparnasse", "Gavroche", weight=1)
497: (4)                G.add_edge("Montparnasse", "Eponine", weight=1)
498: (4)                G.add_edge("Montparnasse", "Thenardier", weight=1)
499: (4)                G.add_edge("Toussaint", "Cosette", weight=2)
500: (4)                G.add_edge("Toussaint", "Javert", weight=1)
501: (4)                G.add_edge("Toussaint", "Valjean", weight=1)
502: (4)                G.add_edge("Child1", "Gavroche", weight=2)
503: (4)                G.add_edge("Child2", "Gavroche", weight=2)
504: (4)                G.add_edge("Child2", "Child1", weight=3)
505: (4)                G.add_edge("Brujon", "Babet", weight=3)
506: (4)                G.add_edge("Brujon", "Gueulemer", weight=3)
507: (4)                G.add_edge("Brujon", "Thenardier", weight=3)
508: (4)                G.add_edge("Brujon", "Gavroche", weight=1)
509: (4)                G.add_edge("Brujon", "Eponine", weight=1)
510: (4)                G.add_edge("Brujon", "Claquesous", weight=1)
511: (4)                G.add_edge("Brujon", "Montparnasse", weight=1)
512: (4)                G.add_edge("MmeHucheloup", "Bossuet", weight=1)
513: (4)                G.add_edge("MmeHucheloup", "Joly", weight=1)
514: (4)                G.add_edge("MmeHucheloup", "Grantaire", weight=1)
515: (4)                G.add_edge("MmeHucheloup", "Bahorel", weight=1)
516: (4)                G.add_edge("MmeHucheloup", "Courfeyrac", weight=1)
517: (4)                G.add_edge("MmeHucheloup", "Gavroche", weight=1)
518: (4)                G.add_edge("MmeHucheloup", "Enjolras", weight=1)
519: (4)                return G

----------------------------------------

File 450 - spectral_graph_forge.py:

1: (0)              """Generates graphs with a given eigenvector structure"""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import np_random_state
4: (0)              __all__ = ["spectral_graph_forge"]
5: (0)              @np_random_state(3)
6: (0)              @nx._dispatchable(returns_graph=True)
7: (0)              def spectral_graph_forge(G, alpha, transformation="identity", seed=None):
8: (4)                  """Returns a random simple graph with spectrum resembling that of `G`
9: (4)                  This algorithm, called Spectral Graph Forge (SGF), computes the
10: (4)                 eigenvectors of a given graph adjacency matrix, filters them and
11: (4)                 builds a random graph with a similar eigenstructure.
12: (4)                 SGF has been proved to be particularly useful for synthesizing
13: (4)                 realistic social networks and it can also be used to anonymize
14: (4)                 graph sensitive data.
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 G : Graph
18: (4)                 alpha :  float
19: (8)                     Ratio representing the percentage of eigenvectors of G to consider,
20: (8)                     values in [0,1].
21: (4)                 transformation : string, optional
22: (8)                     Represents the intended matrix linear transformation, possible values
23: (8)                     are 'identity' and 'modularity'
24: (4)                 seed : integer, random_state, or None (default)
25: (8)                     Indicator of numpy random number generation state.
26: (8)                     See :ref:`Randomness<randomness>`.
27: (4)                 Returns
28: (4)                 -------
29: (4)                 H : Graph
30: (8)                     A graph with a similar eigenvector structure of the input one.
31: (4)                 Raises
32: (4)                 ------
33: (4)                 NetworkXError
34: (8)                     If transformation has a value different from 'identity' or
'modularity'
35: (4)                 Notes
36: (4)                 -----
37: (4)                 Spectral Graph Forge (SGF) generates a random simple graph resembling the
38: (4)                 global properties of the given one.
39: (4)                 It leverages the low-rank approximation of the associated adjacency matrix
```

```
40: (4)                 driven by the *alpha* precision parameter.
41: (4)                 SGF preserves the number of nodes of the input graph and their ordering.
42: (4)                 This way, nodes of output graphs resemble the properties of the input one
43: (4)                 and attributes can be directly mapped.
44: (4)                 It considers the graph adjacency matrices which can optionally be
45: (4)                 transformed to other symmetric real matrices (currently transformation
46: (4)                 options include *identity* and *modularity*).
47: (4)                 The *modularity* transformation, in the sense of Newman's modularity
matrix
48: (4)                 allows the focusing on community structure related properties of the
graph.
49: (4)                 SGF applies a low-rank approximation whose fixed rank is computed from the
50: (4)                 ratio *alpha* of the input graph adjacency matrix dimension.
51: (4)                 This step performs a filtering on the input eigenvectors similar to the
low
52: (4)                 pass filtering common in telecommunications.
53: (4)                 The filtered values (after truncation) are used as input to a Bernoulli
54: (4)                 sampling for constructing a random adjacency matrix.
55: (4)                 References
56: (4)                 ----------
57: (4)                 ..  [1] L. Baldesi, C. T. Butts, A. Markopoulou, "Spectral Graph Forge:
58: (8)                     Graph Generation Targeting Modularity", IEEE Infocom, '18.
59: (8)                     https://arxiv.org/abs/1801.01715
60: (4)                 ..  [2] M. Newman, "Networks: an introduction", Oxford university press,
61: (8)                     2010
62: (4)                 Examples
63: (4)                 --------
64: (4)                 >>> G = nx.karate_club_graph()
65: (4)                 >>> H = nx.spectral_graph_forge(G, 0.3)
66: (4)                 >>>
67: (4)                 """
68: (4)                 import numpy as np
69: (4)                 import scipy as sp
70: (4)                 available_transformations = ["identity", "modularity"]
71: (4)                 alpha = np.clip(alpha, 0, 1)
72: (4)                 A = nx.to_numpy_array(G)
73: (4)                 n = A.shape[1]
74: (4)                 level = round(n * alpha)
75: (4)                 if transformation not in available_transformations:
76: (8)                     msg = f"{transformation!r} is not a valid transformation. "
77: (8)                     msg += f"Transformations: {available_transformations}"
78: (8)                     raise nx.NetworkXError(msg)
79: (4)                 K = np.ones((1, n)) @ A
80: (4)                 B = A
81: (4)                 if transformation == "modularity":
82: (8)                     B -= K.T @ K / K.sum()
83: (4)                 evals, evecs = np.linalg.eigh(B)
84: (4)                 k = np.argsort(np.abs(evals))[::-1]  # indices of evals in descending
order
85: (4)                 evecs[:, k[np.arange(level, n)]] = 0  # set smallest eigenvectors to 0
86: (4)                 B = evecs @ np.diag(evals) @ evecs.T
87: (4)                 if transformation == "modularity":
88: (8)                     B += K.T @ K / K.sum()
89: (4)                 B = np.clip(B, 0, 1)
90: (4)                 np.fill_diagonal(B, 0)
91: (4)                 for i in range(n - 1):
92: (8)                     B[i, i + 1 :] = sp.stats.bernoulli.rvs(B[i, i + 1 :],
random_state=seed)
93: (8)                     B[i + 1 :, i] = np.transpose(B[i, i + 1 :])
94: (4)                 H = nx.from_numpy_array(B)
95: (4)                 return H

----------------------------------------

File 451 - stochastic.py:

1: (0)              """Functions for generating stochastic graphs from a given weighted directed
2: (0)              graph.
3: (0)              """
```

```
import networkx as nx
from networkx.classes import DiGraph, MultiDiGraph
from networkx.utils import not_implemented_for
__all__ = ["stochastic_graph"]
@not_implemented_for("undirected")
@nx._dispatchable(
    edge_attrs="weight", mutates_input={"not copy": 1}, returns_graph=True
)
def stochastic_graph(G, copy=True, weight="weight"):
    """Returns a right-stochastic representation of directed graph `G`.
    A right-stochastic graph is a weighted digraph in which for each
    node, the sum of the weights of all the out-edges of that node is
    1. If the graph is already weighted (for example, via a 'weight'
    edge attribute), the reweighting takes that into account.
    Parameters
    ----------
    G : directed graph
        A :class:`~networkx.DiGraph` or :class:`~networkx.MultiDiGraph`.
    copy : boolean, optional
        If this is True, then this function returns a new graph with
        the stochastic reweighting. Otherwise, the original graph is
        modified in-place (and also returned, for convenience).
    weight : edge attribute key (optional, default='weight')
        Edge attribute key used for reading the existing weight and
        setting the new weight.  If no attribute with this key is found
        for an edge, then the edge weight is assumed to be 1. If an edge
        has a weight, it must be a positive number.
    """
    if copy:
        G = MultiDiGraph(G) if G.is_multigraph() else DiGraph(G)
    degree = dict(G.out_degree(weight=weight))
    for u, v, d in G.edges(data=True):
        if degree[u] == 0:
            d[weight] = 0
        else:
            d[weight] = d.get(weight, 1) / degree[u]
    nx._clear_cache(G)
    return G
```

File 452 - sudoku.py:

```
"""Generator for Sudoku graphs
This module gives a generator for n-Sudoku graphs. It can be used to develop
algorithms for solving or generating Sudoku puzzles.
A completed Sudoku grid is a 9x9 array of integers between 1 and 9, with no
number appearing twice in the same row, column, or 3x3 box.
+---------+---------+---------+
| | 8 6 4 | | 3 7 1 | | 2 5 9 |
| | 3 2 5 | | 8 4 9 | | 7 6 1 |
| | 9 7 1 | | 2 6 5 | | 8 4 3 |
+---------+---------+---------+
| | 4 3 6 | | 1 9 2 | | 5 8 7 |
| | 1 9 8 | | 6 5 7 | | 4 3 2 |
| | 2 5 7 | | 4 8 3 | | 9 1 6 |
+---------+---------+---------+
| | 6 8 9 | | 7 3 4 | | 1 2 5 |
| | 7 1 3 | | 5 2 8 | | 6 9 4 |
| | 5 4 2 | | 9 1 6 | | 3 7 8 |
+---------+---------+---------+
The Sudoku graph is an undirected graph with 81 vertices, corresponding to
the cells of a Sudoku grid. It is a regular graph of degree 20. Two distinct
vertices are adjacent if and only if the corresponding cells belong to the
same row, column, or box. A completed Sudoku grid corresponds to a vertex
coloring of the Sudoku graph with nine colors.
More generally, the n-Sudoku graph is a graph with n^4 vertices, corresponding
to the cells of an n^2 by n^2 grid. Two distinct vertices are adjacent if and
only if they belong to the same row, column, or n by n box.
```

```
 27: (0)             References
 28: (0)             ----------
 29: (0)             .. [1] Herzberg, A. M., & Murty, M. R. (2007). Sudoku squares and chromatic
 30: (4)                 polynomials. Notices of the AMS, 54(6), 708-717.
 31: (0)             .. [2] Sander, Torsten (2009), "Sudoku graphs are integral",
 32: (4)                 Electronic Journal of Combinatorics, 16 (1): Note 25, 7pp, MR 2529816
 33: (0)             .. [3] Wikipedia contributors. "Glossary of Sudoku." Wikipedia, The Free
 34: (4)                 Encyclopedia, 3 Dec. 2019. Web. 22 Dec. 2019.
 35: (0)             """
 36: (0)             import networkx as nx
 37: (0)             from networkx.exception import NetworkXError
 38: (0)             __all__ = ["sudoku_graph"]
 39: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 40: (0)             def sudoku_graph(n=3):
 41: (4)                 """Returns the n-Sudoku graph. The default value of n is 3.
 42: (4)                 The n-Sudoku graph is a graph with n^4 vertices, corresponding to the
 43: (4)                 cells of an n^2 by n^2 grid. Two distinct vertices are adjacent if and
 44: (4)                 only if they belong to the same row, column, or n-by-n box.
 45: (4)                 Parameters
 46: (4)                 ----------
 47: (4)                 n: integer
 48: (7)                    The order of the Sudoku graph, equal to the square root of the
 49: (7)                    number of rows. The default is 3.
 50: (4)                 Returns
 51: (4)                 -------
 52: (4)                 NetworkX graph
 53: (8)                     The n-Sudoku graph Sud(n).
 54: (4)                 Examples
 55: (4)                 --------
 56: (4)                 >>> G = nx.sudoku_graph()
 57: (4)                 >>> G.number_of_nodes()
 58: (4)                 81
 59: (4)                 >>> G.number_of_edges()
 60: (4)                 810
 61: (4)                 >>> sorted(G.neighbors(42))
 62: (4)                 [6, 15, 24, 33, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 51, 52, 53, 60,
 69, 78]
 63: (4)                 >>> G = nx.sudoku_graph(2)
 64: (4)                 >>> G.number_of_nodes()
 65: (4)                 16
 66: (4)                 >>> G.number_of_edges()
 67: (4)                 56
 68: (4)                 References
 69: (4)                 ----------
 70: (4)                 .. [1] Herzberg, A. M., & Murty, M. R. (2007). Sudoku squares and
chromatic
 71: (7)                    polynomials. Notices of the AMS, 54(6), 708-717.
 72: (4)                 .. [2] Sander, Torsten (2009), "Sudoku graphs are integral",
 73: (7)                    Electronic Journal of Combinatorics, 16 (1): Note 25, 7pp, MR 2529816
 74: (4)                 .. [3] Wikipedia contributors. "Glossary of Sudoku." Wikipedia, The Free
 75: (7)                    Encyclopedia, 3 Dec. 2019. Web. 22 Dec. 2019.
 76: (4)                 """
 77: (4)                 if n < 0:
 78: (8)                     raise NetworkXError("The order must be greater than or equal to
zero.")
 79: (4)                 n2 = n * n
 80: (4)                 n3 = n2 * n
 81: (4)                 n4 = n3 * n
 82: (4)                 G = nx.empty_graph(n4)
 83: (4)                 if n < 2:
 84: (8)                     return G
 85: (4)                 for row_no in range(n2):
 86: (8)                     row_start = row_no * n2
 87: (8)                     for j in range(1, n2):
 88: (12)                        for i in range(j):
 89: (16)                            G.add_edge(row_start + i, row_start + j)
 90: (4)                 for col_no in range(n2):
 91: (8)                     for j in range(col_no, n4, n2):
 92: (12)                        for i in range(col_no, j, n2):
```

```
93: (16)                            G.add_edge(i, j)
94: (4)                 for band_no in range(n):
95: (8)                     for stack_no in range(n):
96: (12)                        box_start = n3 * band_no + n * stack_no
97: (12)                        for j in range(1, n2):
98: (16)                            for i in range(j):
99: (20)                                u = box_start + (i % n) + n2 * (i // n)
100: (20)                               v = box_start + (j % n) + n2 * (j // n)
101: (20)                               G.add_edge(u, v)
102: (4)                return G

----------------------------------------

File 453 - time_series.py:

1: (0)              """
2: (0)              Time Series Graphs
3: (0)              """
4: (0)              import itertools
5: (0)              import networkx as nx
6: (0)              __all__ = ["visibility_graph"]
7: (0)              @nx._dispatchable(graphs=None, returns_graph=True)
8: (0)              def visibility_graph(series):
9: (4)                  """
10: (4)                 Return a Visibility Graph of an input Time Series.
11: (4)                 A visibility graph converts a time series into a graph. The constructed
graph
12: (4)                 uses integer nodes to indicate which event in the series the node
represents.
13: (4)                 Edges are formed as follows: consider a bar plot of the series and view
that
14: (4)                 as a side view of a landscape with a node at the top of each bar. An edge
15: (4)                 means that the nodes can be connected by a straight "line-of-sight"
without
16: (4)                 being obscured by any bars between the nodes.
17: (4)                 The resulting graph inherits several properties of the series in its
structure.
18: (4)                 Thereby, periodic series convert into regular graphs, random series
convert
19: (4)                 into random graphs, and fractal series convert into scale-free networks
[1]_.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 series : Sequence[Number]
23: (7)                    A Time Series sequence (iterable and sliceable) of numeric values
24: (7)                    representing times.
25: (4)                 Returns
26: (4)                 -------
27: (4)                 NetworkX Graph
28: (8)                     The Visibility Graph of the input series
29: (4)                 Examples
30: (4)                 --------
31: (4)                 >>> series_list = [range(10), [2, 1, 3, 2, 1, 3, 2, 1, 3, 2, 1, 3]]
32: (4)                 >>> for s in series_list:
33: (4)                 ...     g = nx.visibility_graph(s)
34: (4)                 ...     print(g)
35: (4)                 Graph with 10 nodes and 9 edges
36: (4)                 Graph with 12 nodes and 18 edges
37: (4)                 References
38: (4)                 ----------
39: (4)                 .. [1] Lacasa, Lucas, Bartolo Luque, Fernando Ballesteros, Jordi Luque,
and Juan Carlos Nuno.
40: (11)                       "From time series to complex networks: The visibility graph."
Proceedings of the
41: (11)                       National Academy of Sciences 105, no. 13 (2008): 4972-4975.
42: (11)                       https://www.pnas.org/doi/10.1073/pnas.0709247105
43: (4)                 """
44: (4)                 G = nx.path_graph(len(series))
45: (4)                 nx.set_node_attributes(G, dict(enumerate(series)), "value")
```

```
46: (4)                 for (n1, t1), (n2, t2) in itertools.combinations(enumerate(series), 2):
47: (8)                     slope = (t2 - t1) / (n2 - n1)
48: (8)                     offset = t2 - slope * n2
49: (8)                     obstructed = any(
50: (12)                        t >= slope * n + offset
51: (12)                        for n, t in enumerate(series[n1 + 1 : n2], start=n1 + 1)
52: (8)                     )
53: (8)                     if not obstructed:
54: (12)                        G.add_edge(n1, n2)
55: (4)                 return G

----------------------------------------

File 454 - trees.py:

1: (0)              """Functions for generating trees.
2: (0)              The functions sampling trees at random in this module come
3: (0)              in two variants: labeled and unlabeled. The labeled variants
4: (0)              sample from every possible tree with the given number of nodes
5: (0)              uniformly at random. The unlabeled variants sample from every
6: (0)              possible *isomorphism class* of trees with the given number
7: (0)              of nodes uniformly at random.
8: (0)              To understand the difference, consider the following example.
9: (0)              There are two isomorphism classes of trees with four nodes.
10: (0)             One is that of the path graph, the other is that of the
11: (0)             star graph. The unlabeled variant will return a line graph or
12: (0)             a star graph with probability 1/2.
13: (0)             The labeled variant will return the line graph
14: (0)             with probability 3/4 and the star graph with probability 1/4,
15: (0)             because there are more labeled variants of the line graph
16: (0)             than of the star graph. More precisely, the line graph has
17: (0)             an automorphism group of order 2, whereas the star graph has
18: (0)             an automorphism group of order 6, so the line graph has three
19: (0)             times as many labeled variants as the star graph, and thus
20: (0)             three more chances to be drawn.
21: (0)             Additionally, some functions in this module can sample rooted
22: (0)             trees and forests uniformly at random. A rooted tree is a tree
23: (0)             with a designated root node. A rooted forest is a disjoint union
24: (0)             of rooted trees.
25: (0)             """
26: (0)             import warnings
27: (0)             from collections import Counter, defaultdict
28: (0)             from math import comb, factorial
29: (0)             import networkx as nx
30: (0)             from networkx.utils import py_random_state
31: (0)             __all__ = [
32: (4)                 "prefix_tree",
33: (4)                 "prefix_tree_recursive",
34: (4)                 "random_tree",
35: (4)                 "random_labeled_tree",
36: (4)                 "random_labeled_rooted_tree",
37: (4)                 "random_labeled_rooted_forest",
38: (4)                 "random_unlabeled_tree",
39: (4)                 "random_unlabeled_rooted_tree",
40: (4)                 "random_unlabeled_rooted_forest",
41: (0)             ]
42: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
43: (0)             def prefix_tree(paths):
44: (4)                 """Creates a directed prefix tree from a list of paths.
45: (4)                 Usually the paths are described as strings or lists of integers.
46: (4)                 A "prefix tree" represents the prefix structure of the strings.
47: (4)                 Each node represents a prefix of some string. The root represents
48: (4)                 the empty prefix with children for the single letter prefixes which
49: (4)                 in turn have children for each double letter prefix starting with
50: (4)                 the single letter corresponding to the parent node, and so on.
51: (4)                 More generally the prefixes do not need to be strings. A prefix refers
52: (4)                 to the start of a sequence. The root has children for each one element
53: (4)                 prefix and they have children for each two element prefix that starts
54: (4)                 with the one element sequence of the parent, and so on.
```

```
    Note that this implementation uses integer nodes with an attribute.
    Each node has an attribute "source" whose value is the original element
    of the path to which this node corresponds. For example, suppose `paths`
    consists of one path: "can". Then the nodes `[1, 2, 3]` which represent
    this path have "source" values "c", "a" and "n".
    All the descendants of a node have a common prefix in the sequence/path
    associated with that node. From the returned tree, the prefix for each
    node can be constructed by traversing the tree up to the root and
    accumulating the "source" values along the way.
    The root node is always `0` and has "source" attribute `None`.
    The root is the only node with in-degree zero.
    The nil node is always `-1` and has "source" attribute `"NIL"`.
    The nil node is the only node with out-degree zero.
    Parameters
    ----------
    paths: iterable of paths
        An iterable of paths which are themselves sequences.
        Matching prefixes among these sequences are identified with
        nodes of the prefix tree. One leaf of the tree is associated
        with each path. (Identical paths are associated with the same
        leaf of the tree.)
    Returns
    -------
    tree: DiGraph
        A directed graph representing an arborescence consisting of the
        prefix tree generated by `paths`. Nodes are directed "downward",
        from parent to child. A special "synthetic" root node is added
        to be the parent of the first node in each path. A special
        "synthetic" leaf node, the "nil" node `-1`, is added to be the child
        of all nodes representing the last element in a path. (The
        addition of this nil node technically makes this not an
        arborescence but a directed acyclic graph; removing the nil node
        makes it an arborescence.)
    Notes
    -----
    The prefix tree is also known as a *trie*.
    Examples
    --------
    Create a prefix tree from a list of strings with common prefixes::
        >>> paths = ["ab", "abs", "ad"]
        >>> T = nx.prefix_tree(paths)
        >>> list(T.edges)
        [(0, 1), (1, 2), (1, 4), (2, -1), (2, 3), (3, -1), (4, -1)]
    The leaf nodes can be obtained as predecessors of the nil node::
        >>> root, NIL = 0, -1
        >>> list(T.predecessors(NIL))
        [2, 3, 4]
    To recover the original paths that generated the prefix tree,
    traverse up the tree from the node `-1` to the node `0`::
        >>> recovered = []
        >>> for v in T.predecessors(NIL):
        ...     prefix = ""
        ...     while v != root:
        ...         prefix = str(T.nodes[v]["source"]) + prefix
        ...         v = next(T.predecessors(v))  # only one predecessor
        ...     recovered.append(prefix)
        >>> sorted(recovered)
        ['ab', 'abs', 'ad']
    """
    def get_children(parent, paths):
        children = defaultdict(list)
        for path in paths:
            if not path:
                tree.add_edge(parent, NIL)
                continue
            child, *rest = path
            children[child].append(rest)
        return children
    tree = nx.DiGraph()
```

```
(4)                 root = 0
(4)                 tree.add_node(root, source=None)
(4)                 NIL = -1
(4)                 tree.add_node(NIL, source="NIL")
(4)                 children = get_children(root, paths)
(4)                 stack = [(root, iter(children.items()))]
(4)                 while stack:
(8)                     parent, remaining_children = stack[-1]
(8)                     try:
(12)                        child, remaining_paths = next(remaining_children)
(8)                     except StopIteration:
(12)                        stack.pop()
(12)                        continue
(8)                     new_name = len(tree) - 1
(8)                     tree.add_node(new_name, source=child)
(8)                     tree.add_edge(parent, new_name)
(8)                     children = get_children(new_name, remaining_paths)
(8)                     stack.append((new_name, iter(children.items())))
(4)                 return tree
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def prefix_tree_recursive(paths):
(4)                 """Recursively creates a directed prefix tree from a list of paths.
(4)                 The original recursive version of prefix_tree for comparison. It is
(4)                 the same algorithm but the recursion is unrolled onto a stack.
(4)                 Usually the paths are described as strings or lists of integers.
(4)                 A "prefix tree" represents the prefix structure of the strings.
(4)                 Each node represents a prefix of some string. The root represents
(4)                 the empty prefix with children for the single letter prefixes which
(4)                 in turn have children for each double letter prefix starting with
(4)                 the single letter corresponding to the parent node, and so on.
(4)                 More generally the prefixes do not need to be strings. A prefix refers
(4)                 to the start of a sequence. The root has children for each one element
(4)                 prefix and they have children for each two element prefix that starts
(4)                 with the one element sequence of the parent, and so on.
(4)                 Note that this implementation uses integer nodes with an attribute.
(4)                 Each node has an attribute "source" whose value is the original element
(4)                 of the path to which this node corresponds. For example, suppose `paths`
(4)                 consists of one path: "can". Then the nodes `[1, 2, 3]` which represent
(4)                 this path have "source" values "c", "a" and "n".
(4)                 All the descendants of a node have a common prefix in the sequence/path
(4)                 associated with that node. From the returned tree, ehe prefix for each
(4)                 node can be constructed by traversing the tree up to the root and
(4)                 accumulating the "source" values along the way.
(4)                 The root node is always `0` and has "source" attribute `None`.
(4)                 The root is the only node with in-degree zero.
(4)                 The nil node is always `-1` and has "source" attribute `"NIL"`.
(4)                 The nil node is the only node with out-degree zero.
(4)                 Parameters
(4)                 ----------
(4)                 paths: iterable of paths
(8)                     An iterable of paths which are themselves sequences.
(8)                     Matching prefixes among these sequences are identified with
(8)                     nodes of the prefix tree. One leaf of the tree is associated
(8)                     with each path. (Identical paths are associated with the same
(8)                     leaf of the tree.)
(4)                 Returns
(4)                 -------
(4)                 tree: DiGraph
(8)                     A directed graph representing an arborescence consisting of the
(8)                     prefix tree generated by `paths`. Nodes are directed "downward",
(8)                     from parent to child. A special "synthetic" root node is added
(8)                     to be the parent of the first node in each path. A special
(8)                     "synthetic" leaf node, the "nil" node `-1`, is added to be the child
(8)                     of all nodes representing the last element in a path. (The
(8)                     addition of this nil node technically makes this not an
(8)                     arborescence but a directed acyclic graph; removing the nil node
(8)                     makes it an arborescence.)
(4)                 Notes
(4)                 -----
```

```
    The prefix tree is also known as a *trie*.
    Examples
    --------
    Create a prefix tree from a list of strings with common prefixes::
        >>> paths = ["ab", "abs", "ad"]
        >>> T = nx.prefix_tree(paths)
        >>> list(T.edges)
        [(0, 1), (1, 2), (1, 4), (2, -1), (2, 3), (3, -1), (4, -1)]
    The leaf nodes can be obtained as predecessors of the nil node.
        >>> root, NIL = 0, -1
        >>> list(T.predecessors(NIL))
        [2, 3, 4]
    To recover the original paths that generated the prefix tree,
    traverse up the tree from the node `-1` to the node `0`::
        >>> recovered = []
        >>> for v in T.predecessors(NIL):
        ...     prefix = ""
        ...     while v != root:
        ...         prefix = str(T.nodes[v]["source"]) + prefix
        ...         v = next(T.predecessors(v))  # only one predecessor
        ...     recovered.append(prefix)
        >>> sorted(recovered)
        ['ab', 'abs', 'ad']
    """
    def _helper(paths, root, tree):
        """Recursively create a trie from the given list of paths.
        `paths` is a list of paths, each of which is itself a list of
        nodes, relative to the given `root` (but not including it). This
        list of paths will be interpreted as a tree-like structure, in
        which two paths that share a prefix represent two branches of
        the tree with the same initial segment.
        `root` is the parent of the node at index 0 in each path.
        `tree` is the "accumulator", the :class:`networkx.DiGraph`
        representing the branching to which the new nodes and edges will
        be added.
        """
        children = defaultdict(list)
        for path in paths:
            if not path:
                tree.add_edge(root, NIL)
                continue
            child, *rest = path
            children[child].append(rest)
        for child, remaining_paths in children.items():
            new_name = len(tree) - 1
            tree.add_node(new_name, source=child)
            tree.add_edge(root, new_name)
            _helper(remaining_paths, new_name, tree)
    tree = nx.DiGraph()
    root = 0
    tree.add_node(root, source=None)
    NIL = -1
    tree.add_node(NIL, source="NIL")
    _helper(paths, root, tree)
    return tree
@py_random_state(1)
@nx._dispatchable(graphs=None, returns_graph=True)
def random_tree(n, seed=None, create_using=None):
    """Returns a uniformly random tree on `n` nodes.
    .. deprecated:: 3.2
       ``random_tree`` is deprecated and will be removed in NX v3.4
       Use ``random_labeled_tree`` instead.
    Parameters
    ----------
    n : int
        A positive integer representing the number of nodes in the tree.
    seed : integer, random_state, or None (default)
        Indicator of random number generation state.
        See :ref:`Randomness<randomness>`.
```

```
    create_using : NetworkX graph constructor, optional (default=nx.Graph)
        Graph type to create. If graph instance, then cleared before
populated.

    Returns
    -------
    NetworkX graph
        A tree, given as an undirected graph, whose nodes are numbers in
        the set {0, …, *n* - 1}.
    Raises
    ------
    NetworkXPointlessConcept
        If `n` is zero (because the null graph is not a tree).
    Notes
    -----
    The current implementation of this function generates a uniformly
    random Prüfer sequence then converts that to a tree via the
    :func:`~networkx.from_prufer_sequence` function. Since there is a
    bijection between Prüfer sequences of length *n* - 2 and trees on
    *n* nodes, the tree is chosen uniformly at random from the set of
    all trees on *n* nodes.
    Examples
    --------
    >>> tree = nx.random_tree(n=10, seed=0)
    >>> nx.write_network_text(tree, sources=[0])
    ╙── 0
        ├── 3
        └── 4
            ├── 6
            │   ├── 1
            │   ├── 2
            │   └── 7
            │       └── 8
            │           └── 5
            └── 9
    >>> tree = nx.random_tree(n=10, seed=0, create_using=nx.DiGraph)
    >>> nx.write_network_text(tree)
    ╙── 0
        ├─╼ 3
        └─╼ 4
            ├─╼ 6
            │   ├─╼ 1
            │   ├─╼ 2
            │   └─╼ 7
            │       └─╼ 8
            │           └─╼ 5
            └─╼ 9
    """
    warnings.warn(
        (
            "\n\nrandom_tree is deprecated and will be removed in NX v3.4\n"
            "Use random_labeled_tree instead."
        ),
        DeprecationWarning,
        stacklevel=2,
    )
    if n == 0:
        raise nx.NetworkXPointlessConcept("the null graph is not a tree")
    if n == 1:
        utree = nx.empty_graph(1, create_using)
    else:
        sequence = [seed.choice(range(n)) for i in range(n - 2)]
        utree = nx.from_prufer_sequence(sequence)
    if create_using is None:
        tree = utree
    else:
        tree = nx.empty_graph(0, create_using)
        if tree.is_directed():
            edges = nx.dfs_edges(utree, source=0)
        else:
```

```
(12)                        edges = utree.edges
(8)                     tree.add_nodes_from(utree.nodes)
(8)                     tree.add_edges_from(edges)
(4)                 return tree
(0)             @py_random_state("seed")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_labeled_tree(n, *, seed=None):
(4)                 """Returns a labeled tree on `n` nodes chosen uniformly at random.
(4)                 Generating uniformly distributed random Prüfer sequences and
(4)                 converting them into the corresponding trees is a straightforward
(4)                 method of generating uniformly distributed random labeled trees.
(4)                 This function implements this method.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes, greater than zero.
(4)                 seed : random_state
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`
(4)                 Returns
(4)                 -------
(5)                  :class:`networkx.Graph`
(8)                     A `networkx.Graph` with nodes in the set {0, …, *n* - 1}.
(4)                 Raises
(4)                 ------
(4)                 NetworkXPointlessConcept
(8)                     If `n` is zero (because the null graph is not a tree).
(4)                 """
(4)                 if n == 0:
(8)                     raise nx.NetworkXPointlessConcept("the null graph is not a tree")
(4)                 if n == 1:
(8)                     return nx.empty_graph(1)
(4)                 return nx.from_prufer_sequence([seed.choice(range(n)) for i in range(n -
2)])
(0)             @py_random_state("seed")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_labeled_rooted_tree(n, *, seed=None):
(4)                 """Returns a labeled rooted tree with `n` nodes.
(4)                 The returned tree is chosen uniformly at random from all labeled rooted
trees.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 :class:`networkx.Graph`
(8)                     A `networkx.Graph` with integer nodes 0 <= node <= `n` - 1.
(8)                     The root of the tree is selected uniformly from the nodes.
(8)                     The "root" graph attribute identifies the root of the tree.
(4)                 Notes
(4)                 -----
(4)                 This function returns the result of :func:`random_labeled_tree`
(4)                 with a randomly selected root.
(4)                 Raises
(4)                 ------
(4)                 NetworkXPointlessConcept
(8)                     If `n` is zero (because the null graph is not a tree).
(4)                 """
(4)                 t = random_labeled_tree(n, seed=seed)
(4)                 t.graph["root"] = seed.randint(0, n - 1)
(4)                 return t
(0)             @py_random_state("seed")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_labeled_rooted_forest(n, *, seed=None):
(4)                 """Returns a labeled rooted forest with `n` nodes.
```

```
(4)                 The returned forest is chosen uniformly at random using a
(4)                 generalization of Prüfer sequences [1]_ in the form described in [2]_.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes.
(4)                 seed : random_state
(7)                    See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 :class:`networkx.Graph`
(8)                     A `networkx.Graph` with integer nodes 0 <= node <= `n` - 1.
(8)                     The "roots" graph attribute is a set of integers containing the roots.
(4)                 References
(4)                 ----------
(4)                 .. [1] Knuth, Donald E. "Another Enumeration of Trees."
(8)                     Canadian Journal of Mathematics, 20 (1968): 1077-1086.
(8)                     https://doi.org/10.4153/CJM-1968-104-8
(4)                 .. [2] Rubey, Martin. "Counting Spanning Trees". Diplomarbeit
(8)                     zur Erlangung des akademischen Grades Magister der
(8)                     Naturwissenschaften an der Formal- und Naturwissenschaftlichen
(8)                     Fakultät der Universität Wien. Wien, May 2000.
(4)                 """
(4)                 def _select_k(n, seed):
(8)                     r = seed.randint(0, (n + 1) ** (n - 1) - 1)
(8)                     cum_sum = 0
(8)                     for k in range(1, n):
(12)                        cum_sum += (factorial(n - 1) * n ** (n - k)) // (
(16)                            factorial(k - 1) * factorial(n - k)
(12)                        )
(12)                        if r < cum_sum:
(16)                            return k
(8)                     return n
(4)                 F = nx.empty_graph(n)
(4)                 if n == 0:
(8)                     F.graph["roots"] = {}
(8)                     return F
(4)                 k = _select_k(n, seed)
(4)                 if k == n:
(8)                     F.graph["roots"] = set(range(n))
(8)                     return F  # Nothing to do
(4)                 roots = seed.sample(range(n), k)
(4)                 p = set(range(n)).difference(roots)
(4)                 N = [seed.randint(0, n - 1) for i in range(n - k - 1)]
(4)                 degree = Counter([x for x in N if x in p])
(4)                 iterator = iter(x for x in p if degree[x] == 0)
(4)                 u = last = next(iterator)
(4)                 for v in N:
(8)                     F.add_edge(u, v)
(8)                     degree[v] -= 1
(8)                     if v < last and degree[v] == 0:
(12)                        u = v
(8)                     else:
(12)                        last = u = next(iterator)
(4)                 F.add_edge(u, roots[0])
(4)                 F.graph["roots"] = set(roots)
(4)                 return F
(0)             def _to_nx(edges, n_nodes, root=None, roots=None):
(4)                 """
(4)                 Converts the (edges, n_nodes) input to a :class:`networkx.Graph`.
(4)                 The (edges, n_nodes) input is a list of even length, where each pair
(4)                 of consecutive integers represents an edge, and an integer `n_nodes`.
(4)                 Integers in the list are elements of `range(n_nodes)`.
(4)                 Parameters
(4)                 ----------
(4)                 edges : list of ints
(8)                     The flattened list of edges of the graph.
(4)                 n_nodes : int
(8)                     The number of nodes of the graph.
```

```
    root: int (default=None)
        If not None, the "root" attribute of the graph will be set to this value.
    roots: collection of ints (default=None)
        If not None, he "roots" attribute of the graph will be set to this value.
    Returns
    -------
    :class:`networkx.Graph`
        The graph with `n_nodes` nodes and edges given by `edges`.
    """
    G = nx.empty_graph(n_nodes)
    G.add_edges_from(edges)
    if root is not None:
        G.graph["root"] = root
    if roots is not None:
        G.graph["roots"] = roots
    return G
def _num_rooted_trees(n, cache_trees):
    """Returns the number of unlabeled rooted trees with `n` nodes.
    See also https://oeis.org/A000081.
    Parameters
    ----------
    n : int
        The number of nodes
    cache_trees : list of ints
        The $i$-th element is the number of unlabeled rooted trees with $i$ nodes,
        which is used as a cache (and is extended to length $n+1$ if needed)
    Returns
    -------
    int
        The number of unlabeled rooted trees with `n` nodes.
    """
    for n_i in range(len(cache_trees), n + 1):
        cache_trees.append(
            sum(
                [
                    d * cache_trees[n_i - j * d] * cache_trees[d]
                    for d in range(1, n_i)
                    for j in range(1, (n_i - 1) // d + 1)
                ]
            )
            // (n_i - 1)
        )
    return cache_trees[n]
def _select_jd_trees(n, cache_trees, seed):
    """Returns a pair $(j,d)$ with a specific probability
    Given $n$, returns a pair of positive integers $(j,d)$ with the probability
    specified in formula (5) of Chapter 29 of [1]_.
    Parameters
    ----------
    n : int
        The number of nodes
    cache_trees : list of ints
        Cache for :func:`_num_rooted_trees`.
    seed : random_state
       See :ref:`Randomness<randomness>`.
    Returns
    -------
    (int, int)
        A pair of positive integers $(j,d)$ satisfying formula (5) of
        Chapter 29 of [1]_.
    References
    ----------
    .. [1] Nijenhuis, Albert, and Wilf, Herbert S.
        "Combinatorial algorithms: for computers and calculators."
        Academic Press, 1978.
```

```
(8)                 https://doi.org/10.1016/C2013-0-11243-3
(4)             """
(4)             p = seed.randint(0, _num_rooted_trees(n, cache_trees) * (n - 1) - 1)
(4)             cumsum = 0
(4)             for d in range(n - 1, 0, -1):
(8)                 for j in range(1, (n - 1) // d + 1):
(12)                    cumsum += (
(16)                        d
(16)                        * _num_rooted_trees(n - j * d, cache_trees)
(16)                        * _num_rooted_trees(d, cache_trees)
(12)                    )
(12)                    if p < cumsum:
(16)                        return (j, d)
(0)         def _random_unlabeled_rooted_tree(n, cache_trees, seed):
(4)             """Returns an unlabeled rooted tree with `n` nodes.
(4)             Returns an unlabeled rooted tree with `n` nodes chosen uniformly
(4)             at random using the "RANRUT" algorithm from [1]_.
(4)             The tree is returned in the form: (list_of_edges, number_of_nodes)
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 The number of nodes, greater than zero.
(4)             cache_trees : list ints
(8)                 Cache for :func:`_num_rooted_trees`.
(4)             seed : random_state
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             (list_of_edges, number_of_nodes) : list, int
(8)                 A random unlabeled rooted tree with `n` nodes as a 2-tuple
(8)                 ``(list_of_edges, number_of_nodes)``.
(8)                 The root is node 0.
(4)             References
(4)             ----------
(4)             .. [1] Nijenhuis, Albert, and Wilf, Herbert S.
(8)                 "Combinatorial algorithms: for computers and calculators."
(8)                 Academic Press, 1978.
(8)                 https://doi.org/10.1016/C2013-0-11243-3
(4)             """
(4)             if n == 1:
(8)                 edges, n_nodes = [], 1
(8)                 return edges, n_nodes
(4)             if n == 2:
(8)                 edges, n_nodes = [(0, 1)], 2
(8)                 return edges, n_nodes
(4)             j, d = _select_jd_trees(n, cache_trees, seed)
(4)             t1, t1_nodes = _random_unlabeled_rooted_tree(n - j * d, cache_trees, seed)
(4)             t2, t2_nodes = _random_unlabeled_rooted_tree(d, cache_trees, seed)
(4)             t12 = [(0, t2_nodes * i + t1_nodes) for i in range(j)]
(4)             t1.extend(t12)
(4)             for _ in range(j):
(8)                 t1.extend((n1 + t1_nodes, n2 + t1_nodes) for n1, n2 in t2)
(8)                 t1_nodes += t2_nodes
(4)             return t1, t1_nodes
(0)         @py_random_state("seed")
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def random_unlabeled_rooted_tree(n, *, number_of_trees=None, seed=None):
(4)             """Returns a number of unlabeled rooted trees uniformly at random
(4)             Returns one or more (depending on `number_of_trees`)
(4)             unlabeled rooted trees with `n` nodes drawn uniformly
(4)             at random.
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 The number of nodes
(4)             number_of_trees : int or None (default)
(8)                 If not None, this number of trees is generated and returned.
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
```

```
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             :class:`networkx.Graph` or list of :class:`networkx.Graph`
(8)                 A single `networkx.Graph` (or a list thereof, if `number_of_trees`
(8)                 is specified) with nodes in the set {0, …, *n* - 1}.
(8)                 The "root" graph attribute identifies the root of the tree.
(4)             Notes
(4)             -----
(4)             The trees are generated using the "RANRUT" algorithm from [1]_.
(4)             The algorithm needs to compute some counting functions
(4)             that are relatively expensive: in case several trees are needed,
(4)             it is advisable to use the `number_of_trees` optional argument
(4)             to reuse the counting functions.
(4)             Raises
(4)             ------
(4)             NetworkXPointlessConcept
(8)                 If `n` is zero (because the null graph is not a tree).
(4)             References
(4)             ----------
(4)             .. [1] Nijenhuis, Albert, and Wilf, Herbert S.
(8)                 "Combinatorial algorithms: for computers and calculators."
(8)                 Academic Press, 1978.
(8)                 https://doi.org/10.1016/C2013-0-11243-3
(4)             """
(4)             if n == 0:
(8)                 raise nx.NetworkXPointlessConcept("the null graph is not a tree")
(4)             cache_trees = [0, 1]  # initial cache of number of rooted trees
(4)             if number_of_trees is None:
(8)                 return _to_nx(*_random_unlabeled_rooted_tree(n, cache_trees, seed),
root=0)
(4)             return [
(8)                 _to_nx(*_random_unlabeled_rooted_tree(n, cache_trees, seed), root=0)
(8)                 for i in range(number_of_trees)
(4)             ]
(0)         def _num_rooted_forests(n, q, cache_forests):
(4)             """Returns the number of unlabeled rooted forests with `n` nodes, and with
(4)             no more than `q` nodes per tree. A recursive formula for this is (2) in
(4)             [1]_. This function is implemented using dynamic programming instead of
(4)             recursion.
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 The number of nodes.
(4)             q : int
(8)                 The maximum number of nodes for each tree of the forest.
(4)             cache_forests : list of ints
(8)                 The $i$-th element is the number of unlabeled rooted forests with
(8)                 $i$ nodes, and with no more than `q` nodes per tree; this is used
(8)                 as a cache (and is extended to length `n` + 1 if needed).
(4)             Returns
(4)             -------
(4)             int
(8)                 The number of unlabeled rooted forests with `n` nodes with no more
than
(8)                 `q` nodes per tree.
(4)             References
(4)             ----------
(4)             .. [1] Wilf, Herbert S. "The uniform selection of free trees."
(8)                 Journal of Algorithms 2.2 (1981): 204-207.
(8)                 https://doi.org/10.1016/0196-6774(81)90021-3
(4)             """
(4)             for n_i in range(len(cache_forests), n + 1):
(8)                 q_i = min(n_i, q)
(8)                 cache_forests.append(
(12)                    sum(
(16)                        [
(20)                            d * cache_forests[n_i - j * d] * cache_forests[d - 1]
(20)                            for d in range(1, q_i + 1)
```

```
667: (20)                                for j in range(1, n_i // d + 1)
668: (16)                            ]
669: (12)                        )
670: (12)                        // n_i
671: (8)                     )
672: (4)                 return cache_forests[n]
673: (0)             def _select_jd_forests(n, q, cache_forests, seed):
674: (4)                 """Given `n` and `q`, returns a pair of positive integers $(j,d)$
675: (4)                 such that $j\\leq d$, with probability satisfying (F1) of [1]_.
676: (4)                 Parameters
677: (4)                 ----------
678: (4)                 n : int
679: (8)                     The number of nodes.
680: (4)                 q : int
681: (8)                     The maximum number of nodes for each tree of the forest.
682: (4)                 cache_forests : list of ints
683: (8)                     Cache for :func:`_num_rooted_forests`.
684: (4)                 seed : random_state
685: (8)                     See :ref:`Randomness<randomness>`.
686: (4)                 Returns
687: (4)                 -------
688: (4)                 (int, int)
689: (8)                     A pair of positive integers $(j,d)$
690: (4)                 References
691: (4)                 ----------
692: (4)                 .. [1] Wilf, Herbert S. "The uniform selection of free trees."
693: (8)                     Journal of Algorithms 2.2 (1981): 204-207.
694: (8)                     https://doi.org/10.1016/0196-6774(81)90021-3
695: (4)                 """
696: (4)                 p = seed.randint(0, _num_rooted_forests(n, q, cache_forests) * n - 1)
697: (4)                 cumsum = 0
698: (4)                 for d in range(q, 0, -1):
699: (8)                     for j in range(1, n // d + 1):
700: (12)                        cumsum += (
701: (16)                            d
702: (16)                            * _num_rooted_forests(n - j * d, q, cache_forests)
703: (16)                            * _num_rooted_forests(d - 1, q, cache_forests)
704: (12)                        )
705: (12)                        if p < cumsum:
706: (16)                            return (j, d)
707: (0)             def _random_unlabeled_rooted_forest(n, q, cache_trees, cache_forests, seed):
708: (4)                 """Returns an unlabeled rooted forest with `n` nodes, and with no more
709: (4)                 than `q` nodes per tree, drawn uniformly at random. It is an
implementation
710: (4)                 of the algorithm "Forest" of [1]_.
711: (4)                 Parameters
712: (4)                 ----------
713: (4)                 n : int
714: (8)                     The number of nodes.
715: (4)                 q : int
716: (8)                     The maximum number of nodes per tree.
717: (4)                 cache_trees :
718: (8)                     Cache for :func:`_num_rooted_trees`.
719: (4)                 cache_forests :
720: (8)                     Cache for :func:`_num_rooted_forests`.
721: (4)                 seed : random_state
722: (7)                    See :ref:`Randomness<randomness>`.
723: (4)                 Returns
724: (4)                 -------
725: (4)                 (edges, n, r) : (list, int, list)
726: (8)                     The forest (edges, n) and a list r of root nodes.
727: (4)                 References
728: (4)                 ----------
729: (4)                 .. [1] Wilf, Herbert S. "The uniform selection of free trees."
730: (8)                     Journal of Algorithms 2.2 (1981): 204-207.
731: (8)                     https://doi.org/10.1016/0196-6774(81)90021-3
732: (4)                 """
733: (4)                 if n == 0:
734: (8)                     return ([], 0, [])
```

```
(4)                 j, d = _select_jd_forests(n, q, cache_forests, seed)
(4)                 t1, t1_nodes, r1 = _random_unlabeled_rooted_forest(
(8)                     n - j * d, q, cache_trees, cache_forests, seed
(4)                 )
(4)                 t2, t2_nodes = _random_unlabeled_rooted_tree(d, cache_trees, seed)
(4)                 for _ in range(j):
(8)                     r1.append(t1_nodes)
(8)                     t1.extend((n1 + t1_nodes, n2 + t1_nodes) for n1, n2 in t2)
(8)                     t1_nodes += t2_nodes
(4)                 return t1, t1_nodes, r1
(0)             @py_random_state("seed")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def random_unlabeled_rooted_forest(n, *, q=None, number_of_forests=None,
seed=None):
(4)                 """Returns a forest or list of forests selected at random.
(4)                 Returns one or more (depending on `number_of_forests`)
(4)                 unlabeled rooted forests with `n` nodes, and with no more than
(4)                 `q` nodes per tree, drawn uniformly at random.
(4)                 The "roots" graph attribute identifies the roots of the forest.
(4)                 Parameters
(4)                 ----------
(4)                 n : int
(8)                     The number of nodes
(4)                 q : int or None (default)
(8)                     The maximum number of nodes per tree.
(4)                 number_of_forests : int or None (default)
(8)                     If not None, this number of forests is generated and returned.
(4)                 seed : integer, random_state, or None (default)
(8)                     Indicator of random number generation state.
(8)                     See :ref:`Randomness<randomness>`.
(4)                 Returns
(4)                 -------
(4)                 :class:`networkx.Graph` or list of :class:`networkx.Graph`
(8)                     A single `networkx.Graph` (or a list thereof, if `number_of_forests`
(8)                     is specified) with nodes in the set {0, …, *n* - 1}.
(8)                     The "roots" graph attribute is a set containing the roots
(8)                     of the trees in the forest.
(4)                 Notes
(4)                 -----
(4)                 This function implements the algorithm "Forest" of [1]_.
(4)                 The algorithm needs to compute some counting functions
(4)                 that are relatively expensive: in case several trees are needed,
(4)                 it is advisable to use the `number_of_forests` optional argument
(4)                 to reuse the counting functions.
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If `n` is non-zero but `q` is zero.
(4)                 References
(4)                 ----------
(4)                 .. [1] Wilf, Herbert S. "The uniform selection of free trees."
(8)                     Journal of Algorithms 2.2 (1981): 204-207.
(8)                     https://doi.org/10.1016/0196-6774(81)90021-3
(4)                 """
(4)                 if q is None:
(8)                     q = n
(4)                 if q == 0 and n != 0:
(8)                     raise ValueError("q must be a positive integer if n is positive.")
(4)                 cache_trees = [0, 1]  # initial cache of number of rooted trees
(4)                 cache_forests = [1]  # initial cache of number of rooted forests
(4)                 if number_of_forests is None:
(8)                     g, nodes, rs = _random_unlabeled_rooted_forest(
(12)                        n, q, cache_trees, cache_forests, seed
(8)                     )
(8)                     return _to_nx(g, nodes, roots=set(rs))
(4)                 res = []
(4)                 for i in range(number_of_forests):
(8)                     g, nodes, rs = _random_unlabeled_rooted_forest(
(12)                        n, q, cache_trees, cache_forests, seed
```

```
803: (8)                    )
804: (8)                    res.append(_to_nx(g, nodes, roots=set(rs)))
805: (4)                return res
806: (0)            def _num_trees(n, cache_trees):
807: (4)                """Returns the number of unlabeled trees with `n` nodes.
808: (4)                See also https://oeis.org/A000055.
809: (4)                Parameters
810: (4)                ----------
811: (4)                n : int
812: (8)                    The number of nodes.
813: (4)                cache_trees : list of ints
814: (8)                    Cache for :func:`_num_rooted_trees`.
815: (4)                Returns
816: (4)                -------
817: (4)                int
818: (8)                    The number of unlabeled trees with `n` nodes.
819: (4)                """
820: (4)                r = _num_rooted_trees(n, cache_trees) - sum(
821: (8)                    [
822: (12)                       _num_rooted_trees(j, cache_trees) * _num_rooted_trees(n - j,
cache_trees)
823: (12)                       for j in range(1, n // 2 + 1)
824: (8)                    ]
825: (4)                )
826: (4)                if n % 2 == 0:
827: (8)                    r += comb(_num_rooted_trees(n // 2, cache_trees) + 1, 2)
828: (4)                return r
829: (0)            def _bicenter(n, cache, seed):
830: (4)                """Returns a bi-centroidal tree on `n` nodes drawn uniformly at random.
831: (4)                This function implements the algorithm Bicenter of [1]_.
832: (4)                Parameters
833: (4)                ----------
834: (4)                n : int
835: (8)                    The number of nodes (must be even).
836: (4)                cache : list of ints.
837: (8)                    Cache for :func:`_num_rooted_trees`.
838: (4)                seed : random_state
839: (8)                    See :ref:`Randomness<randomness>`
840: (4)                Returns
841: (4)                -------
842: (4)                (edges, n)
843: (8)                    The tree as a list of edges and number of nodes.
844: (4)                References
845: (4)                ----------
846: (4)                .. [1] Wilf, Herbert S. "The uniform selection of free trees."
847: (8)                    Journal of Algorithms 2.2 (1981): 204-207.
848: (8)                    https://doi.org/10.1016/0196-6774(81)90021-3
849: (4)                """
850: (4)                t, t_nodes = _random_unlabeled_rooted_tree(n // 2, cache, seed)
851: (4)                if seed.randint(0, _num_rooted_trees(n // 2, cache)) == 0:
852: (8)                    t2, t2_nodes = t, t_nodes
853: (4)                else:
854: (8)                    t2, t2_nodes = _random_unlabeled_rooted_tree(n // 2, cache, seed)
855: (4)                t.extend([(n1 + (n // 2), n2 + (n // 2)) for n1, n2 in t2])
856: (4)                t.append((0, n // 2))
857: (4)                return t, t_nodes + t2_nodes
858: (0)            def _random_unlabeled_tree(n, cache_trees, cache_forests, seed):
859: (4)                """Returns a tree on `n` nodes drawn uniformly at random.
860: (4)                It implements the Wilf's algorithm "Free" of [1]_.
861: (4)                Parameters
862: (4)                ----------
863: (4)                n : int
864: (8)                    The number of nodes, greater than zero.
865: (4)                cache_trees : list of ints
866: (8)                    Cache for :func:`_num_rooted_trees`.
867: (4)                cache_forests : list of ints
868: (8)                    Cache for :func:`_num_rooted_forests`.
869: (4)                seed : random_state
870: (8)                    Indicator of random number generation state.
```

```
(8)                 See :ref:`Randomness<randomness>`
(4)             Returns
(4)             -------
(4)             (edges, n)
(8)                 The tree as a list of edges and number of nodes.
(4)             References
(4)             ----------
(4)             .. [1] Wilf, Herbert S. "The uniform selection of free trees."
(8)                 Journal of Algorithms 2.2 (1981): 204-207.
(8)                 https://doi.org/10.1016/0196-6774(81)90021-3
(4)             """
(4)             if n % 2 == 1:
(8)                 p = 0
(4)             else:
(8)                 p = comb(_num_rooted_trees(n // 2, cache_trees) + 1, 2)
(4)             if seed.randint(0, _num_trees(n, cache_trees) - 1) < p:
(8)                 return _bicenter(n, cache_trees, seed)
(4)             else:
(8)                 f, n_f, r = _random_unlabeled_rooted_forest(
(12)                    n - 1, (n - 1) // 2, cache_trees, cache_forests, seed
(8)                 )
(8)                 for i in r:
(12)                    f.append((i, n_f))
(8)                 return f, n_f + 1
(0)         @py_random_state("seed")
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def random_unlabeled_tree(n, *, number_of_trees=None, seed=None):
(4)             """Returns a tree or list of trees chosen randomly.
(4)             Returns one or more (depending on `number_of_trees`)
(4)             unlabeled trees with `n` nodes drawn uniformly at random.
(4)             Parameters
(4)             ----------
(4)             n : int
(8)                 The number of nodes
(4)             number_of_trees : int or None (default)
(8)                 If not None, this number of trees is generated and returned.
(4)             seed : integer, random_state, or None (default)
(8)                 Indicator of random number generation state.
(8)                 See :ref:`Randomness<randomness>`.
(4)             Returns
(4)             -------
(4)             :class:`networkx.Graph` or list of :class:`networkx.Graph`
(8)                 A single `networkx.Graph` (or a list thereof, if
(8)                 `number_of_trees` is specified) with nodes in the set {0, …, *n* - 1}.
(4)             Raises
(4)             ------
(4)             NetworkXPointlessConcept
(8)                 If `n` is zero (because the null graph is not a tree).
(4)             Notes
(4)             -----
(4)             This function generates an unlabeled tree uniformly at random using
(4)             Wilf's algorithm "Free" of [1]_. The algorithm needs to
(4)             compute some counting functions that are relatively expensive:
(4)             in case several trees are needed, it is advisable to use the
(4)             `number_of_trees` optional argument to reuse the counting
(4)             functions.
(4)             References
(4)             ----------
(4)             .. [1] Wilf, Herbert S. "The uniform selection of free trees."
(8)                 Journal of Algorithms 2.2 (1981): 204-207.
(8)                 https://doi.org/10.1016/0196-6774(81)90021-3
(4)             """
(4)             if n == 0:
(8)                 raise nx.NetworkXPointlessConcept("the null graph is not a tree")
(4)             cache_trees = [0, 1]  # initial cache of number of rooted trees
(4)             cache_forests = [1]  # initial cache of number of rooted forests
(4)             if number_of_trees is None:
(8)                 return _to_nx(*_random_unlabeled_tree(n, cache_trees, cache_forests,
seed))
```

```
939: (4)                else:
940: (8)                    return [
941: (12)                       _to_nx(*_random_unlabeled_tree(n, cache_trees, cache_forests,
seed))
942: (12)                       for i in range(number_of_trees)
943: (8)                    ]
```

File 455 - triads.py:

```
1: (0)              """Functions that generate the triad graphs, that is, the possible
2: (0)              digraphs on three nodes.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.classes import DiGraph
6: (0)              __all__ = ["triad_graph"]
7: (0)              TRIAD_EDGES = {
8: (4)                  "003": [],
9: (4)                  "012": ["ab"],
10: (4)                 "102": ["ab", "ba"],
11: (4)                 "021D": ["ba", "bc"],
12: (4)                 "021U": ["ab", "cb"],
13: (4)                 "021C": ["ab", "bc"],
14: (4)                 "111D": ["ac", "ca", "bc"],
15: (4)                 "111U": ["ac", "ca", "cb"],
16: (4)                 "030T": ["ab", "cb", "ac"],
17: (4)                 "030C": ["ba", "cb", "ac"],
18: (4)                 "201": ["ab", "ba", "ac", "ca"],
19: (4)                 "120D": ["bc", "ba", "ac", "ca"],
20: (4)                 "120U": ["ab", "cb", "ac", "ca"],
21: (4)                 "120C": ["ab", "bc", "ac", "ca"],
22: (4)                 "210": ["ab", "bc", "cb", "ac", "ca"],
23: (4)                 "300": ["ab", "ba", "bc", "cb", "ac", "ca"],
24: (0)             }
25: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
26: (0)             def triad_graph(triad_name):
27: (4)                 """Returns the triad graph with the given name.
28: (4)                 Each string in the following tuple is a valid triad name::
29: (8)                     (
30: (12)                        "003",
31: (12)                        "012",
32: (12)                        "102",
33: (12)                        "021D",
34: (12)                        "021U",
35: (12)                        "021C",
36: (12)                        "111D",
37: (12)                        "111U",
38: (12)                        "030T",
39: (12)                        "030C",
40: (12)                        "201",
41: (12)                        "120D",
42: (12)                        "120U",
43: (12)                        "120C",
44: (12)                        "210",
45: (12)                        "300",
46: (8)                     )
47: (4)                 Each triad name corresponds to one of the possible valid digraph on
48: (4)                 three nodes.
49: (4)                 Parameters
50: (4)                 ----------
51: (4)                 triad_name : string
52: (8)                     The name of a triad, as described above.
53: (4)                 Returns
54: (4)                 -------
55: (4)                 :class:`~networkx.DiGraph`
56: (8)                     The digraph on three nodes with the given name. The nodes of the
57: (8)                     graph are the single-character strings 'a', 'b', and 'c'.
58: (4)                 Raises
```

```
59: (4)                 ------
60: (4)                 ValueError
61: (8)                     If `triad_name` is not the name of a triad.
62: (4)                 See also
63: (4)                 --------
64: (4)                 triadic_census
65: (4)                 """
66: (4)                 if triad_name not in TRIAD_EDGES:
67: (8)                     raise ValueError(
68: (12)                        f'unknown triad name "{triad_name}"; use one of the triad names'
69: (12)                        " in the TRIAD_NAMES constant"
70: (8)                     )
71: (4)                 G = DiGraph()
72: (4)                 G.add_nodes_from("abc")
73: (4)                 G.add_edges_from(TRIAD_EDGES[triad_name])
74: (4)                 return G
```

File 456 - __init__.py:

```
1: (0)
```

File 457 - test_atlas.py:

```
1: (0)              from itertools import groupby
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx import graph_atlas, graph_atlas_g
5: (0)              from networkx.generators.atlas import NUM_GRAPHS
6: (0)              from networkx.utils import edges_equal, nodes_equal, pairwise
7: (0)              class TestAtlasGraph:
8: (4)                  """Unit tests for the :func:`~networkx.graph_atlas` function."""
9: (4)                  def test_index_too_small(self):
10: (8)                     with pytest.raises(ValueError):
11: (12)                        graph_atlas(-1)
12: (4)                 def test_index_too_large(self):
13: (8)                     with pytest.raises(ValueError):
14: (12)                        graph_atlas(NUM_GRAPHS)
15: (4)                 def test_graph(self):
16: (8)                     G = graph_atlas(6)
17: (8)                     assert nodes_equal(G.nodes(), range(3))
18: (8)                     assert edges_equal(G.edges(), [(0, 1), (0, 2)])
19: (0)             class TestAtlasGraphG:
20: (4)                 """Unit tests for the :func:`~networkx.graph_atlas_g` function."""
21: (4)                 @classmethod
22: (4)                 def setup_class(cls):
23: (8)                     cls.GAG = graph_atlas_g()
24: (4)                 def test_sizes(self):
25: (8)                     G = self.GAG[0]
26: (8)                     assert G.number_of_nodes() == 0
27: (8)                     assert G.number_of_edges() == 0
28: (8)                     G = self.GAG[7]
29: (8)                     assert G.number_of_nodes() == 3
30: (8)                     assert G.number_of_edges() == 3
31: (4)                 def test_names(self):
32: (8)                     for i, G in enumerate(self.GAG):
33: (12)                        assert int(G.name[1:]) == i
34: (4)                 def test_nondecreasing_nodes(self):
35: (8)                     for n1, n2 in pairwise(map(len, self.GAG)):
36: (12)                        assert n2 <= n1 + 1
37: (4)                 def test_nondecreasing_edges(self):
38: (8)                     for n, group in groupby(self.GAG, key=nx.number_of_nodes):
39: (12)                        for m1, m2 in pairwise(map(nx.number_of_edges, group)):
40: (16)                            assert m2 <= m1 + 1
41: (4)                 def test_nondecreasing_degree_sequence(self):
42: (8)                     exceptions = [("G55", "G56"), ("G1007", "G1008"), ("G1012", "G1013")]
```

```
43: (8)                     for n, group in groupby(self.GAG, key=nx.number_of_nodes):
44: (12)                        for m, group in groupby(group, key=nx.number_of_edges):
45: (16)                            for G1, G2 in pairwise(group):
46: (20)                                if (G1.name, G2.name) in exceptions:
47: (24)                                    continue
48: (20)                                d1 = sorted(d for v, d in G1.degree())
49: (20)                                d2 = sorted(d for v, d in G2.degree())
50: (20)                                assert d1 <= d2
```

File 458 - test_classic.py:

```
1: (0)              """
2: (0)              ====================
3: (0)              Generators - Classic
4: (0)              ====================
5: (0)              Unit tests for various classic graph generators in generators/classic.py
6: (0)              """
7: (0)              import itertools
8: (0)              import typing
9: (0)              import pytest
10: (0)             import networkx as nx
11: (0)             from networkx.algorithms.isomorphism.isomorph import graph_could_be_isomorphic
12: (0)             from networkx.utils import edges_equal, nodes_equal
13: (0)             is_isomorphic = graph_could_be_isomorphic
14: (0)             class TestGeneratorClassic:
15: (4)                 def test_balanced_tree(self):
16: (8)                     for r, h in [(2, 2), (3, 3), (6, 2)]:
17: (12)                        t = nx.balanced_tree(r, h)
18: (12)                        order = t.order()
19: (12)                        assert order == (r ** (h + 1) - 1) / (r - 1)
20: (12)                        assert nx.is_connected(t)
21: (12)                        assert t.size() == order - 1
22: (12)                        dh = nx.degree_histogram(t)
23: (12)                        assert dh[0] == 0  # no nodes of 0
24: (12)                        assert dh[1] == r**h  # nodes of degree 1 are leaves
25: (12)                        assert dh[r] == 1  # root is degree r
26: (12)                        assert dh[r + 1] == order - r**h - 1  # everyone else is degree
r+1
27: (12)                        assert len(dh) == r + 2
28: (4)                 def test_balanced_tree_star(self):
29: (8)                     t = nx.balanced_tree(r=2, h=1)
30: (8)                     assert is_isomorphic(t, nx.star_graph(2))
31: (8)                     t = nx.balanced_tree(r=5, h=1)
32: (8)                     assert is_isomorphic(t, nx.star_graph(5))
33: (8)                     t = nx.balanced_tree(r=10, h=1)
34: (8)                     assert is_isomorphic(t, nx.star_graph(10))
35: (4)                 def test_balanced_tree_path(self):
36: (8)                     """Tests that the balanced tree with branching factor one is the
37: (8)                     path graph.
38: (8)                     """
39: (8)                     T = nx.balanced_tree(1, 4)
40: (8)                     P = nx.path_graph(5)
41: (8)                     assert is_isomorphic(T, P)
42: (4)                 def test_full_rary_tree(self):
43: (8)                     r = 2
44: (8)                     n = 9
45: (8)                     t = nx.full_rary_tree(r, n)
46: (8)                     assert t.order() == n
47: (8)                     assert nx.is_connected(t)
48: (8)                     dh = nx.degree_histogram(t)
49: (8)                     assert dh[0] == 0  # no nodes of 0
50: (8)                     assert dh[1] == 5  # nodes of degree 1 are leaves
51: (8)                     assert dh[r] == 1  # root is degree r
52: (8)                     assert dh[r + 1] == 9 - 5 - 1  # everyone else is degree r+1
53: (8)                     assert len(dh) == r + 2
54: (4)                 def test_full_rary_tree_balanced(self):
55: (8)                     t = nx.full_rary_tree(2, 15)
```

```
56: (8)                     th = nx.balanced_tree(2, 3)
57: (8)                     assert is_isomorphic(t, th)
58: (4)                 def test_full_rary_tree_path(self):
59: (8)                     t = nx.full_rary_tree(1, 10)
60: (8)                     assert is_isomorphic(t, nx.path_graph(10))
61: (4)                 def test_full_rary_tree_empty(self):
62: (8)                     t = nx.full_rary_tree(0, 10)
63: (8)                     assert is_isomorphic(t, nx.empty_graph(10))
64: (8)                     t = nx.full_rary_tree(3, 0)
65: (8)                     assert is_isomorphic(t, nx.empty_graph(0))
66: (4)                 def test_full_rary_tree_3_20(self):
67: (8)                     t = nx.full_rary_tree(3, 20)
68: (8)                     assert t.order() == 20
69: (4)                 def test_barbell_graph(self):
70: (8)                     m1 = 3
71: (8)                     m2 = 5
72: (8)                     b = nx.barbell_graph(m1, m2)
73: (8)                     assert nx.number_of_nodes(b) == 2 * m1 + m2
74: (8)                     assert nx.number_of_edges(b) == m1 * (m1 - 1) + m2 + 1
75: (8)                     m1 = 4
76: (8)                     m2 = 10
77: (8)                     b = nx.barbell_graph(m1, m2)
78: (8)                     assert nx.number_of_nodes(b) == 2 * m1 + m2
79: (8)                     assert nx.number_of_edges(b) == m1 * (m1 - 1) + m2 + 1
80: (8)                     m1 = 3
81: (8)                     m2 = 20
82: (8)                     b = nx.barbell_graph(m1, m2)
83: (8)                     assert nx.number_of_nodes(b) == 2 * m1 + m2
84: (8)                     assert nx.number_of_edges(b) == m1 * (m1 - 1) + m2 + 1
85: (8)                     m1 = 1
86: (8)                     m2 = 20
87: (8)                     pytest.raises(nx.NetworkXError, nx.barbell_graph, m1, m2)
88: (8)                     m1 = 5
89: (8)                     m2 = -2
90: (8)                     pytest.raises(nx.NetworkXError, nx.barbell_graph, m1, m2)
91: (8)                     m1 = 2
92: (8)                     m2 = 5
93: (8)                     b = nx.barbell_graph(m1, m2)
94: (8)                     assert is_isomorphic(b, nx.path_graph(m2 + 4))
95: (8)                     m1 = 2
96: (8)                     m2 = 10
97: (8)                     b = nx.barbell_graph(m1, m2)
98: (8)                     assert is_isomorphic(b, nx.path_graph(m2 + 4))
99: (8)                     m1 = 2
100: (8)                    m2 = 20
101: (8)                    b = nx.barbell_graph(m1, m2)
102: (8)                    assert is_isomorphic(b, nx.path_graph(m2 + 4))
103: (8)                    pytest.raises(
104: (12)                       nx.NetworkXError, nx.barbell_graph, m1, m2,
create_using=nx.DiGraph()
105: (8)                    )
106: (8)                    mb = nx.barbell_graph(m1, m2, create_using=nx.MultiGraph())
107: (8)                    assert edges_equal(mb.edges(), b.edges())
108: (4)                def test_binomial_tree(self):
109: (8)                    graphs = (None, nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph)
110: (8)                    for create_using in graphs:
111: (12)                       for n in range(4):
112: (16)                           b = nx.binomial_tree(n, create_using)
113: (16)                           assert nx.number_of_nodes(b) == 2**n
114: (16)                           assert nx.number_of_edges(b) == (2**n - 1)
115: (4)                def test_complete_graph(self):
116: (8)                    for m in [0, 1, 3, 5]:
117: (12)                       g = nx.complete_graph(m)
118: (12)                       assert nx.number_of_nodes(g) == m
119: (12)                       assert nx.number_of_edges(g) == m * (m - 1) // 2
120: (8)                    mg = nx.complete_graph(m, create_using=nx.MultiGraph)
121: (8)                    assert edges_equal(mg.edges(), g.edges())
122: (8)                    g = nx.complete_graph("abc")
123: (8)                    assert nodes_equal(g.nodes(), ["a", "b", "c"])
```

```
(8)                     assert g.size() == 3
(8)                     g = nx.complete_graph("abcb")
(8)                     assert nodes_equal(g.nodes(), ["a", "b", "c"])
(8)                     assert g.size() == 4
(8)                     g = nx.complete_graph("abcb", create_using=nx.MultiGraph)
(8)                     assert nodes_equal(g.nodes(), ["a", "b", "c"])
(8)                     assert g.size() == 6
(4)                 def test_complete_digraph(self):
(8)                     for m in [0, 1, 3, 5]:
(12)                        g = nx.complete_graph(m, create_using=nx.DiGraph)
(12)                        assert nx.number_of_nodes(g) == m
(12)                        assert nx.number_of_edges(g) == m * (m - 1)
(8)                     g = nx.complete_graph("abc", create_using=nx.DiGraph)
(8)                     assert len(g) == 3
(8)                     assert g.size() == 6
(8)                     assert g.is_directed()
(4)                 def test_circular_ladder_graph(self):
(8)                     G = nx.circular_ladder_graph(5)
(8)                     pytest.raises(
(12)                        nx.NetworkXError, nx.circular_ladder_graph, 5,
create_using=nx.DiGraph
(8)                     )
(8)                     mG = nx.circular_ladder_graph(5, create_using=nx.MultiGraph)
(8)                     assert edges_equal(mG.edges(), G.edges())
(4)                 def test_circulant_graph(self):
(8)                     Ci6_1 = nx.circulant_graph(6, [1])
(8)                     C6 = nx.cycle_graph(6)
(8)                     assert edges_equal(Ci6_1.edges(), C6.edges())
(8)                     Ci7 = nx.circulant_graph(7, [1, 2, 3])
(8)                     K7 = nx.complete_graph(7)
(8)                     assert edges_equal(Ci7.edges(), K7.edges())
(8)                     Ci6_1_3 = nx.circulant_graph(6, [1, 3])
(8)                     K3_3 = nx.complete_bipartite_graph(3, 3)
(8)                     assert is_isomorphic(Ci6_1_3, K3_3)
(4)                 def test_cycle_graph(self):
(8)                     G = nx.cycle_graph(4)
(8)                     assert edges_equal(G.edges(), [(0, 1), (0, 3), (1, 2), (2, 3)])
(8)                     mG = nx.cycle_graph(4, create_using=nx.MultiGraph)
(8)                     assert edges_equal(mG.edges(), [(0, 1), (0, 3), (1, 2), (2, 3)])
(8)                     G = nx.cycle_graph(4, create_using=nx.DiGraph)
(8)                     assert not G.has_edge(2, 1)
(8)                     assert G.has_edge(1, 2)
(8)                     assert G.is_directed()
(8)                     G = nx.cycle_graph("abc")
(8)                     assert len(G) == 3
(8)                     assert G.size() == 3
(8)                     G = nx.cycle_graph("abcb")
(8)                     assert len(G) == 3
(8)                     assert G.size() == 2
(8)                     g = nx.cycle_graph("abc", nx.DiGraph)
(8)                     assert len(g) == 3
(8)                     assert g.size() == 3
(8)                     assert g.is_directed()
(8)                     g = nx.cycle_graph("abcb", nx.DiGraph)
(8)                     assert len(g) == 3
(8)                     assert g.size() == 4
(4)                 def test_dorogovtsev_goltsev_mendes_graph(self):
(8)                     G = nx.dorogovtsev_goltsev_mendes_graph(0)
(8)                     assert edges_equal(G.edges(), [(0, 1)])
(8)                     assert nodes_equal(list(G), [0, 1])
(8)                     G = nx.dorogovtsev_goltsev_mendes_graph(1)
(8)                     assert edges_equal(G.edges(), [(0, 1), (0, 2), (1, 2)])
(8)                     assert nx.average_clustering(G) == 1.0
(8)                     assert sorted(nx.triangles(G).values()) == [1, 1, 1]
(8)                     G = nx.dorogovtsev_goltsev_mendes_graph(10)
(8)                     assert nx.number_of_nodes(G) == 29526
(8)                     assert nx.number_of_edges(G) == 59049
(8)                     assert G.degree(0) == 1024
(8)                     assert G.degree(1) == 1024
```

```
192: (8)                    assert G.degree(2) == 1024
193: (8)                    pytest.raises(
194: (12)                       nx.NetworkXError,
195: (12)                       nx.dorogovtsev_goltsev_mendes_graph,
196: (12)                       7,
197: (12)                       create_using=nx.DiGraph,
198: (8)                    )
199: (8)                    pytest.raises(
200: (12)                       nx.NetworkXError,
201: (12)                       nx.dorogovtsev_goltsev_mendes_graph,
202: (12)                       7,
203: (12)                       create_using=nx.MultiGraph,
204: (8)                    )
205: (4)                def test_create_using(self):
206: (8)                    G = nx.empty_graph()
207: (8)                    assert isinstance(G, nx.Graph)
208: (8)                    pytest.raises(TypeError, nx.empty_graph, create_using=0.0)
209: (8)                    pytest.raises(TypeError, nx.empty_graph, create_using="Graph")
210: (8)                    G = nx.empty_graph(create_using=nx.MultiGraph)
211: (8)                    assert isinstance(G, nx.MultiGraph)
212: (8)                    G = nx.empty_graph(create_using=nx.DiGraph)
213: (8)                    assert isinstance(G, nx.DiGraph)
214: (8)                    G = nx.empty_graph(create_using=nx.DiGraph, default=nx.MultiGraph)
215: (8)                    assert isinstance(G, nx.DiGraph)
216: (8)                    G = nx.empty_graph(create_using=None, default=nx.MultiGraph)
217: (8)                    assert isinstance(G, nx.MultiGraph)
218: (8)                    G = nx.empty_graph(default=nx.MultiGraph)
219: (8)                    assert isinstance(G, nx.MultiGraph)
220: (8)                    G = nx.path_graph(5)
221: (8)                    H = nx.empty_graph(create_using=G)
222: (8)                    assert not H.is_multigraph()
223: (8)                    assert not H.is_directed()
224: (8)                    assert len(H) == 0
225: (8)                    assert G is H
226: (8)                    H = nx.empty_graph(create_using=nx.MultiGraph())
227: (8)                    assert H.is_multigraph()
228: (8)                    assert not H.is_directed()
229: (8)                    assert G is not H
230: (8)                    class Mixin(typing.Protocol):
231: (12)                       pass
232: (8)                    class MyGraph(Mixin, nx.DiGraph):
233: (12)                       pass
234: (8)                    G = nx.empty_graph(create_using=MyGraph)
235: (4)                def test_empty_graph(self):
236: (8)                    G = nx.empty_graph()
237: (8)                    assert nx.number_of_nodes(G) == 0
238: (8)                    G = nx.empty_graph(42)
239: (8)                    assert nx.number_of_nodes(G) == 42
240: (8)                    assert nx.number_of_edges(G) == 0
241: (8)                    G = nx.empty_graph("abc")
242: (8)                    assert len(G) == 3
243: (8)                    assert G.size() == 0
244: (8)                    G = nx.empty_graph(42, create_using=nx.DiGraph(name="duh"))
245: (8)                    assert nx.number_of_nodes(G) == 42
246: (8)                    assert nx.number_of_edges(G) == 0
247: (8)                    assert isinstance(G, nx.DiGraph)
248: (8)                    G = nx.empty_graph(42, create_using=nx.MultiGraph(name="duh"))
249: (8)                    assert nx.number_of_nodes(G) == 42
250: (8)                    assert nx.number_of_edges(G) == 0
251: (8)                    assert isinstance(G, nx.MultiGraph)
252: (8)                    pete = nx.petersen_graph()
253: (8)                    G = nx.empty_graph(42, create_using=pete)
254: (8)                    assert nx.number_of_nodes(G) == 42
255: (8)                    assert nx.number_of_edges(G) == 0
256: (8)                    assert isinstance(G, nx.Graph)
257: (4)                def test_ladder_graph(self):
258: (8)                    for i, G in [
259: (12)                       (0, nx.empty_graph(0)),
260: (12)                       (1, nx.path_graph(2)),
```

```
261: (12)                        (2, nx.hypercube_graph(2)),
262: (12)                        (10, nx.grid_graph([2, 10])),
263: (8)                     ]:
264: (12)                        assert is_isomorphic(nx.ladder_graph(i), G)
265: (8)                     pytest.raises(nx.NetworkXError, nx.ladder_graph, 2,
create_using=nx.DiGraph)
266: (8)                     g = nx.ladder_graph(2)
267: (8)                     mg = nx.ladder_graph(2, create_using=nx.MultiGraph)
268: (8)                     assert edges_equal(mg.edges(), g.edges())
269: (4)                 @pytest.mark.parametrize(("m", "n"), [(3, 5), (4, 10), (3, 20)])
270: (4)                 def test_lollipop_graph_right_sizes(self, m, n):
271: (8)                     G = nx.lollipop_graph(m, n)
272: (8)                     assert nx.number_of_nodes(G) == m + n
273: (8)                     assert nx.number_of_edges(G) == m * (m - 1) / 2 + n
274: (4)                 @pytest.mark.parametrize(("m", "n"), [("ab", ""), ("abc", "defg")])
275: (4)                 def test_lollipop_graph_size_node_sequence(self, m, n):
276: (8)                     G = nx.lollipop_graph(m, n)
277: (8)                     assert nx.number_of_nodes(G) == len(m) + len(n)
278: (8)                     assert nx.number_of_edges(G) == len(m) * (len(m) - 1) / 2 + len(n)
279: (4)                 def test_lollipop_graph_exceptions(self):
280: (8)                     pytest.raises(nx.NetworkXError, nx.lollipop_graph, -1, 2)
281: (8)                     pytest.raises(nx.NetworkXError, nx.lollipop_graph, 1, 20)
282: (8)                     pytest.raises(nx.NetworkXError, nx.lollipop_graph, "", 20)
283: (8)                     pytest.raises(nx.NetworkXError, nx.lollipop_graph, "a", 20)
284: (8)                     pytest.raises(nx.NetworkXError, nx.lollipop_graph, 5, -2)
285: (8)                     with pytest.raises(nx.NetworkXError):
286: (12)                        nx.lollipop_graph(2, 20, create_using=nx.DiGraph)
287: (8)                     with pytest.raises(nx.NetworkXError):
288: (12)                        nx.lollipop_graph(2, 20, create_using=nx.MultiDiGraph)
289: (4)                 @pytest.mark.parametrize(("m", "n"), [(2, 0), (2, 5), (2, 10), ("ab",
20)])
290: (4)                 def test_lollipop_graph_same_as_path_when_m1_is_2(self, m, n):
291: (8)                     G = nx.lollipop_graph(m, n)
292: (8)                     assert is_isomorphic(G, nx.path_graph(n + 2))
293: (4)                 def test_lollipop_graph_for_multigraph(self):
294: (8)                     G = nx.lollipop_graph(5, 20)
295: (8)                     MG = nx.lollipop_graph(5, 20, create_using=nx.MultiGraph)
296: (8)                     assert edges_equal(MG.edges(), G.edges())
297: (4)                 @pytest.mark.parametrize(
298: (8)                     ("m", "n"),
299: (8)                     [(4, "abc"), ("abcd", 3), ([1, 2, 3, 4], "abc"), ("abcd", [1, 2, 3])],
300: (4)                 )
301: (4)                 def test_lollipop_graph_mixing_input_types(self, m, n):
302: (8)                     expected = nx.compose(nx.complete_graph(4), nx.path_graph(range(100,
103)))
303: (8)                     expected.add_edge(0, 100)  # Connect complete graph and path graph
304: (8)                     assert is_isomorphic(nx.lollipop_graph(m, n), expected)
305: (4)                 def test_lollipop_graph_non_builtin_ints(self):
306: (8)                     np = pytest.importorskip("numpy")
307: (8)                     G = nx.lollipop_graph(np.int32(4), np.int64(3))
308: (8)                     expected = nx.compose(nx.complete_graph(4), nx.path_graph(range(100,
103)))
309: (8)                     expected.add_edge(0, 100)  # Connect complete graph and path graph
310: (8)                     assert is_isomorphic(G, expected)
311: (4)                 def test_null_graph(self):
312: (8)                     assert nx.number_of_nodes(nx.null_graph()) == 0
313: (4)                 def test_path_graph(self):
314: (8)                     p = nx.path_graph(0)
315: (8)                     assert is_isomorphic(p, nx.null_graph())
316: (8)                     p = nx.path_graph(1)
317: (8)                     assert is_isomorphic(p, nx.empty_graph(1))
318: (8)                     p = nx.path_graph(10)
319: (8)                     assert nx.is_connected(p)
320: (8)                     assert sorted(d for n, d in p.degree()) == [1, 1, 2, 2, 2, 2, 2, 2, 2,
2]
321: (8)                     assert p.order() - 1 == p.size()
322: (8)                     dp = nx.path_graph(3, create_using=nx.DiGraph)
323: (8)                     assert dp.has_edge(0, 1)
324: (8)                     assert not dp.has_edge(1, 0)
```

```
325: (8)                    mp = nx.path_graph(10, create_using=nx.MultiGraph)
326: (8)                    assert edges_equal(mp.edges(), p.edges())
327: (8)                    G = nx.path_graph("abc")
328: (8)                    assert len(G) == 3
329: (8)                    assert G.size() == 2
330: (8)                    G = nx.path_graph("abcb")
331: (8)                    assert len(G) == 3
332: (8)                    assert G.size() == 2
333: (8)                    g = nx.path_graph("abc", nx.DiGraph)
334: (8)                    assert len(g) == 3
335: (8)                    assert g.size() == 2
336: (8)                    assert g.is_directed()
337: (8)                    g = nx.path_graph("abcb", nx.DiGraph)
338: (8)                    assert len(g) == 3
339: (8)                    assert g.size() == 3
340: (8)                    G = nx.path_graph((1, 2, 3, 2, 4))
341: (8)                    assert G.has_edge(2, 4)
342: (4)                def test_star_graph(self):
343: (8)                    assert is_isomorphic(nx.star_graph(""), nx.empty_graph(0))
344: (8)                    assert is_isomorphic(nx.star_graph([]), nx.empty_graph(0))
345: (8)                    assert is_isomorphic(nx.star_graph(0), nx.empty_graph(1))
346: (8)                    assert is_isomorphic(nx.star_graph(1), nx.path_graph(2))
347: (8)                    assert is_isomorphic(nx.star_graph(2), nx.path_graph(3))
348: (8)                    assert is_isomorphic(nx.star_graph(5), nx.complete_bipartite_graph(1,
5))
349: (8)                    s = nx.star_graph(10)
350: (8)                    assert sorted(d for n, d in s.degree()) == [1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 10]
351: (8)                    pytest.raises(nx.NetworkXError, nx.star_graph, 10,
create_using=nx.DiGraph)
352: (8)                    ms = nx.star_graph(10, create_using=nx.MultiGraph)
353: (8)                    assert edges_equal(ms.edges(), s.edges())
354: (8)                    G = nx.star_graph("abc")
355: (8)                    assert len(G) == 3
356: (8)                    assert G.size() == 2
357: (8)                    G = nx.star_graph("abcb")
358: (8)                    assert len(G) == 3
359: (8)                    assert G.size() == 2
360: (8)                    G = nx.star_graph("abcb", create_using=nx.MultiGraph)
361: (8)                    assert len(G) == 3
362: (8)                    assert G.size() == 3
363: (8)                    G = nx.star_graph("abcdefg")
364: (8)                    assert len(G) == 7
365: (8)                    assert G.size() == 6
366: (4)                def test_non_int_integers_for_star_graph(self):
367: (8)                    np = pytest.importorskip("numpy")
368: (8)                    G = nx.star_graph(np.int32(3))
369: (8)                    assert len(G) == 4
370: (8)                    assert G.size() == 3
371: (4)                @pytest.mark.parametrize(("m", "n"), [(3, 0), (3, 5), (4, 10), (3, 20)])
372: (4)                def test_tadpole_graph_right_sizes(self, m, n):
373: (8)                    G = nx.tadpole_graph(m, n)
374: (8)                    assert nx.number_of_nodes(G) == m + n
375: (8)                    assert nx.number_of_edges(G) == m + n - (m == 2)
376: (4)                @pytest.mark.parametrize(("m", "n"), [("ab", ""), ("ab", "c"), ("abc",
"defg")])
377: (4)                def test_tadpole_graph_size_node_sequences(self, m, n):
378: (8)                    G = nx.tadpole_graph(m, n)
379: (8)                    assert nx.number_of_nodes(G) == len(m) + len(n)
380: (8)                    assert nx.number_of_edges(G) == len(m) + len(n) - (len(m) == 2)
381: (4)                def test_tadpole_graph_exceptions(self):
382: (8)                    pytest.raises(nx.NetworkXError, nx.tadpole_graph, -1, 3)
383: (8)                    pytest.raises(nx.NetworkXError, nx.tadpole_graph, 0, 3)
384: (8)                    pytest.raises(nx.NetworkXError, nx.tadpole_graph, 1, 3)
385: (8)                    pytest.raises(nx.NetworkXError, nx.tadpole_graph, 5, -2)
386: (8)                    with pytest.raises(nx.NetworkXError):
387: (12)                       nx.tadpole_graph(2, 20, create_using=nx.DiGraph)
388: (8)                    with pytest.raises(nx.NetworkXError):
389: (12)                       nx.tadpole_graph(2, 20, create_using=nx.MultiDiGraph)
```

```
    @pytest.mark.parametrize(("m", "n"), [(2, 0), (2, 5), (2, 10), ("ab", 20)])
    def test_tadpole_graph_same_as_path_when_m_is_2(self, m, n):
        G = nx.tadpole_graph(m, n)
        assert is_isomorphic(G, nx.path_graph(n + 2))
    @pytest.mark.parametrize("m", [4, 7])
    def test_tadpole_graph_same_as_cycle_when_m2_is_0(self, m):
        G = nx.tadpole_graph(m, 0)
        assert is_isomorphic(G, nx.cycle_graph(m))
    def test_tadpole_graph_for_multigraph(self):
        G = nx.tadpole_graph(5, 20)
        MG = nx.tadpole_graph(5, 20, create_using=nx.MultiGraph)
        assert edges_equal(MG.edges(), G.edges())
    @pytest.mark.parametrize(
        ("m", "n"),
        [(4, "abc"), ("abcd", 3), ([1, 2, 3, 4], "abc"), ("abcd", [1, 2, 3])],
    )
    def test_tadpole_graph_mixing_input_types(self, m, n):
        expected = nx.compose(nx.cycle_graph(4), nx.path_graph(range(100, 103)))
        expected.add_edge(0, 100)  # Connect cycle and path
        assert is_isomorphic(nx.tadpole_graph(m, n), expected)
    def test_tadpole_graph_non_builtin_integers(self):
        np = pytest.importorskip("numpy")
        G = nx.tadpole_graph(np.int32(4), np.int64(3))
        expected = nx.compose(nx.cycle_graph(4), nx.path_graph(range(100, 103)))
        expected.add_edge(0, 100)  # Connect cycle and path
        assert is_isomorphic(G, expected)
    def test_trivial_graph(self):
        assert nx.number_of_nodes(nx.trivial_graph()) == 1
    def test_turan_graph(self):
        assert nx.number_of_edges(nx.turan_graph(13, 4)) == 63
        assert is_isomorphic(
            nx.turan_graph(13, 4), nx.complete_multipartite_graph(3, 4, 3, 3)
        )
    def test_wheel_graph(self):
        for n, G in [
            ("", nx.null_graph()),
            (0, nx.null_graph()),
            (1, nx.empty_graph(1)),
            (2, nx.path_graph(2)),
            (3, nx.complete_graph(3)),
            (4, nx.complete_graph(4)),
        ]:
            g = nx.wheel_graph(n)
            assert is_isomorphic(g, G)
        g = nx.wheel_graph(10)
        assert sorted(d for n, d in g.degree()) == [3, 3, 3, 3, 3, 3, 3, 3, 3, 9]
        pytest.raises(nx.NetworkXError, nx.wheel_graph, 10, create_using=nx.DiGraph)
        mg = nx.wheel_graph(10, create_using=nx.MultiGraph())
        assert edges_equal(mg.edges(), g.edges())
        G = nx.wheel_graph("abc")
        assert len(G) == 3
        assert G.size() == 3
        G = nx.wheel_graph("abcb")
        assert len(G) == 3
        assert G.size() == 4
        G = nx.wheel_graph("abcb", nx.MultiGraph)
        assert len(G) == 3
        assert G.size() == 6
    def test_non_int_integers_for_wheel_graph(self):
        np = pytest.importorskip("numpy")
        G = nx.wheel_graph(np.int32(3))
        assert len(G) == 3
        assert G.size() == 3
    def test_complete_0_partite_graph(self):
```

```
454: (8)                    """Tests that the complete 0-partite graph is the null graph."""
455: (8)                    G = nx.complete_multipartite_graph()
456: (8)                    H = nx.null_graph()
457: (8)                    assert nodes_equal(G, H)
458: (8)                    assert edges_equal(G.edges(), H.edges())
459: (4)                def test_complete_1_partite_graph(self):
460: (8)                    """Tests that the complete 1-partite graph is the empty graph."""
461: (8)                    G = nx.complete_multipartite_graph(3)
462: (8)                    H = nx.empty_graph(3)
463: (8)                    assert nodes_equal(G, H)
464: (8)                    assert edges_equal(G.edges(), H.edges())
465: (4)                def test_complete_2_partite_graph(self):
466: (8)                    """Tests that the complete 2-partite graph is the complete bipartite
467: (8)                    graph.
468: (8)                    """
469: (8)                    G = nx.complete_multipartite_graph(2, 3)
470: (8)                    H = nx.complete_bipartite_graph(2, 3)
471: (8)                    assert nodes_equal(G, H)
472: (8)                    assert edges_equal(G.edges(), H.edges())
473: (4)                def test_complete_multipartite_graph(self):
474: (8)                    """Tests for generating the complete multipartite graph."""
475: (8)                    G = nx.complete_multipartite_graph(2, 3, 4)
476: (8)                    blocks = [(0, 1), (2, 3, 4), (5, 6, 7, 8)]
477: (8)                    for block in blocks:
478: (12)                       for u, v in itertools.combinations_with_replacement(block, 2):
479: (16)                           assert v not in G[u]
480: (16)                           assert G.nodes[u] == G.nodes[v]
481: (8)                    for block1, block2 in itertools.combinations(blocks, 2):
482: (12)                       for u, v in itertools.product(block1, block2):
483: (16)                           assert v in G[u]
484: (16)                           assert G.nodes[u] != G.nodes[v]
485: (8)                    with pytest.raises(nx.NetworkXError, match="Negative number of
nodes"):
486: (12)                       nx.complete_multipartite_graph(2, -3, 4)
487: (4)                def test_kneser_graph(self):
488: (8)                    assert is_isomorphic(nx.kneser_graph(5, 2), nx.petersen_graph())
489: (8)                    for i in range(1, 7):
490: (12)                       assert is_isomorphic(nx.kneser_graph(i, 1), nx.complete_graph(i))
491: (8)                    for j in range(3, 7):
492: (12)                       assert is_isomorphic(nx.kneser_graph(j, j - 1), nx.empty_graph(j))
493: (8)                    assert nx.number_of_edges(nx.kneser_graph(8, 3)) == 280
```

File 459 - test_cographs.py:

```
1: (0)              """Unit tests for the :mod:`networkx.generators.cographs` module.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              def test_random_cograph():
5: (4)                  n = 3
6: (4)                  G = nx.random_cograph(n)
7: (4)                  assert len(G) == 2**n
8: (4)                  if nx.is_connected(G):
9: (8)                      assert nx.diameter(G) <= 2
10: (4)                 else:
11: (8)                     components = nx.connected_components(G)
12: (8)                     for component in components:
13: (12)                        assert nx.diameter(G.subgraph(component)) <= 2
```

File 460 - test_community.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_random_partition_graph():
4: (4)                  G = nx.random_partition_graph([3, 3, 3], 1, 0, seed=42)
5: (4)                  C = G.graph["partition"]
```

```
 6: (4)                 assert C == [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}]
 7: (4)                 assert len(G) == 9
 8: (4)                 assert len(list(G.edges())) == 9
 9: (4)                 G = nx.random_partition_graph([3, 3, 3], 0, 1)
10: (4)                 C = G.graph["partition"]
11: (4)                 assert C == [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}]
12: (4)                 assert len(G) == 9
13: (4)                 assert len(list(G.edges())) == 27
14: (4)                 G = nx.random_partition_graph([3, 3, 3], 1, 0, directed=True)
15: (4)                 C = G.graph["partition"]
16: (4)                 assert C == [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}]
17: (4)                 assert len(G) == 9
18: (4)                 assert len(list(G.edges())) == 18
19: (4)                 G = nx.random_partition_graph([3, 3, 3], 0, 1, directed=True)
20: (4)                 C = G.graph["partition"]
21: (4)                 assert C == [{0, 1, 2}, {3, 4, 5}, {6, 7, 8}]
22: (4)                 assert len(G) == 9
23: (4)                 assert len(list(G.edges())) == 54
24: (4)                 G = nx.random_partition_graph([1, 2, 3, 4, 5], 0.5, 0.1)
25: (4)                 C = G.graph["partition"]
26: (4)                 assert C == [{0}, {1, 2}, {3, 4, 5}, {6, 7, 8, 9}, {10, 11, 12, 13, 14}]
27: (4)                 assert len(G) == 15
28: (4)                 rpg = nx.random_partition_graph
29: (4)                 pytest.raises(nx.NetworkXError, rpg, [1, 2, 3], 1.1, 0.1)
30: (4)                 pytest.raises(nx.NetworkXError, rpg, [1, 2, 3], -0.1, 0.1)
31: (4)                 pytest.raises(nx.NetworkXError, rpg, [1, 2, 3], 0.1, 1.1)
32: (4)                 pytest.raises(nx.NetworkXError, rpg, [1, 2, 3], 0.1, -0.1)
33: (0)             def test_planted_partition_graph():
34: (4)                 G = nx.planted_partition_graph(4, 3, 1, 0, seed=42)
35: (4)                 C = G.graph["partition"]
36: (4)                 assert len(C) == 4
37: (4)                 assert len(G) == 12
38: (4)                 assert len(list(G.edges())) == 12
39: (4)                 G = nx.planted_partition_graph(4, 3, 0, 1)
40: (4)                 C = G.graph["partition"]
41: (4)                 assert len(C) == 4
42: (4)                 assert len(G) == 12
43: (4)                 assert len(list(G.edges())) == 54
44: (4)                 G = nx.planted_partition_graph(10, 4, 0.5, 0.1, seed=42)
45: (4)                 C = G.graph["partition"]
46: (4)                 assert len(C) == 10
47: (4)                 assert len(G) == 40
48: (4)                 G = nx.planted_partition_graph(4, 3, 1, 0, directed=True)
49: (4)                 C = G.graph["partition"]
50: (4)                 assert len(C) == 4
51: (4)                 assert len(G) == 12
52: (4)                 assert len(list(G.edges())) == 24
53: (4)                 G = nx.planted_partition_graph(4, 3, 0, 1, directed=True)
54: (4)                 C = G.graph["partition"]
55: (4)                 assert len(C) == 4
56: (4)                 assert len(G) == 12
57: (4)                 assert len(list(G.edges())) == 108
58: (4)                 G = nx.planted_partition_graph(10, 4, 0.5, 0.1, seed=42, directed=True)
59: (4)                 C = G.graph["partition"]
60: (4)                 assert len(C) == 10
61: (4)                 assert len(G) == 40
62: (4)                 ppg = nx.planted_partition_graph
63: (4)                 pytest.raises(nx.NetworkXError, ppg, 3, 3, 1.1, 0.1)
64: (4)                 pytest.raises(nx.NetworkXError, ppg, 3, 3, -0.1, 0.1)
65: (4)                 pytest.raises(nx.NetworkXError, ppg, 3, 3, 0.1, 1.1)
66: (4)                 pytest.raises(nx.NetworkXError, ppg, 3, 3, 0.1, -0.1)
67: (0)             def test_relaxed_caveman_graph():
68: (4)                 G = nx.relaxed_caveman_graph(4, 3, 0)
69: (4)                 assert len(G) == 12
70: (4)                 G = nx.relaxed_caveman_graph(4, 3, 1)
71: (4)                 assert len(G) == 12
72: (4)                 G = nx.relaxed_caveman_graph(4, 3, 0.5)
73: (4)                 assert len(G) == 12
74: (4)                 G = nx.relaxed_caveman_graph(4, 3, 0.5, seed=42)
```

```
 75: (4)                 assert len(G) == 12
 76: (0)             def test_connected_caveman_graph():
 77: (4)                 G = nx.connected_caveman_graph(4, 3)
 78: (4)                 assert len(G) == 12
 79: (4)                 G = nx.connected_caveman_graph(1, 5)
 80: (4)                 K5 = nx.complete_graph(5)
 81: (4)                 K5.remove_edge(3, 4)
 82: (4)                 assert nx.is_isomorphic(G, K5)
 83: (4)                 pytest.raises(nx.NetworkXError, nx.connected_caveman_graph, 4, 1)
 84: (0)             def test_caveman_graph():
 85: (4)                 G = nx.caveman_graph(4, 3)
 86: (4)                 assert len(G) == 12
 87: (4)                 G = nx.caveman_graph(5, 1)
 88: (4)                 E5 = nx.empty_graph(5)
 89: (4)                 assert nx.is_isomorphic(G, E5)
 90: (4)                 G = nx.caveman_graph(1, 5)
 91: (4)                 K5 = nx.complete_graph(5)
 92: (4)                 assert nx.is_isomorphic(G, K5)
 93: (0)             def test_gaussian_random_partition_graph():
 94: (4)                 G = nx.gaussian_random_partition_graph(100, 10, 10, 0.3, 0.01)
 95: (4)                 assert len(G) == 100
 96: (4)                 G = nx.gaussian_random_partition_graph(100, 10, 10, 0.3, 0.01,
directed=True)
 97: (4)                 assert len(G) == 100
 98: (4)                 G = nx.gaussian_random_partition_graph(
 99: (8)                     100, 10, 10, 0.3, 0.01, directed=False, seed=42
100: (4)                 )
101: (4)                 assert len(G) == 100
102: (4)                 assert not isinstance(G, nx.DiGraph)
103: (4)                 G = nx.gaussian_random_partition_graph(
104: (8)                     100, 10, 10, 0.3, 0.01, directed=True, seed=42
105: (4)                 )
106: (4)                 assert len(G) == 100
107: (4)                 assert isinstance(G, nx.DiGraph)
108: (4)                 pytest.raises(
109: (8)                     nx.NetworkXError, nx.gaussian_random_partition_graph, 100, 101, 10, 1,
0
110: (4)                 )
111: (4)                 G = nx.gaussian_random_partition_graph(10, 0.5, 0.5, 0.5, 0.5, seed=1)
112: (4)                 assert len(G) == 10
113: (0)             def test_ring_of_cliques():
114: (4)                 for i in range(2, 20, 3):
115: (8)                     for j in range(2, 20, 3):
116: (12)                        G = nx.ring_of_cliques(i, j)
117: (12)                        assert G.number_of_nodes() == i * j
118: (12)                        if i != 2 or j != 1:
119: (16)                            expected_num_edges = i * (((j * (j - 1)) // 2) + 1)
120: (12)                        else:
121: (16)                            expected_num_edges = i * (((j * (j - 1)) // 2) + 1) - 1
122: (12)                        assert G.number_of_edges() == expected_num_edges
123: (4)                 with pytest.raises(
124: (8)                     nx.NetworkXError, match="A ring of cliques must have at least two
cliques"
125: (4)                 ):
126: (8)                     nx.ring_of_cliques(1, 5)
127: (4)                 with pytest.raises(
128: (8)                     nx.NetworkXError, match="The cliques must have at least two nodes"
129: (4)                 ):
130: (8)                     nx.ring_of_cliques(3, 0)
131: (0)             def test_windmill_graph():
132: (4)                 for n in range(2, 20, 3):
133: (8)                     for k in range(2, 20, 3):
134: (12)                        G = nx.windmill_graph(n, k)
135: (12)                        assert G.number_of_nodes() == (k - 1) * n + 1
136: (12)                        assert G.number_of_edges() == n * k * (k - 1) / 2
137: (12)                        assert G.degree(0) == G.number_of_nodes() - 1
138: (12)                        for i in range(1, G.number_of_nodes()):
139: (16)                            assert G.degree(i) == k - 1
140: (4)                 with pytest.raises(
```

```
141: (8)                    nx.NetworkXError, match="A windmill graph must have at least two
cliques"
142: (4)                ):
143: (8)                    nx.windmill_graph(1, 3)
144: (4)                with pytest.raises(
145: (8)                    nx.NetworkXError, match="The cliques must have at least two nodes"
146: (4)                ):
147: (8)                    nx.windmill_graph(3, 0)
148: (0)            def test_stochastic_block_model():
149: (4)                sizes = [75, 75, 300]
150: (4)                probs = [[0.25, 0.05, 0.02], [0.05, 0.35, 0.07], [0.02, 0.07, 0.40]]
151: (4)                G = nx.stochastic_block_model(sizes, probs, seed=0)
152: (4)                C = G.graph["partition"]
153: (4)                assert len(C) == 3
154: (4)                assert len(G) == 450
155: (4)                assert G.size() == 22160
156: (4)                GG = nx.stochastic_block_model(sizes, probs, range(450), seed=0)
157: (4)                assert G.nodes == GG.nodes
158: (4)                sbm = nx.stochastic_block_model
159: (4)                badnodelist = list(range(400))  # not enough nodes to match sizes
160: (4)                badprobs1 = [[0.25, 0.05, 1.02], [0.05, 0.35, 0.07], [0.02, 0.07, 0.40]]
161: (4)                badprobs2 = [[0.25, 0.05, 0.02], [0.05, -0.35, 0.07], [0.02, 0.07, 0.40]]
162: (4)                probs_rect1 = [[0.25, 0.05, 0.02], [0.05, -0.35, 0.07]]
163: (4)                probs_rect2 = [[0.25, 0.05], [0.05, -0.35], [0.02, 0.07]]
164: (4)                asymprobs = [[0.25, 0.05, 0.01], [0.05, -0.35, 0.07], [0.02, 0.07, 0.40]]
165: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, badprobs1)
166: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, badprobs2)
167: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, probs_rect1,
directed=True)
168: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, probs_rect2,
directed=True)
169: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, asymprobs, directed=False)
170: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, probs, badnodelist)
171: (4)                nodelist = [0] + list(range(449))  # repeated node name in nodelist
172: (4)                pytest.raises(nx.NetworkXException, sbm, sizes, probs, nodelist)
173: (4)                GG = nx.stochastic_block_model(sizes, probs, seed=0, selfloops=True)
174: (4)                assert G.nodes == GG.nodes
175: (4)                GG = nx.stochastic_block_model(sizes, probs, selfloops=True,
directed=True)
176: (4)                assert G.nodes == GG.nodes
177: (4)                GG = nx.stochastic_block_model(sizes, probs, seed=0, sparse=False)
178: (4)                assert G.nodes == GG.nodes
179: (0)            def test_generator():
180: (4)                n = 250
181: (4)                tau1 = 3
182: (4)                tau2 = 1.5
183: (4)                mu = 0.1
184: (4)                G = nx.LFR_benchmark_graph(
185: (8)                    n, tau1, tau2, mu, average_degree=5, min_community=20, seed=10
186: (4)                )
187: (4)                assert len(G) == 250
188: (4)                C = {frozenset(G.nodes[v]["community"]) for v in G}
189: (4)                assert nx.community.is_partition(G.nodes(), C)
190: (0)            def test_invalid_tau1():
191: (4)                with pytest.raises(nx.NetworkXError, match="tau2 must be greater than
one"):
192: (8)                    n = 100
193: (8)                    tau1 = 2
194: (8)                    tau2 = 1
195: (8)                    mu = 0.1
196: (8)                    nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2)
197: (0)            def test_invalid_tau2():
198: (4)                with pytest.raises(nx.NetworkXError, match="tau1 must be greater than
one"):
199: (8)                    n = 100
200: (8)                    tau1 = 1
201: (8)                    tau2 = 2
202: (8)                    mu = 0.1
203: (8)                    nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2)
```

```
(0)             def test_mu_too_large():
(4)                 with pytest.raises(nx.NetworkXError, match="mu must be in the interval \\
[0, 1\\]"):
(8)                     n = 100
(8)                     tau1 = 2
(8)                     tau2 = 2
(8)                     mu = 1.1
(8)                     nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2)
(0)             def test_mu_too_small():
(4)                 with pytest.raises(nx.NetworkXError, match="mu must be in the interval \\
[0, 1\\]"):
(8)                     n = 100
(8)                     tau1 = 2
(8)                     tau2 = 2
(8)                     mu = -1
(8)                     nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2)
(0)             def test_both_degrees_none():
(4)                 with pytest.raises(
(8)                     nx.NetworkXError,
(8)                     match="Must assign exactly one of min_degree and average_degree",
(4)                 ):
(8)                     n = 100
(8)                     tau1 = 2
(8)                     tau2 = 2
(8)                     mu = 1
(8)                     nx.LFR_benchmark_graph(n, tau1, tau2, mu)
(0)             def test_neither_degrees_none():
(4)                 with pytest.raises(
(8)                     nx.NetworkXError,
(8)                     match="Must assign exactly one of min_degree and average_degree",
(4)                 ):
(8)                     n = 100
(8)                     tau1 = 2
(8)                     tau2 = 2
(8)                     mu = 1
(8)                     nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2,
average_degree=5)
(0)             def test_max_iters_exceeded():
(4)                 with pytest.raises(
(8)                     nx.ExceededMaxIterations,
(8)                     match="Could not assign communities; try increasing min_community",
(4)                 ):
(8)                     n = 10
(8)                     tau1 = 2
(8)                     tau2 = 2
(8)                     mu = 0.1
(8)                     nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2, max_iters=10,
seed=1)
(0)             def test_max_deg_out_of_range():
(4)                 with pytest.raises(
(8)                     nx.NetworkXError, match="max_degree must be in the interval \\(0,
n\\]"
(4)                 ):
(8)                     n = 10
(8)                     tau1 = 2
(8)                     tau2 = 2
(8)                     mu = 0.1
(8)                     nx.LFR_benchmark_graph(
(12)                        n, tau1, tau2, mu, max_degree=n + 1, max_iters=10, seed=1
(8)                     )
(0)             def test_max_community():
(4)                 n = 250
(4)                 tau1 = 3
(4)                 tau2 = 1.5
(4)                 mu = 0.1
(4)                 G = nx.LFR_benchmark_graph(
(8)                     n,
(8)                     tau1,
(8)                     tau2,
```

```
268: (8)                    mu,
269: (8)                    average_degree=5,
270: (8)                    max_degree=100,
271: (8)                    min_community=50,
272: (8)                    max_community=200,
273: (8)                    seed=10,
274: (4)                )
275: (4)                assert len(G) == 250
276: (4)                C = {frozenset(G.nodes[v]["community"]) for v in G}
277: (4)                assert nx.community.is_partition(G.nodes(), C)
278: (0)            def test_powerlaw_iterations_exceeded():
279: (4)                with pytest.raises(
280: (8)                    nx.ExceededMaxIterations, match="Could not create power law sequence"
281: (4)                ):
282: (8)                    n = 100
283: (8)                    tau1 = 2
284: (8)                    tau2 = 2
285: (8)                    mu = 1
286: (8)                    nx.LFR_benchmark_graph(n, tau1, tau2, mu, min_degree=2, max_iters=0)
287: (0)            def test_no_scipy_zeta():
288: (4)                zeta2 = 1.6449340668482264
289: (4)                assert abs(zeta2 - nx.generators.community._hurwitz_zeta(2, 1, 0.0001)) <
0.01
290: (0)            def test_generate_min_degree_itr():
291: (4)                with pytest.raises(
292: (8)                    nx.ExceededMaxIterations, match="Could not match average_degree"
293: (4)                ):
294: (8)                    nx.generators.community._generate_min_degree(2, 2, 1, 0.01, 0)
```

File 461 - test_directed.py:

```
1: (0)              """Generators - Directed Graphs
2: (0)              ----------------------------
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx.classes import Graph, MultiDiGraph
7: (0)              from networkx.generators.directed import (
8: (4)                  gn_graph,
9: (4)                  gnc_graph,
10: (4)                 gnr_graph,
11: (4)                 random_k_out_graph,
12: (4)                 random_uniform_k_out_graph,
13: (4)                 scale_free_graph,
14: (0)             )
15: (0)             class TestGeneratorsDirected:
16: (4)                 def test_smoke_test_random_graphs(self):
17: (8)                     gn_graph(100)
18: (8)                     gnr_graph(100, 0.5)
19: (8)                     gnc_graph(100)
20: (8)                     scale_free_graph(100)
21: (8)                     gn_graph(100, seed=42)
22: (8)                     gnr_graph(100, 0.5, seed=42)
23: (8)                     gnc_graph(100, seed=42)
24: (8)                     scale_free_graph(100, seed=42)
25: (4)                 def test_create_using_keyword_arguments(self):
26: (8)                     pytest.raises(nx.NetworkXError, gn_graph, 100, create_using=Graph())
27: (8)                     pytest.raises(nx.NetworkXError, gnr_graph, 100, 0.5,
create_using=Graph())
28: (8)                     pytest.raises(nx.NetworkXError, gnc_graph, 100, create_using=Graph())
29: (8)                     G = gn_graph(100, seed=1)
30: (8)                     MG = gn_graph(100, create_using=MultiDiGraph(), seed=1)
31: (8)                     assert sorted(G.edges()) == sorted(MG.edges())
32: (8)                     G = gnr_graph(100, 0.5, seed=1)
33: (8)                     MG = gnr_graph(100, 0.5, create_using=MultiDiGraph(), seed=1)
34: (8)                     assert sorted(G.edges()) == sorted(MG.edges())
35: (8)                     G = gnc_graph(100, seed=1)
```

```
(8)                     MG = gnc_graph(100, create_using=MultiDiGraph(), seed=1)
(8)                     assert sorted(G.edges()) == sorted(MG.edges())
(8)                     G = scale_free_graph(
(12)                        100,
(12)                        alpha=0.3,
(12)                        beta=0.4,
(12)                        gamma=0.3,
(12)                        delta_in=0.3,
(12)                        delta_out=0.1,
(12)                        initial_graph=nx.cycle_graph(4, create_using=MultiDiGraph),
(12)                        seed=1,
(8)                     )
(8)                     pytest.raises(ValueError, scale_free_graph, 100, 0.5, 0.4, 0.3)
(8)                     pytest.raises(ValueError, scale_free_graph, 100, alpha=-0.3)
(8)                     pytest.raises(ValueError, scale_free_graph, 100, beta=-0.3)
(8)                     pytest.raises(ValueError, scale_free_graph, 100, gamma=-0.3)
(4)                 def test_parameters(self):
(8)                     G = nx.DiGraph()
(8)                     G.add_node(0)
(8)                     def kernel(x):
(12)                        return x
(8)                     assert nx.is_isomorphic(gn_graph(1), G)
(8)                     assert nx.is_isomorphic(gn_graph(1, kernel=kernel), G)
(8)                     assert nx.is_isomorphic(gnc_graph(1), G)
(8)                     assert nx.is_isomorphic(gnr_graph(1, 0.5), G)
(0)             def test_scale_free_graph_negative_delta():
(4)                 with pytest.raises(ValueError, match="delta_in must be >= 0."):
(8)                     scale_free_graph(10, delta_in=-1)
(4)                 with pytest.raises(ValueError, match="delta_out must be >= 0."):
(8)                     scale_free_graph(10, delta_out=-1)
(0)             def test_non_numeric_ordering():
(4)                 G = MultiDiGraph([("a", "b"), ("b", "c"), ("c", "a")])
(4)                 s = scale_free_graph(3, initial_graph=G)
(4)                 assert len(s) == 3
(4)                 assert len(s.edges) == 3
(0)             @pytest.mark.parametrize("ig", (nx.Graph(), nx.DiGraph([(0, 1)])))
(0)             def test_scale_free_graph_initial_graph_kwarg(ig):
(4)                 with pytest.raises(nx.NetworkXError):
(8)                     scale_free_graph(100, initial_graph=ig)
(0)             class TestRandomKOutGraph:
(4)                 """Unit tests for the
(4)                 :func:`~networkx.generators.directed.random_k_out_graph` function.
(4)                 """
(4)                 def test_regularity(self):
(8)                     """Tests that the generated graph is `k`-out-regular."""
(8)                     n = 10
(8)                     k = 3
(8)                     alpha = 1
(8)                     G = random_k_out_graph(n, k, alpha)
(8)                     assert all(d == k for v, d in G.out_degree())
(8)                     G = random_k_out_graph(n, k, alpha, seed=42)
(8)                     assert all(d == k for v, d in G.out_degree())
(4)                 def test_no_self_loops(self):
(8)                     """Tests for forbidding self-loops."""
(8)                     n = 10
(8)                     k = 3
(8)                     alpha = 1
(8)                     G = random_k_out_graph(n, k, alpha, self_loops=False)
(8)                     assert nx.number_of_selfloops(G) == 0
(4)                 def test_negative_alpha(self):
(8)                     with pytest.raises(ValueError, match="alpha must be positive"):
(12)                        random_k_out_graph(10, 3, -1)
(0)             class TestUniformRandomKOutGraph:
(4)                 """Unit tests for the
(4)                 :func:`~networkx.generators.directed.random_uniform_k_out_graph`
(4)                 function.
(4)                 """
(4)                 def test_regularity(self):
(8)                     """Tests that the generated graph is `k`-out-regular."""
```

```
105: (8)                    n = 10
106: (8)                    k = 3
107: (8)                    G = random_uniform_k_out_graph(n, k)
108: (8)                    assert all(d == k for v, d in G.out_degree())
109: (8)                    G = random_uniform_k_out_graph(n, k, seed=42)
110: (8)                    assert all(d == k for v, d in G.out_degree())
111: (4)                def test_no_self_loops(self):
112: (8)                    """Tests for forbidding self-loops."""
113: (8)                    n = 10
114: (8)                    k = 3
115: (8)                    G = random_uniform_k_out_graph(n, k, self_loops=False)
116: (8)                    assert nx.number_of_selfloops(G) == 0
117: (8)                    assert all(d == k for v, d in G.out_degree())
118: (4)                def test_with_replacement(self):
119: (8)                    n = 10
120: (8)                    k = 3
121: (8)                    G = random_uniform_k_out_graph(n, k, with_replacement=True)
122: (8)                    assert G.is_multigraph()
123: (8)                    assert all(d == k for v, d in G.out_degree())
124: (8)                    n = 10
125: (8)                    k = 9
126: (8)                    G = random_uniform_k_out_graph(n, k, with_replacement=False,
self_loops=False)
127: (8)                    assert nx.number_of_selfloops(G) == 0
128: (8)                    assert all(d == k for v, d in G.out_degree())
129: (4)                def test_without_replacement(self):
130: (8)                    n = 10
131: (8)                    k = 3
132: (8)                    G = random_uniform_k_out_graph(n, k, with_replacement=False)
133: (8)                    assert not G.is_multigraph()
134: (8)                    assert all(d == k for v, d in G.out_degree())

----------------------------------------

File 462 - test_degree_seq.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestConfigurationModel:
4: (4)                  """Unit tests for the :func:`~networkx.configuration_model`
5: (4)                  function.
6: (4)                  """
7: (4)                  def test_empty_degree_sequence(self):
8: (8)                      """Tests that an empty degree sequence yields the null graph."""
9: (8)                      G = nx.configuration_model([])
10: (8)                     assert len(G) == 0
11: (4)                 def test_degree_zero(self):
12: (8)                     """Tests that a degree sequence of all zeros yields the empty
13: (8)                     graph.
14: (8)                     """
15: (8)                     G = nx.configuration_model([0, 0, 0])
16: (8)                     assert len(G) == 3
17: (8)                     assert G.number_of_edges() == 0
18: (4)                 def test_degree_sequence(self):
19: (8)                     """Tests that the degree sequence of the generated graph matches
20: (8)                     the input degree sequence.
21: (8)                     """
22: (8)                     deg_seq = [5, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
23: (8)                     G = nx.configuration_model(deg_seq, seed=12345678)
24: (8)                     assert sorted((d for n, d in G.degree()), reverse=True) == [
25: (12)                        5,
26: (12)                        3,
27: (12)                        3,
28: (12)                        3,
29: (12)                        3,
30: (12)                        2,
31: (12)                        2,
32: (12)                        2,
33: (12)                        1,
```

```
34: (12)                        1,
35: (12)                        1,
36: (8)                     ]
37: (8)                     assert sorted((d for n, d in G.degree(range(len(deg_seq)))),
reverse=True) == [
38: (12)                        5,
39: (12)                        3,
40: (12)                        3,
41: (12)                        3,
42: (12)                        3,
43: (12)                        2,
44: (12)                        2,
45: (12)                        2,
46: (12)                        1,
47: (12)                        1,
48: (12)                        1,
49: (8)                     ]
50: (4)                 def test_random_seed(self):
51: (8)                     """Tests that each call with the same random seed generates the
52: (8)                     same graph.
53: (8)                     """
54: (8)                     deg_seq = [3] * 12
55: (8)                     G1 = nx.configuration_model(deg_seq, seed=1000)
56: (8)                     G2 = nx.configuration_model(deg_seq, seed=1000)
57: (8)                     assert nx.is_isomorphic(G1, G2)
58: (8)                     G1 = nx.configuration_model(deg_seq, seed=10)
59: (8)                     G2 = nx.configuration_model(deg_seq, seed=10)
60: (8)                     assert nx.is_isomorphic(G1, G2)
61: (4)                 def test_directed_disallowed(self):
62: (8)                     """Tests that attempting to create a configuration model graph
63: (8)                     using a directed graph yields an exception.
64: (8)                     """
65: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
66: (12)                        nx.configuration_model([], create_using=nx.DiGraph())
67: (4)                 def test_odd_degree_sum(self):
68: (8)                     """Tests that a degree sequence whose sum is odd yields an
69: (8)                     exception.
70: (8)                     """
71: (8)                     with pytest.raises(nx.NetworkXError):
72: (12)                        nx.configuration_model([1, 2])
73: (0)             def test_directed_configuration_raise_unequal():
74: (4)                 with pytest.raises(nx.NetworkXError):
75: (8)                     zin = [5, 3, 3, 3, 3, 2, 2, 2, 1, 1]
76: (8)                     zout = [5, 3, 3, 3, 3, 2, 2, 2, 1, 2]
77: (8)                     nx.directed_configuration_model(zin, zout)
78: (0)             def test_directed_configuration_model():
79: (4)                 G = nx.directed_configuration_model([], [], seed=0)
80: (4)                 assert len(G) == 0
81: (0)             def test_simple_directed_configuration_model():
82: (4)                 G = nx.directed_configuration_model([1, 1], [1, 1], seed=0)
83: (4)                 assert len(G) == 2
84: (0)             def test_expected_degree_graph_empty():
85: (4)                 deg_seq = []
86: (4)                 G = nx.expected_degree_graph(deg_seq)
87: (4)                 assert dict(G.degree()) == {}
88: (0)             def test_expected_degree_graph():
89: (4)                 deg_seq = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
90: (4)                 G1 = nx.expected_degree_graph(deg_seq, seed=1000)
91: (4)                 assert len(G1) == 12
92: (4)                 G2 = nx.expected_degree_graph(deg_seq, seed=1000)
93: (4)                 assert nx.is_isomorphic(G1, G2)
94: (4)                 G1 = nx.expected_degree_graph(deg_seq, seed=10)
95: (4)                 G2 = nx.expected_degree_graph(deg_seq, seed=10)
96: (4)                 assert nx.is_isomorphic(G1, G2)
97: (0)             def test_expected_degree_graph_selfloops():
98: (4)                 deg_seq = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
99: (4)                 G1 = nx.expected_degree_graph(deg_seq, seed=1000, selfloops=False)
100: (4)                G2 = nx.expected_degree_graph(deg_seq, seed=1000, selfloops=False)
101: (4)                assert nx.is_isomorphic(G1, G2)
```

```
102: (4)                assert len(G1) == 12
103: (0)            def test_expected_degree_graph_skew():
104: (4)                deg_seq = [10, 2, 2, 2, 2]
105: (4)                G1 = nx.expected_degree_graph(deg_seq, seed=1000)
106: (4)                G2 = nx.expected_degree_graph(deg_seq, seed=1000)
107: (4)                assert nx.is_isomorphic(G1, G2)
108: (4)                assert len(G1) == 5
109: (0)            def test_havel_hakimi_construction():
110: (4)                G = nx.havel_hakimi_graph([])
111: (4)                assert len(G) == 0
112: (4)                z = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
113: (4)                pytest.raises(nx.NetworkXError, nx.havel_hakimi_graph, z)
114: (4)                z = ["A", 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
115: (4)                pytest.raises(nx.NetworkXError, nx.havel_hakimi_graph, z)
116: (4)                z = [5, 4, 3, 3, 3, 2, 2, 2]
117: (4)                G = nx.havel_hakimi_graph(z)
118: (4)                G = nx.configuration_model(z)
119: (4)                z = [6, 5, 4, 4, 2, 1, 1, 1]
120: (4)                pytest.raises(nx.NetworkXError, nx.havel_hakimi_graph, z)
121: (4)                z = [10, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2]
122: (4)                G = nx.havel_hakimi_graph(z)
123: (4)                pytest.raises(nx.NetworkXError, nx.havel_hakimi_graph, z,
create_using=nx.DiGraph())
124: (0)            def test_directed_havel_hakimi():
125: (4)                n, r = 100, 10
126: (4)                p = 1.0 / r
127: (4)                for i in range(r):
128: (8)                    G1 = nx.erdos_renyi_graph(n, p * (i + 1), None, True)
129: (8)                    din1 = [d for n, d in G1.in_degree()]
130: (8)                    dout1 = [d for n, d in G1.out_degree()]
131: (8)                    G2 = nx.directed_havel_hakimi_graph(din1, dout1)
132: (8)                    din2 = [d for n, d in G2.in_degree()]
133: (8)                    dout2 = [d for n, d in G2.out_degree()]
134: (8)                    assert sorted(din1) == sorted(din2)
135: (8)                    assert sorted(dout1) == sorted(dout2)
136: (4)                dout = [1000, 3, 3, 3, 3, 2, 2, 2, 1, 1, 1]
137: (4)                din = [103, 102, 102, 102, 102, 102, 102, 102, 102, 102]
138: (4)                pytest.raises(nx.exception.NetworkXError, nx.directed_havel_hakimi_graph,
din, dout)
139: (4)                dout = [1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
140: (4)                din = [2, 2, 2, 2, 2, 2, 2, 2, 0, 2]
141: (4)                G2 = nx.directed_havel_hakimi_graph(din, dout)
142: (4)                dout2 = (d for n, d in G2.out_degree())
143: (4)                din2 = (d for n, d in G2.in_degree())
144: (4)                assert sorted(dout) == sorted(dout2)
145: (4)                assert sorted(din) == sorted(din2)
146: (4)                din = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
147: (4)                pytest.raises(nx.exception.NetworkXError, nx.directed_havel_hakimi_graph,
din, dout)
148: (4)                din = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, -2]
149: (4)                pytest.raises(nx.exception.NetworkXError, nx.directed_havel_hakimi_graph,
din, dout)
150: (0)            def test_degree_sequence_tree():
151: (4)                z = [1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
152: (4)                G = nx.degree_sequence_tree(z)
153: (4)                assert len(G) == len(z)
154: (4)                assert len(list(G.edges())) == sum(z) / 2
155: (4)                pytest.raises(
156: (8)                    nx.NetworkXError, nx.degree_sequence_tree, z,
create_using=nx.DiGraph()
157: (4)                )
158: (4)                z = [1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
159: (4)                pytest.raises(nx.NetworkXError, nx.degree_sequence_tree, z)
160: (0)            def test_random_degree_sequence_graph():
161: (4)                d = [1, 2, 2, 3]
162: (4)                G = nx.random_degree_sequence_graph(d, seed=42)
163: (4)                assert d == sorted(d for n, d in G.degree())
164: (0)            def test_random_degree_sequence_graph_raise():
165: (4)                z = [1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4]
```

```
166: (4)                pytest.raises(nx.NetworkXUnfeasible, nx.random_degree_sequence_graph, z)
167: (0)            def test_random_degree_sequence_large():
168: (4)                G1 = nx.fast_gnp_random_graph(100, 0.1, seed=42)
169: (4)                d1 = (d for n, d in G1.degree())
170: (4)                G2 = nx.random_degree_sequence_graph(d1, seed=42)
171: (4)                d2 = (d for n, d in G2.degree())
172: (4)                assert sorted(d1) == sorted(d2)
```

File 463 - test_ego.py:

```
1: (0)              """
2: (0)              ego graph
3: (0)              ---------
4: (0)              """
5: (0)              import networkx as nx
6: (0)              from networkx.utils import edges_equal, nodes_equal
7: (0)              class TestGeneratorEgo:
8: (4)                  def test_ego(self):
9: (8)                      G = nx.star_graph(3)
10: (8)                     H = nx.ego_graph(G, 0)
11: (8)                     assert nx.is_isomorphic(G, H)
12: (8)                     G.add_edge(1, 11)
13: (8)                     G.add_edge(2, 22)
14: (8)                     G.add_edge(3, 33)
15: (8)                     H = nx.ego_graph(G, 0)
16: (8)                     assert nx.is_isomorphic(nx.star_graph(3), H)
17: (8)                     G = nx.path_graph(3)
18: (8)                     H = nx.ego_graph(G, 0)
19: (8)                     assert edges_equal(H.edges(), [(0, 1)])
20: (8)                     H = nx.ego_graph(G, 0, undirected=True)
21: (8)                     assert edges_equal(H.edges(), [(0, 1)])
22: (8)                     H = nx.ego_graph(G, 0, center=False)
23: (8)                     assert edges_equal(H.edges(), [])
24: (4)                 def test_ego_distance(self):
25: (8)                     G = nx.Graph()
26: (8)                     G.add_edge(0, 1, weight=2, distance=1)
27: (8)                     G.add_edge(1, 2, weight=2, distance=2)
28: (8)                     G.add_edge(2, 3, weight=2, distance=1)
29: (8)                     assert nodes_equal(nx.ego_graph(G, 0, radius=3).nodes(), [0, 1, 2, 3])
30: (8)                     eg = nx.ego_graph(G, 0, radius=3, distance="weight")
31: (8)                     assert nodes_equal(eg.nodes(), [0, 1])
32: (8)                     eg = nx.ego_graph(G, 0, radius=3, distance="weight", undirected=True)
33: (8)                     assert nodes_equal(eg.nodes(), [0, 1])
34: (8)                     eg = nx.ego_graph(G, 0, radius=3, distance="distance")
35: (8)                     assert nodes_equal(eg.nodes(), [0, 1, 2])
```

File 464 - test_duplication.py:

```
1: (0)              """Unit tests for the :mod:`networkx.generators.duplication` module.
2: (0)              """
3: (0)              import pytest
4: (0)              from networkx.exception import NetworkXError
5: (0)              from networkx.generators.duplication import (
6: (4)                  duplication_divergence_graph,
7: (4)                  partial_duplication_graph,
8: (0)              )
9: (0)              class TestDuplicationDivergenceGraph:
10: (4)                 """Unit tests for the
11: (4)                 :func:`networkx.generators.duplication.duplication_divergence_graph`
12: (4)                 function.
13: (4)                 """
14: (4)                 def test_final_size(self):
15: (8)                     G = duplication_divergence_graph(3, p=1)
16: (8)                     assert len(G) == 3
17: (8)                     G = duplication_divergence_graph(3, p=1, seed=42)
```

```
18: (8)                     assert len(G) == 3
19: (4)                 def test_probability_too_large(self):
20: (8)                     with pytest.raises(NetworkXError):
21: (12)                        duplication_divergence_graph(3, p=2)
22: (4)                 def test_probability_too_small(self):
23: (8)                     with pytest.raises(NetworkXError):
24: (12)                        duplication_divergence_graph(3, p=-1)
25: (4)                 def test_non_extreme_probability_value(self):
26: (8)                     G = duplication_divergence_graph(6, p=0.3, seed=42)
27: (8)                     assert len(G) == 6
28: (8)                     assert list(G.degree()) == [(0, 2), (1, 3), (2, 2), (3, 3), (4, 1),
(5, 1)]
29: (4)                 def test_minimum_desired_nodes(self):
30: (8)                     with pytest.raises(
31: (12)                        NetworkXError, match=".*n must be greater than or equal to 2"
32: (8)                     ):
33: (12)                        duplication_divergence_graph(1, p=1)
34: (0)             class TestPartialDuplicationGraph:
35: (4)                 """Unit tests for the
36: (4)                 :func:`networkx.generators.duplication.partial_duplication_graph`
37: (4)                 function.
38: (4)                 """
39: (4)                 def test_final_size(self):
40: (8)                     N = 10
41: (8)                     n = 5
42: (8)                     p = 0.5
43: (8)                     q = 0.5
44: (8)                     G = partial_duplication_graph(N, n, p, q)
45: (8)                     assert len(G) == N
46: (8)                     G = partial_duplication_graph(N, n, p, q, seed=42)
47: (8)                     assert len(G) == N
48: (4)                 def test_initial_clique_size(self):
49: (8)                     N = 10
50: (8)                     n = 10
51: (8)                     p = 0.5
52: (8)                     q = 0.5
53: (8)                     G = partial_duplication_graph(N, n, p, q)
54: (8)                     assert len(G) == n
55: (4)                 def test_invalid_initial_size(self):
56: (8)                     with pytest.raises(NetworkXError):
57: (12)                        N = 5
58: (12)                        n = 10
59: (12)                        p = 0.5
60: (12)                        q = 0.5
61: (12)                        G = partial_duplication_graph(N, n, p, q)
62: (4)                 def test_invalid_probabilities(self):
63: (8)                     N = 1
64: (8)                     n = 1
65: (8)                     for p, q in [(0.5, 2), (0.5, -1), (2, 0.5), (-1, 0.5)]:
66: (12)                        args = (N, n, p, q)
67: (12)                        pytest.raises(NetworkXError, partial_duplication_graph, *args)

----------------------------------------

File 465 - test_expanders.py:

1: (0)              """Unit tests for the :mod:`networkx.generators.expanders` module.
2: (0)              """
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              @pytest.mark.parametrize("n", (2, 3, 5, 6, 10))
6: (0)              def test_margulis_gabber_galil_graph_properties(n):
7: (4)                  g = nx.margulis_gabber_galil_graph(n)
8: (4)                  assert g.number_of_nodes() == n * n
9: (4)                  for node in g:
10: (8)                     assert g.degree(node) == 8
11: (8)                     assert len(node) == 2
12: (8)                     for i in node:
13: (12)                        assert int(i) == i
```

```
14: (12)                        assert 0 <= i < n
15: (0)             @pytest.mark.parametrize("n", (2, 3, 5, 6, 10))
16: (0)             def test_margulis_gabber_galil_graph_eigvals(n):
17: (4)                 np = pytest.importorskip("numpy")
18: (4)                 sp = pytest.importorskip("scipy")
19: (4)                 g = nx.margulis_gabber_galil_graph(n)
20: (4)                 w = sorted(sp.linalg.eigvalsh(nx.adjacency_matrix(g).toarray()))
21: (4)                 assert w[-2] < 5 * np.sqrt(2)
22: (0)             @pytest.mark.parametrize("p", (3, 5, 7, 11))  # Primes
23: (0)             def test_chordal_cycle_graph(p):
24: (4)                 """Test for the :func:`networkx.chordal_cycle_graph` function."""
25: (4)                 G = nx.chordal_cycle_graph(p)
26: (4)                 assert len(G) == p
27: (0)             @pytest.mark.parametrize("p", (3, 5, 7, 11, 13))  # Primes
28: (0)             def test_paley_graph(p):
29: (4)                 """Test for the :func:`networkx.paley_graph` function."""
30: (4)                 G = nx.paley_graph(p)
31: (4)                 assert len(G) == p
32: (4)                 in_degrees = {G.in_degree(node) for node in G.nodes}
33: (4)                 out_degrees = {G.out_degree(node) for node in G.nodes}
34: (4)                 assert len(in_degrees) == 1 and in_degrees.pop() == (p - 1) // 2
35: (4)                 assert len(out_degrees) == 1 and out_degrees.pop() == (p - 1) // 2
36: (4)                 if p % 4 == 1:
37: (8)                     for u, v in G.edges:
38: (12)                        assert (v, u) in G.edges
39: (0)             @pytest.mark.parametrize("d, n", [(2, 7), (4, 10), (4, 16)])
40: (0)             def test_maybe_regular_expander(d, n):
41: (4)                 pytest.importorskip("numpy")
42: (4)                 G = nx.maybe_regular_expander(n, d)
43: (4)                 assert len(G) == n, "Should have n nodes"
44: (4)                 assert len(G.edges) == n * d / 2, "Should have n*d/2 edges"
45: (4)                 assert nx.is_k_regular(G, d), "Should be d-regular"
46: (0)             @pytest.mark.parametrize("n", (3, 5, 6, 10))
47: (0)             def test_is_regular_expander(n):
48: (4)                 pytest.importorskip("numpy")
49: (4)                 pytest.importorskip("scipy")
50: (4)                 G = nx.complete_graph(n)
51: (4)                 assert nx.is_regular_expander(G) == True, "Should be a regular expander"
52: (0)             @pytest.mark.parametrize("d, n", [(2, 7), (4, 10), (4, 16)])
53: (0)             def test_random_regular_expander(d, n):
54: (4)                 pytest.importorskip("numpy")
55: (4)                 pytest.importorskip("scipy")
56: (4)                 G = nx.random_regular_expander_graph(n, d)
57: (4)                 assert len(G) == n, "Should have n nodes"
58: (4)                 assert len(G.edges) == n * d / 2, "Should have n*d/2 edges"
59: (4)                 assert nx.is_k_regular(G, d), "Should be d-regular"
60: (4)                 assert nx.is_regular_expander(G) == True, "Should be a regular expander"
61: (0)             def test_random_regular_expander_explicit_construction():
62: (4)                 pytest.importorskip("numpy")
63: (4)                 pytest.importorskip("scipy")
64: (4)                 G = nx.random_regular_expander_graph(d=4, n=5)
65: (4)                 assert len(G) == 5 and len(G.edges) == 10, "Should be a complete graph"
66: (0)             @pytest.mark.parametrize("graph_type", (nx.Graph, nx.DiGraph,
nx.MultiDiGraph))
67: (0)             def test_margulis_gabber_galil_graph_badinput(graph_type):
68: (4)                 with pytest.raises(
69: (8)                     nx.NetworkXError, match="`create_using` must be an undirected
multigraph"
70: (4)                 ):
71: (8)                     nx.margulis_gabber_galil_graph(3, create_using=graph_type)
72: (0)             @pytest.mark.parametrize("graph_type", (nx.Graph, nx.DiGraph,
nx.MultiDiGraph))
73: (0)             def test_chordal_cycle_graph_badinput(graph_type):
74: (4)                 with pytest.raises(
75: (8)                     nx.NetworkXError, match="`create_using` must be an undirected
multigraph"
76: (4)                 ):
77: (8)                     nx.chordal_cycle_graph(3, create_using=graph_type)
78: (0)             def test_paley_graph_badinput():
```

```
79: (4)                 with pytest.raises(
80: (8)                     nx.NetworkXError, match="`create_using` cannot be a multigraph."
81: (4)                 ):
82: (8)                     nx.paley_graph(3, create_using=nx.MultiGraph)
83: (0)             def test_maybe_regular_expander_badinput():
84: (4)                 pytest.importorskip("numpy")
85: (4)                 pytest.importorskip("scipy")
86: (4)                 with pytest.raises(nx.NetworkXError, match="n must be a positive
integer"):
87: (8)                     nx.maybe_regular_expander(n=-1, d=2)
88: (4)                 with pytest.raises(nx.NetworkXError, match="d must be greater than or
equal to 2"):
89: (8)                     nx.maybe_regular_expander(n=10, d=0)
90: (4)                 with pytest.raises(nx.NetworkXError, match="Need n-1>= d to have room"):
91: (8)                     nx.maybe_regular_expander(n=5, d=6)
92: (0)             def test_is_regular_expander_badinput():
93: (4)                 pytest.importorskip("numpy")
94: (4)                 pytest.importorskip("scipy")
95: (4)                 with pytest.raises(nx.NetworkXError, match="epsilon must be non
negative"):
96: (8)                     nx.is_regular_expander(nx.Graph(), epsilon=-1)
97: (0)             def test_random_regular_expander_badinput():
98: (4)                 pytest.importorskip("numpy")
99: (4)                 pytest.importorskip("scipy")
100: (4)                with pytest.raises(nx.NetworkXError, match="n must be a positive
integer"):
101: (8)                    nx.random_regular_expander_graph(n=-1, d=2)
102: (4)                with pytest.raises(nx.NetworkXError, match="d must be greater than or
equal to 2"):
103: (8)                    nx.random_regular_expander_graph(n=10, d=0)
104: (4)                with pytest.raises(nx.NetworkXError, match="Need n-1>= d to have room"):
105: (8)                    nx.random_regular_expander_graph(n=5, d=6)
106: (4)                with pytest.raises(nx.NetworkXError, match="epsilon must be non
negative"):
107: (8)                    nx.random_regular_expander_graph(n=4, d=2, epsilon=-1)
```

File 466 - test_geometric.py:

```
1: (0)              import math
2: (0)              import random
3: (0)              from itertools import combinations
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              def l1dist(x, y):
7: (4)                  return sum(abs(a - b) for a, b in zip(x, y))
8: (0)              class TestRandomGeometricGraph:
9: (4)                  """Unit tests for :func:`~networkx.random_geometric_graph`"""
10: (4)                 def test_number_of_nodes(self):
11: (8)                     G = nx.random_geometric_graph(50, 0.25, seed=42)
12: (8)                     assert len(G) == 50
13: (8)                     G = nx.random_geometric_graph(range(50), 0.25, seed=42)
14: (8)                     assert len(G) == 50
15: (4)                 def test_distances(self):
16: (8)                     """Tests that pairs of vertices adjacent if and only if they are
17: (8)                     within the prescribed radius.
18: (8)                     """
19: (8)                     G = nx.random_geometric_graph(50, 0.25)
20: (8)                     for u, v in combinations(G, 2):
21: (12)                        if v in G[u]:
22: (16)                            assert math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
23: (12)                        else:
24: (16)                            assert not math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <=
0.25
25: (4)                 def test_p(self):
26: (8)                     """Tests for providing an alternate distance metric to the
generator."""
27: (8)                     G = nx.random_geometric_graph(50, 0.25, p=1)
```

```
 28: (8)                     for u, v in combinations(G, 2):
 29: (12)                        if v in G[u]:
 30: (16)                            assert l1dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
 31: (12)                        else:
 32: (16)                            assert not l1dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <=
 0.25
 33: (4)                 def test_node_names(self):
 34: (8)                     """Tests using values other than sequential numbers as node IDs."""
 35: (8)                     import string
 36: (8)                     nodes = list(string.ascii_lowercase)
 37: (8)                     G = nx.random_geometric_graph(nodes, 0.25)
 38: (8)                     assert len(G) == len(nodes)
 39: (8)                     for u, v in combinations(G, 2):
 40: (12)                        if v in G[u]:
 41: (16)                            assert math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
 42: (12)                        else:
 43: (16)                            assert not math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <=
 0.25
 44: (4)                 def test_pos_name(self):
 45: (8)                     G = nx.random_geometric_graph(50, 0.25, seed=42, pos_name="coords")
 46: (8)                     assert all(len(d["coords"]) == 2 for n, d in G.nodes.items())
 47: (0)             class TestSoftRandomGeometricGraph:
 48: (4)                 """Unit tests for :func:`~networkx.soft_random_geometric_graph`"""
 49: (4)                 def test_number_of_nodes(self):
 50: (8)                     G = nx.soft_random_geometric_graph(50, 0.25, seed=42)
 51: (8)                     assert len(G) == 50
 52: (8)                     G = nx.soft_random_geometric_graph(range(50), 0.25, seed=42)
 53: (8)                     assert len(G) == 50
 54: (4)                 def test_distances(self):
 55: (8)                     """Tests that pairs of vertices adjacent if and only if they are
 56: (8)                     within the prescribed radius.
 57: (8)                     """
 58: (8)                     G = nx.soft_random_geometric_graph(50, 0.25)
 59: (8)                     for u, v in combinations(G, 2):
 60: (12)                        if v in G[u]:
 61: (16)                            assert math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
 62: (4)                 def test_p(self):
 63: (8)                     """Tests for providing an alternate distance metric to the
 generator."""
 64: (8)                     def dist(x, y):
 65: (12)                        return sum(abs(a - b) for a, b in zip(x, y))
 66: (8)                     G = nx.soft_random_geometric_graph(50, 0.25, p=1)
 67: (8)                     for u, v in combinations(G, 2):
 68: (12)                        if v in G[u]:
 69: (16)                            assert dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
 70: (4)                 def test_node_names(self):
 71: (8)                     """Tests using values other than sequential numbers as node IDs."""
 72: (8)                     import string
 73: (8)                     nodes = list(string.ascii_lowercase)
 74: (8)                     G = nx.soft_random_geometric_graph(nodes, 0.25)
 75: (8)                     assert len(G) == len(nodes)
 76: (8)                     for u, v in combinations(G, 2):
 77: (12)                        if v in G[u]:
 78: (16)                            assert math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
 79: (4)                 def test_p_dist_default(self):
 80: (8)                     """Tests default p_dict = 0.5 returns graph with edge count <= RGG
 with
 81: (8)                     same n, radius, dim and positions
 82: (8)                     """
 83: (8)                     nodes = 50
 84: (8)                     dim = 2
 85: (8)                     pos = {v: [random.random() for i in range(dim)] for v in range(nodes)}
 86: (8)                     RGG = nx.random_geometric_graph(50, 0.25, pos=pos)
 87: (8)                     SRGG = nx.soft_random_geometric_graph(50, 0.25, pos=pos)
 88: (8)                     assert len(SRGG.edges()) <= len(RGG.edges())
 89: (4)                 def test_p_dist_zero(self):
 90: (8)                     """Tests if p_dict = 0 returns disconnected graph with 0 edges"""
 91: (8)                     def p_dist(dist):
 92: (12)                        return 0
```

```
93: (8)                     G = nx.soft_random_geometric_graph(50, 0.25, p_dist=p_dist)
94: (8)                     assert len(G.edges) == 0
95: (4)                 def test_pos_name(self):
96: (8)                     G = nx.soft_random_geometric_graph(50, 0.25, seed=42,
pos_name="coords")
97: (8)                     assert all(len(d["coords"]) == 2 for n, d in G.nodes.items())
98: (0)             def join(G, u, v, theta, alpha, metric):
99: (4)                 """Returns ``True`` if and only if the nodes whose attributes are
100: (4)                ``du`` and ``dv`` should be joined, according to the threshold
101: (4)                condition for geographical threshold graphs.
102: (4)                ``G`` is an undirected NetworkX graph, and ``u`` and ``v`` are nodes
103: (4)                in that graph. The nodes must have node attributes ``'pos'`` and
104: (4)                ``'weight'``.
105: (4)                ``metric`` is a distance metric.
106: (4)                """
107: (4)                du, dv = G.nodes[u], G.nodes[v]
108: (4)                u_pos, v_pos = du["pos"], dv["pos"]
109: (4)                u_weight, v_weight = du["weight"], dv["weight"]
110: (4)                return (u_weight + v_weight) * metric(u_pos, v_pos) ** alpha >= theta
111: (0)            class TestGeographicalThresholdGraph:
112: (4)                """Unit tests for :func:`~networkx.geographical_threshold_graph`"""
113: (4)                def test_number_of_nodes(self):
114: (8)                    G = nx.geographical_threshold_graph(50, 100, seed=42)
115: (8)                    assert len(G) == 50
116: (8)                    G = nx.geographical_threshold_graph(range(50), 100, seed=42)
117: (8)                    assert len(G) == 50
118: (4)                def test_distances(self):
119: (8)                    """Tests that pairs of vertices adjacent if and only if their
120: (8)                    distances meet the given threshold.
121: (8)                    """
122: (8)                    G = nx.geographical_threshold_graph(50, 10)
123: (8)                    for u, v in combinations(G, 2):
124: (12)                       if v in G[u]:
125: (16)                           assert join(G, u, v, 10, -2, math.dist)
126: (12)                       else:
127: (16)                           assert not join(G, u, v, 10, -2, math.dist)
128: (4)                def test_metric(self):
129: (8)                    """Tests for providing an alternate distance metric to the
generator."""
130: (8)                    G = nx.geographical_threshold_graph(50, 10, metric=l1dist)
131: (8)                    for u, v in combinations(G, 2):
132: (12)                       if v in G[u]:
133: (16)                           assert join(G, u, v, 10, -2, l1dist)
134: (12)                       else:
135: (16)                           assert not join(G, u, v, 10, -2, l1dist)
136: (4)                def test_p_dist_zero(self):
137: (8)                    """Tests if p_dict = 0 returns disconnected graph with 0 edges"""
138: (8)                    def p_dist(dist):
139: (12)                       return 0
140: (8)                    G = nx.geographical_threshold_graph(50, 1, p_dist=p_dist)
141: (8)                    assert len(G.edges) == 0
142: (4)                def test_pos_weight_name(self):
143: (8)                    gtg = nx.geographical_threshold_graph
144: (8)                    G = gtg(50, 100, seed=42, pos_name="coords", weight_name="wt")
145: (8)                    assert all(len(d["coords"]) == 2 for n, d in G.nodes.items())
146: (8)                    assert all(d["wt"] > 0 for n, d in G.nodes.items())
147: (0)            class TestWaxmanGraph:
148: (4)                """Unit tests for the :func:`~networkx.waxman_graph` function."""
149: (4)                def test_number_of_nodes_1(self):
150: (8)                    G = nx.waxman_graph(50, 0.5, 0.1, seed=42)
151: (8)                    assert len(G) == 50
152: (8)                    G = nx.waxman_graph(range(50), 0.5, 0.1, seed=42)
153: (8)                    assert len(G) == 50
154: (4)                def test_number_of_nodes_2(self):
155: (8)                    G = nx.waxman_graph(50, 0.5, 0.1, L=1)
156: (8)                    assert len(G) == 50
157: (8)                    G = nx.waxman_graph(range(50), 0.5, 0.1, L=1)
158: (8)                    assert len(G) == 50
159: (4)                def test_metric(self):
```

```
160: (8)                    """Tests for providing an alternate distance metric to the
generator."""
161: (8)                    G = nx.waxman_graph(50, 0.5, 0.1, metric=l1dist)
162: (8)                    assert len(G) == 50
163: (4)                def test_pos_name(self):
164: (8)                    G = nx.waxman_graph(50, 0.5, 0.1, seed=42, pos_name="coords")
165: (8)                    assert all(len(d["coords"]) == 2 for n, d in G.nodes.items())
166: (0)            class TestNavigableSmallWorldGraph:
167: (4)                def test_navigable_small_world(self):
168: (8)                    G = nx.navigable_small_world_graph(5, p=1, q=0, seed=42)
169: (8)                    gg = nx.grid_2d_graph(5, 5).to_directed()
170: (8)                    assert nx.is_isomorphic(G, gg)
171: (8)                    G = nx.navigable_small_world_graph(5, p=1, q=0, dim=3)
172: (8)                    gg = nx.grid_graph([5, 5, 5]).to_directed()
173: (8)                    assert nx.is_isomorphic(G, gg)
174: (8)                    G = nx.navigable_small_world_graph(5, p=1, q=0, dim=1)
175: (8)                    gg = nx.grid_graph([5]).to_directed()
176: (8)                    assert nx.is_isomorphic(G, gg)
177: (4)                def test_invalid_diameter_value(self):
178: (8)                    with pytest.raises(nx.NetworkXException, match=".*p must be >= 1"):
179: (12)                       nx.navigable_small_world_graph(5, p=0, q=0, dim=1)
180: (4)                def test_invalid_long_range_connections_value(self):
181: (8)                    with pytest.raises(nx.NetworkXException, match=".*q must be >= 0"):
182: (12)                       nx.navigable_small_world_graph(5, p=1, q=-1, dim=1)
183: (4)                def test_invalid_exponent_for_decaying_probability_value(self):
184: (8)                    with pytest.raises(nx.NetworkXException, match=".*r must be >= 0"):
185: (12)                       nx.navigable_small_world_graph(5, p=1, q=0, r=-1, dim=1)
186: (4)                def test_r_between_0_and_1(self):
187: (8)                    """Smoke test for radius in range [0, 1]"""
188: (8)                    G = nx.navigable_small_world_graph(3, p=1, q=0, r=0.5, dim=2, seed=42)
189: (8)                    expected = nx.grid_2d_graph(3, 3, create_using=nx.DiGraph)
190: (8)                    assert nx.utils.graphs_equal(G, expected)
191: (4)                @pytest.mark.parametrize("seed", range(2478, 2578, 10))
192: (4)                def test_r_general_scaling(self, seed):
193: (8)                    """The probability of adding a long-range edge scales with `1 /
dist**r`,
194: (8)                    so a navigable_small_world graph created with r < 1 should generally
195: (8)                    result in more edges than a navigable_small_world graph with r >= 1
196: (8)                    (for 0 < q << n).
197: (8)                    N.B. this is probabilistic, so this test may not hold for all
seeds."""
198: (8)                    G1 = nx.navigable_small_world_graph(7, q=3, r=0.5, seed=seed)
199: (8)                    G2 = nx.navigable_small_world_graph(7, q=3, r=1, seed=seed)
200: (8)                    G3 = nx.navigable_small_world_graph(7, q=3, r=2, seed=seed)
201: (8)                    assert G1.number_of_edges() > G2.number_of_edges()
202: (8)                    assert G2.number_of_edges() > G3.number_of_edges()
203: (0)            class TestThresholdedRandomGeometricGraph:
204: (4)                """Unit tests for :func:`~networkx.thresholded_random_geometric_graph`"""
205: (4)                def test_number_of_nodes(self):
206: (8)                    G = nx.thresholded_random_geometric_graph(50, 0.2, 0.1, seed=42)
207: (8)                    assert len(G) == 50
208: (8)                    G = nx.thresholded_random_geometric_graph(range(50), 0.2, 0.1,
seed=42)
209: (8)                    assert len(G) == 50
210: (4)                def test_distances(self):
211: (8)                    """Tests that pairs of vertices adjacent if and only if they are
212: (8)                    within the prescribed radius.
213: (8)                    """
214: (8)                    G = nx.thresholded_random_geometric_graph(50, 0.25, 0.1, seed=42)
215: (8)                    for u, v in combinations(G, 2):
216: (12)                       if v in G[u]:
217: (16)                           assert math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
218: (4)                def test_p(self):
219: (8)                    """Tests for providing an alternate distance metric to the
generator."""
220: (8)                    def dist(x, y):
221: (12)                       return sum(abs(a - b) for a, b in zip(x, y))
222: (8)                    G = nx.thresholded_random_geometric_graph(50, 0.25, 0.1, p=1, seed=42)
223: (8)                    for u, v in combinations(G, 2):
```

```
224: (12)                        if v in G[u]:
225: (16)                            assert dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
226: (4)                def test_node_names(self):
227: (8)                    """Tests using values other than sequential numbers as node IDs."""
228: (8)                    import string
229: (8)                    nodes = list(string.ascii_lowercase)
230: (8)                    G = nx.thresholded_random_geometric_graph(nodes, 0.25, 0.1, seed=42)
231: (8)                    assert len(G) == len(nodes)
232: (8)                    for u, v in combinations(G, 2):
233: (12)                        if v in G[u]:
234: (16)                            assert math.dist(G.nodes[u]["pos"], G.nodes[v]["pos"]) <= 0.25
235: (4)                def test_theta(self):
236: (8)                    """Tests that pairs of vertices adjacent if and only if their sum
237: (8)                    weights exceeds the threshold parameter theta.
238: (8)                    """
239: (8)                    G = nx.thresholded_random_geometric_graph(50, 0.25, 0.1, seed=42)
240: (8)                    for u, v in combinations(G, 2):
241: (12)                        if v in G[u]:
242: (16)                            assert (G.nodes[u]["weight"] + G.nodes[v]["weight"]) >= 0.1
243: (4)                def test_pos_name(self):
244: (8)                    trgg = nx.thresholded_random_geometric_graph
245: (8)                    G = trgg(50, 0.25, 0.1, seed=42, pos_name="p", weight_name="wt")
246: (8)                    assert all(len(d["p"]) == 2 for n, d in G.nodes.items())
247: (8)                    assert all(d["wt"] > 0 for n, d in G.nodes.items())
248: (0)            def test_geometric_edges_pos_attribute():
249: (4)                G = nx.Graph()
250: (4)                G.add_nodes_from(
251: (8)                    [
252: (12)                        (0, {"position": (0, 0)}),
253: (12)                        (1, {"position": (0, 1)}),
254: (12)                        (2, {"position": (1, 0)}),
255: (8)                    ]
256: (4)                )
257: (4)                expected_edges = [(0, 1), (0, 2)]
258: (4)                assert expected_edges == nx.geometric_edges(G, radius=1,
pos_name="position")
259: (0)            def test_geometric_edges_raises_no_pos():
260: (4)                G = nx.path_graph(3)
261: (4)                msg = "all nodes. must have a '"
262: (4)                with pytest.raises(nx.NetworkXError, match=msg):
263: (8)                    nx.geometric_edges(G, radius=1)
264: (0)            def test_number_of_nodes_S1():
265: (4)                G = nx.geometric_soft_configuration_graph(
266: (8)                    beta=1.5, n=100, gamma=2.7, mean_degree=10, seed=42
267: (4)                )
268: (4)                assert len(G) == 100
269: (0)            def test_set_attributes_S1():
270: (4)                G = nx.geometric_soft_configuration_graph(
271: (8)                    beta=1.5, n=100, gamma=2.7, mean_degree=10, seed=42
272: (4)                )
273: (4)                kappas = nx.get_node_attributes(G, "kappa")
274: (4)                assert len(kappas) == 100
275: (4)                thetas = nx.get_node_attributes(G, "theta")
276: (4)                assert len(thetas) == 100
277: (4)                radii = nx.get_node_attributes(G, "radius")
278: (4)                assert len(radii) == 100
279: (0)            def test_mean_kappas_mean_degree_S1():
280: (4)                G = nx.geometric_soft_configuration_graph(
281: (8)                    beta=2.5, n=50, gamma=2.7, mean_degree=10, seed=8023
282: (4)                )
283: (4)                kappas = nx.get_node_attributes(G, "kappa")
284: (4)                mean_kappas = sum(kappas.values()) / len(kappas)
285: (4)                assert math.fabs(mean_kappas - 10) < 0.5
286: (4)                degrees = dict(G.degree())
287: (4)                mean_degree = sum(degrees.values()) / len(degrees)
288: (4)                assert math.fabs(mean_degree - 10) < 1
289: (0)            def test_dict_kappas_S1():
290: (4)                kappas = {i: 10 for i in range(1000)}
291: (4)                G = nx.geometric_soft_configuration_graph(beta=1, kappas=kappas)
```

```
292: (4)                assert len(G) == 1000
293: (4)                kappas = nx.get_node_attributes(G, "kappa")
294: (4)                assert all(kappa == 10 for kappa in kappas.values())
295: (0)            def test_beta_clustering_S1():
296: (4)                G1 = nx.geometric_soft_configuration_graph(
297: (8)                    beta=1.5, n=100, gamma=3.5, mean_degree=10, seed=42
298: (4)                )
299: (4)                G2 = nx.geometric_soft_configuration_graph(
300: (8)                    beta=3.0, n=100, gamma=3.5, mean_degree=10, seed=42
301: (4)                )
302: (4)                assert nx.average_clustering(G1) < nx.average_clustering(G2)
303: (0)            def test_wrong_parameters_S1():
304: (4)                with pytest.raises(
305: (8)                    nx.NetworkXError,
306: (8)                    match="Please provide either kappas, or all 3 of: n, gamma and
mean_degree.",
307: (4)                ):
308: (8)                    G = nx.geometric_soft_configuration_graph(
309: (12)                       beta=1.5, gamma=3.5, mean_degree=10, seed=42
310: (8)                    )
311: (4)                with pytest.raises(
312: (8)                    nx.NetworkXError,
313: (8)                    match="When kappas is input, n, gamma and mean_degree must not be.",
314: (4)                ):
315: (8)                    kappas = {i: 10 for i in range(1000)}
316: (8)                    G = nx.geometric_soft_configuration_graph(
317: (12)                       beta=1.5, kappas=kappas, gamma=2.3, seed=42
318: (8)                    )
319: (4)                with pytest.raises(
320: (8)                    nx.NetworkXError,
321: (8)                    match="Please provide either kappas, or all 3 of: n, gamma and
mean_degree.",
322: (4)                ):
323: (8)                    G = nx.geometric_soft_configuration_graph(beta=1.5, seed=42)
324: (0)            def test_negative_beta_S1():
325: (4)                with pytest.raises(
326: (8)                    nx.NetworkXError, match="The parameter beta cannot be smaller or equal
to 0."
327: (4)                ):
328: (8)                    G = nx.geometric_soft_configuration_graph(
329: (12)                       beta=-1, n=100, gamma=2.3, mean_degree=10, seed=42
330: (8)                    )
331: (0)            def test_non_zero_clustering_beta_lower_one_S1():
332: (4)                G = nx.geometric_soft_configuration_graph(
333: (8)                    beta=0.5, n=100, gamma=3.5, mean_degree=10, seed=42
334: (4)                )
335: (4)                assert nx.average_clustering(G) > 0
336: (0)            def test_mean_degree_influence_on_connectivity_S1():
337: (4)                low_mean_degree = 2
338: (4)                high_mean_degree = 20
339: (4)                G_low = nx.geometric_soft_configuration_graph(
340: (8)                    beta=1.2, n=100, gamma=2.7, mean_degree=low_mean_degree, seed=42
341: (4)                )
342: (4)                G_high = nx.geometric_soft_configuration_graph(
343: (8)                    beta=1.2, n=100, gamma=2.7, mean_degree=high_mean_degree, seed=42
344: (4)                )
345: (4)                assert nx.number_connected_components(G_low) >
nx.number_connected_components(
346: (8)                    G_high
347: (4)                )
348: (0)            def test_compare_mean_kappas_different_gammas_S1():
349: (4)                G1 = nx.geometric_soft_configuration_graph(
350: (8)                    beta=1.5, n=20, gamma=2.7, mean_degree=5, seed=42
351: (4)                )
352: (4)                G2 = nx.geometric_soft_configuration_graph(
353: (8)                    beta=1.5, n=20, gamma=3.5, mean_degree=5, seed=42
354: (4)                )
355: (4)                kappas1 = nx.get_node_attributes(G1, "kappa")
356: (4)                mean_kappas1 = sum(kappas1.values()) / len(kappas1)
```

```
357: (4)                kappas2 = nx.get_node_attributes(G2, "kappa")
358: (4)                mean_kappas2 = sum(kappas2.values()) / len(kappas2)
359: (4)                assert math.fabs(mean_kappas1 - mean_kappas2) < 1
```

File 467 - test_harary_graph.py:

```
1: (0)              """Unit tests for the :mod:`networkx.generators.harary_graph` module.
2: (0)              """
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms.isomorphism.isomorph import is_isomorphic
6: (0)              from networkx.generators.harary_graph import hkn_harary_graph,
hnm_harary_graph
7: (0)              class TestHararyGraph:
8: (4)                  """
9: (4)                  Suppose n nodes, m >= n-1 edges, d = 2m // n, r = 2m % n
10: (4)                 """
11: (4)                 def test_hnm_harary_graph(self):
12: (8)                     for n, m in [(5, 5), (6, 12), (7, 14)]:
13: (12)                        G1 = hnm_harary_graph(n, m)
14: (12)                        d = 2 * m // n
15: (12)                        G2 = nx.circulant_graph(n, list(range(1, d // 2 + 1)))
16: (12)                        assert is_isomorphic(G1, G2)
17: (8)                     for n, m in [(5, 7), (6, 13), (7, 16)]:
18: (12)                        G1 = hnm_harary_graph(n, m)
19: (12)                        d = 2 * m // n
20: (12)                        G2 = nx.circulant_graph(n, list(range(1, d // 2 + 1)))
21: (12)                        assert set(G2.edges) < set(G1.edges)
22: (12)                        assert G1.number_of_edges() == m
23: (8)                     for n, m in [(6, 9), (8, 12), (10, 15)]:
24: (12)                        G1 = hnm_harary_graph(n, m)
25: (12)                        d = 2 * m // n
26: (12)                        L = list(range(1, (d + 1) // 2))
27: (12)                        L.append(n // 2)
28: (12)                        G2 = nx.circulant_graph(n, L)
29: (12)                        assert is_isomorphic(G1, G2)
30: (8)                     for n, m in [(6, 10), (8, 13), (10, 17)]:
31: (12)                        G1 = hnm_harary_graph(n, m)
32: (12)                        d = 2 * m // n
33: (12)                        L = list(range(1, (d + 1) // 2))
34: (12)                        L.append(n // 2)
35: (12)                        G2 = nx.circulant_graph(n, L)
36: (12)                        assert set(G2.edges) < set(G1.edges)
37: (12)                        assert G1.number_of_edges() == m
38: (8)                     for n, m in [(5, 4), (7, 12), (9, 14)]:
39: (12)                        G1 = hnm_harary_graph(n, m)
40: (12)                        d = 2 * m // n
41: (12)                        L = list(range(1, (d + 1) // 2))
42: (12)                        G2 = nx.circulant_graph(n, L)
43: (12)                        assert set(G2.edges) < set(G1.edges)
44: (12)                        assert G1.number_of_edges() == m
45: (8)                     n = 0
46: (8)                     m = 0
47: (8)                     pytest.raises(nx.NetworkXError, hnm_harary_graph, n, m)
48: (8)                     n = 6
49: (8)                     m = 4
50: (8)                     pytest.raises(nx.NetworkXError, hnm_harary_graph, n, m)
51: (8)                     n = 6
52: (8)                     m = 16
53: (8)                     pytest.raises(nx.NetworkXError, hnm_harary_graph, n, m)
54: (4)                 """
55: (8)                     Suppose connectivity k, number of nodes n
56: (4)                 """
57: (4)                 def test_hkn_harary_graph(self):
58: (8)                     for k, n in [(1, 6), (1, 7)]:
59: (12)                        G1 = hkn_harary_graph(k, n)
60: (12)                        G2 = nx.path_graph(n)
```

```
61: (12)                        assert is_isomorphic(G1, G2)
62: (8)                     for k, n in [(2, 6), (2, 7), (4, 6), (4, 7)]:
63: (12)                        G1 = hkn_harary_graph(k, n)
64: (12)                        G2 = nx.circulant_graph(n, list(range(1, k // 2 + 1)))
65: (12)                        assert is_isomorphic(G1, G2)
66: (8)                     for k, n in [(3, 6), (5, 8), (7, 10)]:
67: (12)                        G1 = hkn_harary_graph(k, n)
68: (12)                        L = list(range(1, (k + 1) // 2))
69: (12)                        L.append(n // 2)
70: (12)                        G2 = nx.circulant_graph(n, L)
71: (12)                        assert is_isomorphic(G1, G2)
72: (8)                     for k, n in [(3, 5), (5, 9), (7, 11)]:
73: (12)                        G1 = hkn_harary_graph(k, n)
74: (12)                        G2 = nx.circulant_graph(n, list(range(1, (k + 1) // 2)))
75: (12)                        eSet1 = set(G1.edges)
76: (12)                        eSet2 = set(G2.edges)
77: (12)                        eSet3 = set()
78: (12)                        half = n // 2
79: (12)                        for i in range(half + 1):
80: (16)                            eSet3.add((i, (i + half) % n))
81: (12)                        assert eSet1 == eSet2 | eSet3
82: (8)                     k = 0
83: (8)                     n = 0
84: (8)                     pytest.raises(nx.NetworkXError, hkn_harary_graph, k, n)
85: (8)                     k = 6
86: (8)                     n = 6
87: (8)                     pytest.raises(nx.NetworkXError, hkn_harary_graph, k, n)

----------------------------------------

File 468 - test_internet_as_graphs.py:

1: (0)              from pytest import approx
2: (0)              from networkx import is_connected, neighbors
3: (0)              from networkx.generators.internet_as_graphs import random_internet_as_graph
4: (0)              class TestInternetASTopology:
5: (4)                  @classmethod
6: (4)                  def setup_class(cls):
7: (8)                      cls.n = 1000
8: (8)                      cls.seed = 42
9: (8)                      cls.G = random_internet_as_graph(cls.n, cls.seed)
10: (8)                     cls.T = []
11: (8)                     cls.M = []
12: (8)                     cls.C = []
13: (8)                     cls.CP = []
14: (8)                     cls.customers = {}
15: (8)                     cls.providers = {}
16: (8)                     for i in cls.G.nodes():
17: (12)                        if cls.G.nodes[i]["type"] == "T":
18: (16)                            cls.T.append(i)
19: (12)                        elif cls.G.nodes[i]["type"] == "M":
20: (16)                            cls.M.append(i)
21: (12)                        elif cls.G.nodes[i]["type"] == "C":
22: (16)                            cls.C.append(i)
23: (12)                        elif cls.G.nodes[i]["type"] == "CP":
24: (16)                            cls.CP.append(i)
25: (12)                        else:
26: (16)                            raise ValueError("Inconsistent data in the graph node
attributes")
27: (12)                        cls.set_customers(i)
28: (12)                        cls.set_providers(i)
29: (4)                 @classmethod
30: (4)                 def set_customers(cls, i):
31: (8)                     if i not in cls.customers:
32: (12)                        cls.customers[i] = set()
33: (12)                        for j in neighbors(cls.G, i):
34: (16)                            e = cls.G.edges[(i, j)]
35: (16)                            if e["type"] == "transit":
36: (20)                                customer = int(e["customer"])
```

```
 37: (20)                                if j == customer:
 38: (24)                                    cls.set_customers(j)
 39: (24)                                    cls.customers[i] =
cls.customers[i].union(cls.customers[j])
 40: (24)                                    cls.customers[i].add(j)
 41: (20)                                elif i != customer:
 42: (24)                                    raise ValueError(
 43: (28)                                        "Inconsistent data in the graph edge attributes"
 44: (24)                                    )
 45: (4)                 @classmethod
 46: (4)                 def set_providers(cls, i):
 47: (8)                     if i not in cls.providers:
 48: (12)                        cls.providers[i] = set()
 49: (12)                        for j in neighbors(cls.G, i):
 50: (16)                            e = cls.G.edges[(i, j)]
 51: (16)                            if e["type"] == "transit":
 52: (20)                                customer = int(e["customer"])
 53: (20)                                if i == customer:
 54: (24)                                    cls.set_providers(j)
 55: (24)                                    cls.providers[i] =
cls.providers[i].union(cls.providers[j])
 56: (24)                                    cls.providers[i].add(j)
 57: (20)                                elif j != customer:
 58: (24)                                    raise ValueError(
 59: (28)                                        "Inconsistent data in the graph edge attributes"
 60: (24)                                    )
 61: (4)                 def test_wrong_input(self):
 62: (8)                     G = random_internet_as_graph(0)
 63: (8)                     assert len(G.nodes()) == 0
 64: (8)                     G = random_internet_as_graph(-1)
 65: (8)                     assert len(G.nodes()) == 0
 66: (8)                     G = random_internet_as_graph(1)
 67: (8)                     assert len(G.nodes()) == 1
 68: (4)                 def test_node_numbers(self):
 69: (8)                     assert len(self.G.nodes()) == self.n
 70: (8)                     assert len(self.T) < 7
 71: (8)                     assert len(self.M) == round(self.n * 0.15)
 72: (8)                     assert len(self.CP) == round(self.n * 0.05)
 73: (8)                     numb = self.n - len(self.T) - len(self.M) - len(self.CP)
 74: (8)                     assert len(self.C) == numb
 75: (4)                 def test_connectivity(self):
 76: (8)                     assert is_connected(self.G)
 77: (4)                 def test_relationships(self):
 78: (8)                     for i in self.T:
 79: (12)                        assert len(self.providers[i]) == 0
 80: (8)                     for i in self.C:
 81: (12)                        assert len(self.customers[i]) == 0
 82: (8)                     for i in self.CP:
 83: (12)                        assert len(self.customers[i]) == 0
 84: (8)                     for i in self.G.nodes():
 85: (12)                        assert len(self.customers[i].intersection(self.providers[i])) == 0
 86: (8)                     for i, j in self.G.edges():
 87: (12)                        if self.G.edges[(i, j)]["type"] == "peer":
 88: (16)                            assert j not in self.customers[i]
 89: (16)                            assert i not in self.customers[j]
 90: (16)                            assert j not in self.providers[i]
 91: (16)                            assert i not in self.providers[j]
 92: (4)                 def test_degree_values(self):
 93: (8)                     d_m = 0  # multihoming degree for M nodes
 94: (8)                     d_cp = 0  # multihoming degree for CP nodes
 95: (8)                     d_c = 0  # multihoming degree for C nodes
 96: (8)                     p_m_m = 0  # avg number of peering edges between M and M
 97: (8)                     p_cp_m = 0  # avg number of peering edges between CP and M
 98: (8)                     p_cp_cp = 0  # avg number of peering edges between CP and CP
 99: (8)                     t_m = 0  # probability M's provider is T
 100: (8)                    t_cp = 0  # probability CP's provider is T
 101: (8)                    t_c = 0  # probability C's provider is T
 102: (8)                    for i, j in self.G.edges():
 103: (12)                       e = self.G.edges[(i, j)]
```

```
104: (12)                        if e["type"] == "transit":
105: (16)                            cust = int(e["customer"])
106: (16)                            if i == cust:
107: (20)                                prov = j
108: (16)                            elif j == cust:
109: (20)                                prov = i
110: (16)                            else:
111: (20)                                raise ValueError("Inconsistent data in the graph edge
attributes")
112: (16)                            if cust in self.M:
113: (20)                                d_m += 1
114: (20)                                if self.G.nodes[prov]["type"] == "T":
115: (24)                                    t_m += 1
116: (16)                            elif cust in self.C:
117: (20)                                d_c += 1
118: (20)                                if self.G.nodes[prov]["type"] == "T":
119: (24)                                    t_c += 1
120: (16)                            elif cust in self.CP:
121: (20)                                d_cp += 1
122: (20)                                if self.G.nodes[prov]["type"] == "T":
123: (24)                                    t_cp += 1
124: (16)                            else:
125: (20)                                raise ValueError("Inconsistent data in the graph edge
attributes")
126: (12)                        elif e["type"] == "peer":
127: (16)                            if self.G.nodes[i]["type"] == "M" and self.G.nodes[j]["type"]
== "M":
128: (20)                                p_m_m += 1
129: (16)                            if self.G.nodes[i]["type"] == "CP" and self.G.nodes[j]["type"]
== "CP":
130: (20)                                p_cp_cp += 1
131: (16)                            if (
132: (20)                                self.G.nodes[i]["type"] == "M"
133: (20)                                and self.G.nodes[j]["type"] == "CP"
134: (20)                                or self.G.nodes[i]["type"] == "CP"
135: (20)                                and self.G.nodes[j]["type"] == "M"
136: (16)                            ):
137: (20)                                p_cp_m += 1
138: (12)                        else:
139: (16)                            raise ValueError("Unexpected data in the graph edge
attributes")
140: (8)                    assert d_m / len(self.M) == approx((2 + (2.5 * self.n) / 10000),
abs=1e-0)
141: (8)                    assert d_cp / len(self.CP) == approx((2 + (1.5 * self.n) / 10000),
abs=1e-0)
142: (8)                    assert d_c / len(self.C) == approx((1 + (5 * self.n) / 100000),
abs=1e-0)
143: (8)                    assert p_m_m / len(self.M) == approx((1 + (2 * self.n) / 10000),
abs=1e-0)
144: (8)                    assert p_cp_m / len(self.CP) == approx((0.2 + (2 * self.n) / 10000),
abs=1e-0)
145: (8)                    assert p_cp_cp / len(self.CP) == approx(
146: (12)                       (0.05 + (2 * self.n) / 100000), abs=1e-0
147: (8)                    )
148: (8)                    assert t_m / d_m == approx(0.375, abs=1e-1)
149: (8)                    assert t_cp / d_cp == approx(0.375, abs=1e-1)
150: (8)                    assert t_c / d_c == approx(0.125, abs=1e-1)

----------------------------------------

File 469 - test_intersection.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestIntersectionGraph:
4: (4)                  def test_random_intersection_graph(self):
5: (8)                      G = nx.uniform_random_intersection_graph(10, 5, 0.5)
6: (8)                      assert len(G) == 10
7: (4)                  def test_k_random_intersection_graph(self):
```

```
8: (8)                      G = nx.k_random_intersection_graph(10, 5, 2)
9: (8)                      assert len(G) == 10
10: (4)                 def test_k_random_intersection_graph_seeded(self):
11: (8)                     G = nx.k_random_intersection_graph(10, 5, 2, seed=1234)
12: (8)                     assert len(G) == 10
13: (4)                 def test_general_random_intersection_graph(self):
14: (8)                     G = nx.general_random_intersection_graph(10, 5, [0.1, 0.2, 0.2, 0.1,
0.1])
15: (8)                     assert len(G) == 10
16: (8)                     pytest.raises(
17: (12)                        ValueError,
18: (12)                        nx.general_random_intersection_graph,
19: (12)                        10,
20: (12)                        5,
21: (12)                        [0.1, 0.2, 0.2, 0.1],
22: (8)                     )

----------------------------------------

File 470 - test_interval_graph.py:

1: (0)              """Unit tests for the :mod:`networkx.generators.interval_graph` module.
2: (0)              """
3: (0)              import math
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx.generators.interval_graph import interval_graph
7: (0)              from networkx.utils import edges_equal
8: (0)              class TestIntervalGraph:
9: (4)                  """Unit tests for
:func:`networkx.generators.interval_graph.interval_graph`"""
10: (4)                 def test_empty(self):
11: (8)                     """Tests for trivial case of empty input"""
12: (8)                     assert len(interval_graph([])) == 0
13: (4)                 def test_interval_graph_check_invalid(self):
14: (8)                     """Tests for conditions that raise Exceptions"""
15: (8)                     invalids_having_none = [None, (1, 2)]
16: (8)                     with pytest.raises(TypeError):
17: (12)                        interval_graph(invalids_having_none)
18: (8)                     invalids_having_set = [{1, 2}]
19: (8)                     with pytest.raises(TypeError):
20: (12)                        interval_graph(invalids_having_set)
21: (8)                     invalids_having_seq_but_not_length2 = [(1, 2, 3)]
22: (8)                     with pytest.raises(TypeError):
23: (12)                        interval_graph(invalids_having_seq_but_not_length2)
24: (8)                     invalids_interval = [[3, 2]]
25: (8)                     with pytest.raises(ValueError):
26: (12)                        interval_graph(invalids_interval)
27: (4)                 def test_interval_graph_0(self):
28: (8)                     intervals = [(1, 2), (1, 3)]
29: (8)                     expected_graph = nx.Graph()
30: (8)                     expected_graph.add_edge(*intervals)
31: (8)                     actual_g = interval_graph(intervals)
32: (8)                     assert set(actual_g.nodes) == set(expected_graph.nodes)
33: (8)                     assert edges_equal(expected_graph, actual_g)
34: (4)                 def test_interval_graph_1(self):
35: (8)                     intervals = [(1, 2), (2, 3), (3, 4), (1, 4)]
36: (8)                     expected_graph = nx.Graph()
37: (8)                     expected_graph.add_nodes_from(intervals)
38: (8)                     e1 = ((1, 4), (1, 2))
39: (8)                     e2 = ((1, 4), (2, 3))
40: (8)                     e3 = ((1, 4), (3, 4))
41: (8)                     e4 = ((3, 4), (2, 3))
42: (8)                     e5 = ((1, 2), (2, 3))
43: (8)                     expected_graph.add_edges_from([e1, e2, e3, e4, e5])
44: (8)                     actual_g = interval_graph(intervals)
45: (8)                     assert set(actual_g.nodes) == set(expected_graph.nodes)
46: (8)                     assert edges_equal(expected_graph, actual_g)
47: (4)                 def test_interval_graph_2(self):
```

```
48: (8)                     intervals = [(1, 2), [3, 5], [6, 8], (9, 10)]
49: (8)                     expected_graph = nx.Graph()
50: (8)                     expected_graph.add_nodes_from([(1, 2), (3, 5), (6, 8), (9, 10)])
51: (8)                     actual_g = interval_graph(intervals)
52: (8)                     assert set(actual_g.nodes) == set(expected_graph.nodes)
53: (8)                     assert edges_equal(expected_graph, actual_g)
54: (4)                 def test_interval_graph_3(self):
55: (8)                     intervals = [(1, 4), [3, 5], [2.5, 4]]
56: (8)                     expected_graph = nx.Graph()
57: (8)                     expected_graph.add_nodes_from([(1, 4), (3, 5), (2.5, 4)])
58: (8)                     e1 = ((1, 4), (3, 5))
59: (8)                     e2 = ((1, 4), (2.5, 4))
60: (8)                     e3 = ((3, 5), (2.5, 4))
61: (8)                     expected_graph.add_edges_from([e1, e2, e3])
62: (8)                     actual_g = interval_graph(intervals)
63: (8)                     assert set(actual_g.nodes) == set(expected_graph.nodes)
64: (8)                     assert edges_equal(expected_graph, actual_g)
65: (4)                 def test_interval_graph_4(self):
66: (8)                     """test all possible overlaps"""
67: (8)                     intervals = [
68: (12)                        (0, 2),
69: (12)                        (-2, -1),
70: (12)                        (-2, 0),
71: (12)                        (-2, 1),
72: (12)                        (-2, 2),
73: (12)                        (-2, 3),
74: (12)                        (0, 1),
75: (12)                        (0, 2),
76: (12)                        (0, 3),
77: (12)                        (1, 2),
78: (12)                        (1, 3),
79: (12)                        (2, 3),
80: (12)                        (3, 4),
81: (8)                     ]
82: (8)                     expected_graph = nx.Graph()
83: (8)                     expected_graph.add_nodes_from(intervals)
84: (8)                     expected_nbrs = {
85: (12)                        (-2, 0),
86: (12)                        (-2, 1),
87: (12)                        (-2, 2),
88: (12)                        (-2, 3),
89: (12)                        (0, 1),
90: (12)                        (0, 2),
91: (12)                        (0, 3),
92: (12)                        (1, 2),
93: (12)                        (1, 3),
94: (12)                        (2, 3),
95: (8)                     }
96: (8)                     actual_g = nx.interval_graph(intervals)
97: (8)                     actual_nbrs = nx.neighbors(actual_g, (0, 2))
98: (8)                     assert set(actual_nbrs) == expected_nbrs
99: (4)                 def test_interval_graph_5(self):
100: (8)                    """this test is to see that an interval supports infinite number"""
101: (8)                    intervals = {(-math.inf, 0), (-1, -1), (0.5, 0.5), (1, 1), (1,
math.inf)}
102: (8)                    expected_graph = nx.Graph()
103: (8)                    expected_graph.add_nodes_from(intervals)
104: (8)                    e1 = ((-math.inf, 0), (-1, -1))
105: (8)                    e2 = ((1, 1), (1, math.inf))
106: (8)                    expected_graph.add_edges_from([e1, e2])
107: (8)                    actual_g = interval_graph(intervals)
108: (8)                    assert set(actual_g.nodes) == set(expected_graph.nodes)
109: (8)                    assert edges_equal(expected_graph, actual_g)

----------------------------------------

File 471 - test_lattice.py:

1: (0)              """Unit tests for the :mod:`networkx.generators.lattice` module."""
```

```
2: (0)              from itertools import product
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.utils import edges_equal
6: (0)              class TestGrid2DGraph:
7: (4)                  """Unit tests for :func:`networkx.generators.lattice.grid_2d_graph`"""
8: (4)                  def test_number_of_vertices(self):
9: (8)                      m, n = 5, 6
10: (8)                     G = nx.grid_2d_graph(m, n)
11: (8)                     assert len(G) == m * n
12: (4)                 def test_degree_distribution(self):
13: (8)                     m, n = 5, 6
14: (8)                     G = nx.grid_2d_graph(m, n)
15: (8)                     expected_histogram = [0, 0, 4, 2 * (m + n) - 8, (m - 2) * (n - 2)]
16: (8)                     assert nx.degree_histogram(G) == expected_histogram
17: (4)                 def test_directed(self):
18: (8)                     m, n = 5, 6
19: (8)                     G = nx.grid_2d_graph(m, n)
20: (8)                     H = nx.grid_2d_graph(m, n, create_using=nx.DiGraph())
21: (8)                     assert H.succ == G.adj
22: (8)                     assert H.pred == G.adj
23: (4)                 def test_multigraph(self):
24: (8)                     m, n = 5, 6
25: (8)                     G = nx.grid_2d_graph(m, n)
26: (8)                     H = nx.grid_2d_graph(m, n, create_using=nx.MultiGraph())
27: (8)                     assert list(H.edges()) == list(G.edges())
28: (4)                 def test_periodic(self):
29: (8)                     G = nx.grid_2d_graph(0, 0, periodic=True)
30: (8)                     assert dict(G.degree()) == {}
31: (8)                     for m, n, H in [
32: (12)                        (2, 2, nx.cycle_graph(4)),
33: (12)                        (1, 7, nx.cycle_graph(7)),
34: (12)                        (7, 1, nx.cycle_graph(7)),
35: (12)                        (2, 5, nx.circular_ladder_graph(5)),
36: (12)                        (5, 2, nx.circular_ladder_graph(5)),
37: (12)                        (2, 4, nx.cubical_graph()),
38: (12)                        (4, 2, nx.cubical_graph()),
39: (8)                     ]:
40: (12)                        G = nx.grid_2d_graph(m, n, periodic=True)
41: (12)                        assert nx.could_be_isomorphic(G, H)
42: (4)                 def test_periodic_iterable(self):
43: (8)                     m, n = 3, 7
44: (8)                     for a, b in product([0, 1], [0, 1]):
45: (12)                        G = nx.grid_2d_graph(m, n, periodic=(a, b))
46: (12)                        assert G.number_of_nodes() == m * n
47: (12)                        assert G.number_of_edges() == (m + a - 1) * n + (n + b - 1) * m
48: (4)                 def test_periodic_directed(self):
49: (8)                     G = nx.grid_2d_graph(4, 2, periodic=True)
50: (8)                     H = nx.grid_2d_graph(4, 2, periodic=True, create_using=nx.DiGraph())
51: (8)                     assert H.succ == G.adj
52: (8)                     assert H.pred == G.adj
53: (4)                 def test_periodic_multigraph(self):
54: (8)                     G = nx.grid_2d_graph(4, 2, periodic=True)
55: (8)                     H = nx.grid_2d_graph(4, 2, periodic=True,
create_using=nx.MultiGraph())
56: (8)                     assert list(G.edges()) == list(H.edges())
57: (4)                 def test_exceptions(self):
58: (8)                     pytest.raises(nx.NetworkXError, nx.grid_2d_graph, -3, 2)
59: (8)                     pytest.raises(nx.NetworkXError, nx.grid_2d_graph, 3, -2)
60: (8)                     pytest.raises(TypeError, nx.grid_2d_graph, 3.3, 2)
61: (8)                     pytest.raises(TypeError, nx.grid_2d_graph, 3, 2.2)
62: (4)                 def test_node_input(self):
63: (8)                     G = nx.grid_2d_graph(4, 2, periodic=True)
64: (8)                     H = nx.grid_2d_graph(range(4), range(2), periodic=True)
65: (8)                     assert nx.is_isomorphic(H, G)
66: (8)                     H = nx.grid_2d_graph("abcd", "ef", periodic=True)
67: (8)                     assert nx.is_isomorphic(H, G)
68: (8)                     G = nx.grid_2d_graph(5, 6)
69: (8)                     H = nx.grid_2d_graph(range(5), range(6))
```

```
70: (8)                     assert edges_equal(H, G)
71: (0)             class TestGridGraph:
72: (4)                 """Unit tests for :func:`networkx.generators.lattice.grid_graph`"""
73: (4)                 def test_grid_graph(self):
74: (8)                     """grid_graph([n,m]) is a connected simple graph with the
75: (8)                     following properties:
76: (8)                     number_of_nodes = n*m
77: (8)                     degree_histogram = [0,0,4,2*(n+m)-8,(n-2)*(m-2)]
78: (8)                     """
79: (8)                     for n, m in [(3, 5), (5, 3), (4, 5), (5, 4)]:
80: (12)                        dim = [n, m]
81: (12)                        g = nx.grid_graph(dim)
82: (12)                        assert len(g) == n * m
83: (12)                        assert nx.degree_histogram(g) == [
84: (16)                            0,
85: (16)                            0,
86: (16)                            4,
87: (16)                            2 * (n + m) - 8,
88: (16)                            (n - 2) * (m - 2),
89: (12)                        ]
90: (8)                     for n, m in [(1, 5), (5, 1)]:
91: (12)                        dim = [n, m]
92: (12)                        g = nx.grid_graph(dim)
93: (12)                        assert len(g) == n * m
94: (12)                        assert nx.is_isomorphic(g, nx.path_graph(5))
95: (4)                 def test_node_input(self):
96: (8)                     G = nx.grid_graph([range(7, 9), range(3, 6)])
97: (8)                     assert len(G) == 2 * 3
98: (8)                     assert nx.is_isomorphic(G, nx.grid_graph([2, 3]))
99: (4)                 def test_periodic_iterable(self):
100: (8)                    m, n, k = 3, 7, 5
101: (8)                    for a, b, c in product([0, 1], [0, 1], [0, 1]):
102: (12)                       G = nx.grid_graph([m, n, k], periodic=(a, b, c))
103: (12)                       num_e = (m + a - 1) * n * k + (n + b - 1) * m * k + (k + c - 1) *
m * n
104: (12)                       assert G.number_of_nodes() == m * n * k
105: (12)                       assert G.number_of_edges() == num_e
106: (0)            class TestHypercubeGraph:
107: (4)                """Unit tests for :func:`networkx.generators.lattice.hypercube_graph`"""
108: (4)                def test_special_cases(self):
109: (8)                    for n, H in [
110: (12)                       (0, nx.null_graph()),
111: (12)                       (1, nx.path_graph(2)),
112: (12)                       (2, nx.cycle_graph(4)),
113: (12)                       (3, nx.cubical_graph()),
114: (8)                    ]:
115: (12)                       G = nx.hypercube_graph(n)
116: (12)                       assert nx.could_be_isomorphic(G, H)
117: (4)                def test_degree_distribution(self):
118: (8)                    for n in range(1, 10):
119: (12)                       G = nx.hypercube_graph(n)
120: (12)                       expected_histogram = [0] * n + [2**n]
121: (12)                       assert nx.degree_histogram(G) == expected_histogram
122: (0)            class TestTriangularLatticeGraph:
123: (4)                "Tests for :func:`networkx.generators.lattice.triangular_lattice_graph`"
124: (4)                def test_lattice_points(self):
125: (8)                    """Tests that the graph is really a triangular lattice."""
126: (8)                    for m, n in [(2, 3), (2, 2), (2, 1), (3, 3), (3, 2), (3, 4)]:
127: (12)                       G = nx.triangular_lattice_graph(m, n)
128: (12)                       N = (n + 1) // 2
129: (12)                       assert len(G) == (m + 1) * (1 + N) - (n % 2) * ((m + 1) // 2)
130: (8)                    for i, j in G.nodes():
131: (12)                       nbrs = G[(i, j)]
132: (12)                       if i < N:
133: (16)                           assert (i + 1, j) in nbrs
134: (12)                       if j < m:
135: (16)                           assert (i, j + 1) in nbrs
136: (12)                       if j < m and (i > 0 or j % 2) and (i < N or (j + 1) % 2):
137: (16)                           assert (i + 1, j + 1) in nbrs or (i - 1, j + 1) in nbrs
```

```
    def test_directed(self):
        """Tests for creating a directed triangular lattice."""
        G = nx.triangular_lattice_graph(3, 4, create_using=nx.Graph())
        H = nx.triangular_lattice_graph(3, 4, create_using=nx.DiGraph())
        assert H.is_directed()
        for u, v in H.edges():
            assert v[1] >= u[1]
            if v[1] == u[1]:
                assert v[0] > u[0]
    def test_multigraph(self):
        """Tests for creating a triangular lattice multigraph."""
        G = nx.triangular_lattice_graph(3, 4, create_using=nx.Graph())
        H = nx.triangular_lattice_graph(3, 4, create_using=nx.MultiGraph())
        assert list(H.edges()) == list(G.edges())
    def test_periodic(self):
        G = nx.triangular_lattice_graph(4, 6, periodic=True)
        assert len(G) == 12
        assert G.size() == 36
        assert len([n for n, d in G.degree() if d != 6]) == 0
        G = nx.triangular_lattice_graph(5, 7, periodic=True)
        TLG = nx.triangular_lattice_graph
        pytest.raises(nx.NetworkXError, TLG, 2, 4, periodic=True)
        pytest.raises(nx.NetworkXError, TLG, 4, 4, periodic=True)
        pytest.raises(nx.NetworkXError, TLG, 2, 6, periodic=True)
class TestHexagonalLatticeGraph:
    "Tests for :func:`networkx.generators.lattice.hexagonal_lattice_graph`"
    def test_lattice_points(self):
        """Tests that the graph is really a hexagonal lattice."""
        for m, n in [(4, 5), (4, 4), (4, 3), (3, 2), (3, 3), (3, 5)]:
            G = nx.hexagonal_lattice_graph(m, n)
            assert len(G) == 2 * (m + 1) * (n + 1) - 2
        C_6 = nx.cycle_graph(6)
        hexagons = [
            [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)],
            [(0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4)],
            [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)],
            [(2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)],
            [(2, 2), (2, 3), (2, 4), (3, 2), (3, 3), (3, 4)],
        ]
        for hexagon in hexagons:
            assert nx.is_isomorphic(G.subgraph(hexagon), C_6)
    def test_directed(self):
        """Tests for creating a directed hexagonal lattice."""
        G = nx.hexagonal_lattice_graph(3, 5, create_using=nx.Graph())
        H = nx.hexagonal_lattice_graph(3, 5, create_using=nx.DiGraph())
        assert H.is_directed()
        pos = nx.get_node_attributes(H, "pos")
        for u, v in H.edges():
            assert pos[v][1] >= pos[u][1]
            if pos[v][1] == pos[u][1]:
                assert pos[v][0] > pos[u][0]
    def test_multigraph(self):
        """Tests for creating a hexagonal lattice multigraph."""
        G = nx.hexagonal_lattice_graph(3, 5, create_using=nx.Graph())
        H = nx.hexagonal_lattice_graph(3, 5, create_using=nx.MultiGraph())
        assert list(H.edges()) == list(G.edges())
    def test_periodic(self):
        G = nx.hexagonal_lattice_graph(4, 6, periodic=True)
        assert len(G) == 48
        assert G.size() == 72
        assert len([n for n, d in G.degree() if d != 3]) == 0
        G = nx.hexagonal_lattice_graph(5, 8, periodic=True)
        HLG = nx.hexagonal_lattice_graph
        pytest.raises(nx.NetworkXError, HLG, 2, 7, periodic=True)
        pytest.raises(nx.NetworkXError, HLG, 1, 4, periodic=True)
        pytest.raises(nx.NetworkXError, HLG, 2, 1, periodic=True)
```

File 472 - test_joint_degree_seq.py:

```
1: (0)              import time
2: (0)              from networkx.algorithms.assortativity import degree_mixing_dict
3: (0)              from networkx.generators import gnm_random_graph, powerlaw_cluster_graph
4: (0)              from networkx.generators.joint_degree_seq import (
5: (4)                  directed_joint_degree_graph,
6: (4)                  is_valid_directed_joint_degree,
7: (4)                  is_valid_joint_degree,
8: (4)                  joint_degree_graph,
9: (0)              )
10: (0)             def test_is_valid_joint_degree():
11: (4)                 """Tests for conditions that invalidate a joint degree dict"""
12: (4)                 joint_degrees = {
13: (8)                     1: {4: 1},
14: (8)                     2: {2: 2, 3: 2, 4: 2},
15: (8)                     3: {2: 2, 4: 1},
16: (8)                     4: {1: 1, 2: 2, 3: 1},
17: (4)                 }
18: (4)                 assert is_valid_joint_degree(joint_degrees)
19: (4)                 joint_degrees_1 = {
20: (8)                     1: {4: 1.5},
21: (8)                     2: {2: 2, 3: 2, 4: 2},
22: (8)                     3: {2: 2, 4: 1},
23: (8)                     4: {1: 1.5, 2: 2, 3: 1},
24: (4)                 }
25: (4)                 assert not is_valid_joint_degree(joint_degrees_1)
26: (4)                 joint_degrees_2 = {
27: (8)                     1: {4: 1},
28: (8)                     2: {2: 2, 3: 2, 4: 3},
29: (8)                     3: {2: 2, 4: 1},
30: (8)                     4: {1: 1, 2: 3, 3: 1},
31: (4)                 }
32: (4)                 assert not is_valid_joint_degree(joint_degrees_2)
33: (4)                 joint_degrees_3 = {
34: (8)                     1: {4: 2},
35: (8)                     2: {2: 2, 3: 2, 4: 2},
36: (8)                     3: {2: 2, 4: 1},
37: (8)                     4: {1: 2, 2: 2, 3: 1},
38: (4)                 }
39: (4)                 assert not is_valid_joint_degree(joint_degrees_3)
40: (4)                 joint_degrees_5 = {1: {1: 9}}
41: (4)                 assert not is_valid_joint_degree(joint_degrees_5)
42: (0)             def test_joint_degree_graph(ntimes=10):
43: (4)                 for _ in range(ntimes):
44: (8)                     seed = int(time.time())
45: (8)                     n, m, p = 20, 10, 1
46: (8)                     g = powerlaw_cluster_graph(n, m, p, seed=seed)
47: (8)                     joint_degrees_g = degree_mixing_dict(g, normalized=False)
48: (8)                     G = joint_degree_graph(joint_degrees_g)
49: (8)                     joint_degrees_G = degree_mixing_dict(G, normalized=False)
50: (8)                     assert joint_degrees_g == joint_degrees_G
51: (0)             def test_is_valid_directed_joint_degree():
52: (4)                 in_degrees = [0, 1, 1, 2]
53: (4)                 out_degrees = [1, 1, 1, 1]
54: (4)                 nkk = {1: {1: 2, 2: 2}}
55: (4)                 assert is_valid_directed_joint_degree(in_degrees, out_degrees, nkk)
56: (4)                 nkk = {1: {1: 1.5, 2: 2.5}}
57: (4)                 assert not is_valid_directed_joint_degree(in_degrees, out_degrees, nkk)
58: (4)                 nkk = {1: {1: 2, 2: 1}}
59: (4)                 assert not is_valid_directed_joint_degree(in_degrees, out_degrees, nkk)
60: (4)                 out_degrees = [1, 1, 1]
61: (4)                 nkk = {1: {1: 2, 2: 2}}
62: (4)                 assert not is_valid_directed_joint_degree(in_degrees, out_degrees, nkk)
63: (4)                 in_degrees = [0, 1, 2]
64: (4)                 assert not is_valid_directed_joint_degree(in_degrees, out_degrees, nkk)
65: (0)             def test_directed_joint_degree_graph(n=15, m=100, ntimes=1000):
66: (4)                 for _ in range(ntimes):
67: (8)                     g = gnm_random_graph(n, m, None, directed=True)
```

```
68: (8)                     in_degrees = list(dict(g.in_degree()).values())
69: (8)                     out_degrees = list(dict(g.out_degree()).values())
70: (8)                     nkk = degree_mixing_dict(g)
71: (8)                     G = directed_joint_degree_graph(in_degrees, out_degrees, nkk)
72: (8)                     assert in_degrees == list(dict(G.in_degree()).values())
73: (8)                     assert out_degrees == list(dict(G.out_degree()).values())
74: (8)                     assert nkk == degree_mixing_dict(G)
```

File 473 - test_line.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.generators import line
4: (0)              from networkx.utils import edges_equal
5: (0)              class TestGeneratorLine:
6: (4)                  def test_star(self):
7: (8)                      G = nx.star_graph(5)
8: (8)                      L = nx.line_graph(G)
9: (8)                      assert nx.is_isomorphic(L, nx.complete_graph(5))
10: (4)                 def test_path(self):
11: (8)                     G = nx.path_graph(5)
12: (8)                     L = nx.line_graph(G)
13: (8)                     assert nx.is_isomorphic(L, nx.path_graph(4))
14: (4)                 def test_cycle(self):
15: (8)                     G = nx.cycle_graph(5)
16: (8)                     L = nx.line_graph(G)
17: (8)                     assert nx.is_isomorphic(L, G)
18: (4)                 def test_digraph1(self):
19: (8)                     G = nx.DiGraph([(0, 1), (0, 2), (0, 3)])
20: (8)                     L = nx.line_graph(G)
21: (8)                     assert L.adj == {(0, 1): {}, (0, 2): {}, (0, 3): {}}
22: (4)                 def test_multigraph1(self):
23: (8)                     G = nx.MultiGraph([(0, 1), (0, 1), (1, 0), (0, 2), (2, 0), (0, 3)])
24: (8)                     L = nx.line_graph(G)
25: (8)                     assert edges_equal(
26: (12)                        L.edges(),
27: (12)                        [
28: (16)                            ((0, 3, 0), (0, 1, 0)),
29: (16)                            ((0, 3, 0), (0, 2, 0)),
30: (16)                            ((0, 3, 0), (0, 2, 1)),
31: (16)                            ((0, 3, 0), (0, 1, 1)),
32: (16)                            ((0, 3, 0), (0, 1, 2)),
33: (16)                            ((0, 1, 0), (0, 1, 1)),
34: (16)                            ((0, 1, 0), (0, 2, 0)),
35: (16)                            ((0, 1, 0), (0, 1, 2)),
36: (16)                            ((0, 1, 0), (0, 2, 1)),
37: (16)                            ((0, 1, 1), (0, 1, 2)),
38: (16)                            ((0, 1, 1), (0, 2, 0)),
39: (16)                            ((0, 1, 1), (0, 2, 1)),
40: (16)                            ((0, 1, 2), (0, 2, 0)),
41: (16)                            ((0, 1, 2), (0, 2, 1)),
42: (16)                            ((0, 2, 0), (0, 2, 1)),
43: (12)                        ],
44: (8)                     )
45: (4)                 def test_multigraph2(self):
46: (8)                     G = nx.MultiGraph([(1, 2), (2, 1)])
47: (8)                     L = nx.line_graph(G)
48: (8)                     assert edges_equal(L.edges(), [((1, 2, 0), (1, 2, 1))])
49: (4)                 def test_multidigraph1(self):
50: (8)                     G = nx.MultiDiGraph([(1, 2), (2, 1)])
51: (8)                     L = nx.line_graph(G)
52: (8)                     assert edges_equal(L.edges(), [((1, 2, 0), (2, 1, 0)), ((2, 1, 0), (1,
2, 0))])
53: (4)                 def test_multidigraph2(self):
54: (8)                     G = nx.MultiDiGraph([(0, 1), (0, 1), (0, 1), (1, 2)])
55: (8)                     L = nx.line_graph(G)
56: (8)                     assert edges_equal(
```

```
57: (12)                        L.edges(),
58: (12)                        [((0, 1, 0), (1, 2, 0)), ((0, 1, 1), (1, 2, 0)), ((0, 1, 2), (1,
2, 0))],
59: (8)                     )
60: (4)                 def test_digraph2(self):
61: (8)                     G = nx.DiGraph([(0, 1), (1, 2), (2, 3)])
62: (8)                     L = nx.line_graph(G)
63: (8)                     assert edges_equal(L.edges(), [((0, 1), (1, 2)), ((1, 2), (2, 3))])
64: (4)                 def test_create1(self):
65: (8)                     G = nx.DiGraph([(0, 1), (1, 2), (2, 3)])
66: (8)                     L = nx.line_graph(G, create_using=nx.Graph())
67: (8)                     assert edges_equal(L.edges(), [((0, 1), (1, 2)), ((1, 2), (2, 3))])
68: (4)                 def test_create2(self):
69: (8)                     G = nx.Graph([(0, 1), (1, 2), (2, 3)])
70: (8)                     L = nx.line_graph(G, create_using=nx.DiGraph())
71: (8)                     assert edges_equal(L.edges(), [((0, 1), (1, 2)), ((1, 2), (2, 3))])
72: (0)             class TestGeneratorInverseLine:
73: (4)                 def test_example(self):
74: (8)                     G = nx.Graph()
75: (8)                     G_edges = [
76: (12)                        [1, 2],
77: (12)                        [1, 3],
78: (12)                        [1, 4],
79: (12)                        [1, 5],
80: (12)                        [2, 3],
81: (12)                        [2, 5],
82: (12)                        [2, 6],
83: (12)                        [2, 7],
84: (12)                        [3, 4],
85: (12)                        [3, 5],
86: (12)                        [6, 7],
87: (12)                        [6, 8],
88: (12)                        [7, 8],
89: (8)                     ]
90: (8)                     G.add_edges_from(G_edges)
91: (8)                     H = nx.inverse_line_graph(G)
92: (8)                     solution = nx.Graph()
93: (8)                     solution_edges = [
94: (12)                        ("a", "b"),
95: (12)                        ("a", "c"),
96: (12)                        ("a", "d"),
97: (12)                        ("a", "e"),
98: (12)                        ("c", "d"),
99: (12)                        ("e", "f"),
100: (12)                       ("e", "g"),
101: (12)                       ("f", "g"),
102: (8)                    ]
103: (8)                    solution.add_edges_from(solution_edges)
104: (8)                    assert nx.is_isomorphic(H, solution)
105: (4)                def test_example_2(self):
106: (8)                    G = nx.Graph()
107: (8)                    G_edges = [[1, 2], [1, 3], [2, 3], [3, 4], [3, 5], [4, 5]]
108: (8)                    G.add_edges_from(G_edges)
109: (8)                    H = nx.inverse_line_graph(G)
110: (8)                    solution = nx.Graph()
111: (8)                    solution_edges = [("a", "c"), ("b", "c"), ("c", "d"), ("d", "e"),
("d", "f")]
112: (8)                    solution.add_edges_from(solution_edges)
113: (8)                    assert nx.is_isomorphic(H, solution)
114: (4)                def test_pair(self):
115: (8)                    G = nx.path_graph(2)
116: (8)                    H = nx.inverse_line_graph(G)
117: (8)                    solution = nx.path_graph(3)
118: (8)                    assert nx.is_isomorphic(H, solution)
119: (4)                def test_line(self):
120: (8)                    G = nx.path_graph(5)
121: (8)                    solution = nx.path_graph(6)
122: (8)                    H = nx.inverse_line_graph(G)
123: (8)                    assert nx.is_isomorphic(H, solution)
```

```
(4)     def test_triangle_graph(self):
(8)         G = nx.complete_graph(3)
(8)         H = nx.inverse_line_graph(G)
(8)         alternative_solution = nx.Graph()
(8)         alternative_solution.add_edges_from([[0, 1], [0, 2], [0, 3]])
(8)         assert nx.is_isomorphic(H, G) or nx.is_isomorphic(H,
alternative_solution)
(4)     def test_cycle(self):
(8)         G = nx.cycle_graph(5)
(8)         H = nx.inverse_line_graph(G)
(8)         assert nx.is_isomorphic(H, G)
(4)     def test_empty(self):
(8)         G = nx.Graph()
(8)         H = nx.inverse_line_graph(G)
(8)         assert nx.is_isomorphic(H, nx.complete_graph(1))
(4)     def test_K1(self):
(8)         G = nx.complete_graph(1)
(8)         H = nx.inverse_line_graph(G)
(8)         solution = nx.path_graph(2)
(8)         assert nx.is_isomorphic(H, solution)
(4)     def test_edgeless_graph(self):
(8)         G = nx.empty_graph(5)
(8)         with pytest.raises(nx.NetworkXError, match="edgeless graph"):
(12)            nx.inverse_line_graph(G)
(4)     def test_selfloops_error(self):
(8)         G = nx.cycle_graph(4)
(8)         G.add_edge(0, 0)
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(4)     def test_non_line_graphs(self):
(8)         claw = nx.star_graph(3)
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, claw)
(8)         wheel = nx.wheel_graph(6)
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, wheel)
(8)         K5m = nx.complete_graph(5)
(8)         K5m.remove_edge(0, 1)
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, K5m)
(8)         G = nx.compose(nx.path_graph(2), nx.complete_bipartite_graph(2, 3))
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(8)         G = nx.diamond_graph()
(8)         G.add_edges_from([(4, 0), (5, 3)])
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(8)         G.add_edge(4, 5)
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(8)         G = nx.diamond_graph()
(8)         G.add_edges_from([(4, 0), (4, 3)])
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(8)         G = nx.diamond_graph()
(8)         G.add_edges_from([(4, 0), (4, 1), (4, 2), (5, 3)])
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(8)         G.add_edges_from([(5, 1), (5, 2)])
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(8)         G = nx.diamond_graph()
(8)         G.add_edges_from([(4, 0), (4, 1), (5, 2), (5, 3)])
(8)         pytest.raises(nx.NetworkXError, nx.inverse_line_graph, G)
(4)     def test_wrong_graph_type(self):
(8)         G = nx.DiGraph()
(8)         G_edges = [[0, 1], [0, 2], [0, 3]]
(8)         G.add_edges_from(G_edges)
(8)         pytest.raises(nx.NetworkXNotImplemented, nx.inverse_line_graph, G)
(8)         G = nx.MultiGraph()
(8)         G_edges = [[0, 1], [0, 2], [0, 3]]
(8)         G.add_edges_from(G_edges)
(8)         pytest.raises(nx.NetworkXNotImplemented, nx.inverse_line_graph, G)
(4)     def test_line_inverse_line_complete(self):
(8)         G = nx.complete_graph(10)
(8)         H = nx.line_graph(G)
(8)         J = nx.inverse_line_graph(H)
(8)         assert nx.is_isomorphic(G, J)
(4)     def test_line_inverse_line_path(self):
```

```
192: (8)                    G = nx.path_graph(10)
193: (8)                    H = nx.line_graph(G)
194: (8)                    J = nx.inverse_line_graph(H)
195: (8)                    assert nx.is_isomorphic(G, J)
196: (4)                def test_line_inverse_line_hypercube(self):
197: (8)                    G = nx.hypercube_graph(5)
198: (8)                    H = nx.line_graph(G)
199: (8)                    J = nx.inverse_line_graph(H)
200: (8)                    assert nx.is_isomorphic(G, J)
201: (4)                def test_line_inverse_line_cycle(self):
202: (8)                    G = nx.cycle_graph(10)
203: (8)                    H = nx.line_graph(G)
204: (8)                    J = nx.inverse_line_graph(H)
205: (8)                    assert nx.is_isomorphic(G, J)
206: (4)                def test_line_inverse_line_star(self):
207: (8)                    G = nx.star_graph(20)
208: (8)                    H = nx.line_graph(G)
209: (8)                    J = nx.inverse_line_graph(H)
210: (8)                    assert nx.is_isomorphic(G, J)
211: (4)                def test_line_inverse_line_multipartite(self):
212: (8)                    G = nx.complete_multipartite_graph(3, 4, 5)
213: (8)                    H = nx.line_graph(G)
214: (8)                    J = nx.inverse_line_graph(H)
215: (8)                    assert nx.is_isomorphic(G, J)
216: (4)                def test_line_inverse_line_dgm(self):
217: (8)                    G = nx.dorogovtsev_goltsev_mendes_graph(4)
218: (8)                    H = nx.line_graph(G)
219: (8)                    J = nx.inverse_line_graph(H)
220: (8)                    assert nx.is_isomorphic(G, J)
221: (4)                def test_line_different_node_types(self):
222: (8)                    G = nx.path_graph([1, 2, 3, "a", "b", "c"])
223: (8)                    H = nx.line_graph(G)
224: (8)                    J = nx.inverse_line_graph(H)
225: (8)                    assert nx.is_isomorphic(G, J)
226: (0)            class TestGeneratorPrivateFunctions:
227: (4)                def test_triangles_error(self):
228: (8)                    G = nx.diamond_graph()
229: (8)                    pytest.raises(nx.NetworkXError, line._triangles, G, (4, 0))
230: (8)                    pytest.raises(nx.NetworkXError, line._triangles, G, (0, 3))
231: (4)                def test_odd_triangles_error(self):
232: (8)                    G = nx.diamond_graph()
233: (8)                    pytest.raises(nx.NetworkXError, line._odd_triangle, G, (0, 1, 4))
234: (8)                    pytest.raises(nx.NetworkXError, line._odd_triangle, G, (0, 1, 3))
235: (4)                def test_select_starting_cell_error(self):
236: (8)                    G = nx.diamond_graph()
237: (8)                    pytest.raises(nx.NetworkXError, line._select_starting_cell, G, (4, 0))
238: (8)                    pytest.raises(nx.NetworkXError, line._select_starting_cell, G, (0, 3))
239: (4)                def test_diamond_graph(self):
240: (8)                    G = nx.diamond_graph()
241: (8)                    for edge in G.edges:
242: (12)                       cell = line._select_starting_cell(G, starting_edge=edge)
243: (12)                       assert len(cell) == 3
244: (12)                       assert all(v in G[u] for u in cell for v in cell if u != v)
```

File 474 - test_mycielski.py:

```
1: (0)              """Unit tests for the :mod:`networkx.generators.mycielski` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestMycielski:
5: (4)                  def test_construction(self):
6: (8)                      G = nx.path_graph(2)
7: (8)                      M = nx.mycielskian(G)
8: (8)                      assert nx.is_isomorphic(M, nx.cycle_graph(5))
9: (4)                  def test_size(self):
10: (8)                     G = nx.path_graph(2)
11: (8)                     M = nx.mycielskian(G, 2)
```

```
12: (8)                     assert len(M) == 11
13: (8)                     assert M.size() == 20
14: (4)                 def test_mycielski_graph_generator(self):
15: (8)                     G = nx.mycielski_graph(1)
16: (8)                     assert nx.is_isomorphic(G, nx.empty_graph(1))
17: (8)                     G = nx.mycielski_graph(2)
18: (8)                     assert nx.is_isomorphic(G, nx.path_graph(2))
19: (8)                     G = nx.mycielski_graph(3)
20: (8)                     assert nx.is_isomorphic(G, nx.cycle_graph(5))
21: (8)                     G = nx.mycielski_graph(4)
22: (8)                     assert nx.is_isomorphic(G, nx.mycielskian(nx.cycle_graph(5)))
23: (8)                     with pytest.raises(nx.NetworkXError, match="must satisfy n >= 1"):
24: (12)                        nx.mycielski_graph(0)
```

File 475 - test_nonisomorphic_trees.py:

```
1: (0)              """
2: (0)              Unit tests for WROM algorithm generator in generators/nonisomorphic_trees.py
3: (0)              """
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              from networkx.utils import edges_equal
7: (0)              class TestGeneratorNonIsomorphicTrees:
8: (4)                  def test_tree_structure(self):
9: (8)                      def f(x):
10: (12)                        return list(nx.nonisomorphic_trees(x))
11: (8)                     for i in f(6):
12: (12)                        assert nx.is_tree(i)
13: (8)                     for i in f(8):
14: (12)                        assert nx.is_tree(i)
15: (4)                 def test_nonisomorphism(self):
16: (8)                     def f(x):
17: (12)                        return list(nx.nonisomorphic_trees(x))
18: (8)                     trees = f(6)
19: (8)                     for i in range(len(trees)):
20: (12)                        for j in range(i + 1, len(trees)):
21: (16)                            assert not nx.is_isomorphic(trees[i], trees[j])
22: (8)                     trees = f(8)
23: (8)                     for i in range(len(trees)):
24: (12)                        for j in range(i + 1, len(trees)):
25: (16)                            assert not nx.is_isomorphic(trees[i], trees[j])
26: (4)                 def test_number_of_nonisomorphic_trees(self):
27: (8)                     assert nx.number_of_nonisomorphic_trees(2) == 1
28: (8)                     assert nx.number_of_nonisomorphic_trees(3) == 1
29: (8)                     assert nx.number_of_nonisomorphic_trees(4) == 2
30: (8)                     assert nx.number_of_nonisomorphic_trees(5) == 3
31: (8)                     assert nx.number_of_nonisomorphic_trees(6) == 6
32: (8)                     assert nx.number_of_nonisomorphic_trees(7) == 11
33: (8)                     assert nx.number_of_nonisomorphic_trees(8) == 23
34: (4)                 def test_nonisomorphic_trees(self):
35: (8)                     def f(x):
36: (12)                        return list(nx.nonisomorphic_trees(x))
37: (8)                     assert edges_equal(f(3)[0].edges(), [(0, 1), (0, 2)])
38: (8)                     assert edges_equal(f(4)[0].edges(), [(0, 1), (0, 3), (1, 2)])
39: (8)                     assert edges_equal(f(4)[1].edges(), [(0, 1), (0, 2), (0, 3)])
40: (4)                 def test_nonisomorphic_trees_matrix(self):
41: (8)                     trees_2 = [[[0, 1], [1, 0]]]
42: (8)                     with pytest.deprecated_call():
43: (12)                        assert list(nx.nonisomorphic_trees(2, create="matrix")) == trees_2
44: (8)                     trees_3 = [[[0, 1, 1], [1, 0, 0], [1, 0, 0]]]
45: (8)                     with pytest.deprecated_call():
46: (12)                        assert list(nx.nonisomorphic_trees(3, create="matrix")) == trees_3
47: (8)                     trees_4 = [
48: (12)                        [[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]],
49: (12)                        [[0, 1, 1, 1], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]],
50: (8)                     ]
51: (8)                     with pytest.deprecated_call():
```

```
52: (12)                        assert list(nx.nonisomorphic_trees(4, create="matrix")) == trees_4

----------------------------------------

File 476 - test_random_clustered.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              class TestRandomClusteredGraph:
4: (4)                  def test_custom_joint_degree_sequence(self):
5: (8)                      node = [1, 1, 1, 2, 1, 2, 0, 0]
6: (8)                      tri = [0, 0, 0, 0, 0, 1, 1, 1]
7: (8)                      joint_degree_sequence = zip(node, tri)
8: (8)                      G = nx.random_clustered_graph(joint_degree_sequence)
9: (8)                      assert G.number_of_nodes() == 8
10: (8)                     assert G.number_of_edges() == 7
11: (4)                 def test_tuple_joint_degree_sequence(self):
12: (8)                     G = nx.random_clustered_graph([(1, 2), (2, 1), (1, 1), (1, 1), (1, 1),
(2, 0)])
13: (8)                     assert G.number_of_nodes() == 6
14: (8)                     assert G.number_of_edges() == 10
15: (4)                 def test_invalid_joint_degree_sequence_type(self):
16: (8)                     with pytest.raises(nx.NetworkXError, match="Invalid degree sequence"):
17: (12)                        nx.random_clustered_graph([[1, 1], [2, 1], [0, 1]])
18: (4)                 def test_invalid_joint_degree_sequence_value(self):
19: (8)                     with pytest.raises(nx.NetworkXError, match="Invalid degree sequence"):
20: (12)                        nx.random_clustered_graph([[1, 1], [1, 2], [0, 1]])
21: (4)                 def test_directed_graph_raises_error(self):
22: (8)                     with pytest.raises(nx.NetworkXError, match="Directed Graph not
supported"):
23: (12)                        nx.random_clustered_graph(
24: (16)                            [(1, 2), (2, 1), (1, 1), (1, 1), (1, 1), (2, 0)],
25: (16)                            create_using=nx.DiGraph,
26: (12)                        )

----------------------------------------

File 477 - test_random_graphs.py:

1: (0)              """Unit tests for the :mod:`networkx.generators.random_graphs` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              _gnp_generators = [
5: (4)                  nx.gnp_random_graph,
6: (4)                  nx.fast_gnp_random_graph,
7: (4)                  nx.binomial_graph,
8: (4)                  nx.erdos_renyi_graph,
9: (0)              ]
10: (0)             @pytest.mark.parametrize("generator", _gnp_generators)
11: (0)             @pytest.mark.parametrize("directed", (True, False))
12: (0)             def test_gnp_generators_negative_edge_probability(generator, directed):
13: (4)                 """If the edge probability `p` is <=0, the resulting graph should have no
edges."""
14: (4)                 G = generator(10, -1.1, directed=directed)
15: (4)                 assert len(G) == 10
16: (4)                 assert G.number_of_edges() == 0
17: (4)                 assert G.is_directed() == directed
18: (0)             @pytest.mark.parametrize("generator", _gnp_generators)
19: (0)             @pytest.mark.parametrize(
20: (4)                 ("directed", "expected_num_edges"),
21: (4)                 [(False, 45), (True, 90)],
22: (0)             )
23: (0)             def test_gnp_generators_greater_than_1_edge_probability(
24: (4)                 generator, directed, expected_num_edges
25: (0)             ):
26: (4)                 """If the edge probability `p` is >=1, the resulting graph should be
complete."""
27: (4)                 G = generator(10, 1.1, directed=directed)
28: (4)                 assert len(G) == 10
```

```
29: (4)                 assert G.number_of_edges() == expected_num_edges
30: (4)                 assert G.is_directed() == directed
31: (0)             @pytest.mark.parametrize("generator", _gnp_generators)
32: (0)             @pytest.mark.parametrize("directed", (True, False))
33: (0)             def test_gnp_generators_basic(generator, directed):
34: (4)                 """If the edge probability `p` is >0 and <1, test only the basic
properties."""
35: (4)                 G = generator(10, 0.1, directed=directed)
36: (4)                 assert len(G) == 10
37: (4)                 assert G.is_directed() == directed
38: (0)             @pytest.mark.parametrize("generator", _gnp_generators)
39: (0)             def test_gnp_generators_for_p_close_to_1(generator):
40: (4)                 """If the edge probability `p` is close to 1, the resulting graph should
have all edges."""
41: (4)                 runs = 100
42: (4)                 edges = sum(
43: (8)                     generator(10, 0.99999, directed=True).number_of_edges() for _ in
range(runs)
44: (4)                 )
45: (4)                 assert abs(edges / float(runs) - 90) <= runs * 2.0 / 100
46: (0)             @pytest.mark.parametrize("generator", _gnp_generators)
47: (0)             @pytest.mark.parametrize("p", (0.2, 0.8))
48: (0)             @pytest.mark.parametrize("directed", (True, False))
49: (0)             def test_gnp_generators_edge_probability(generator, p, directed):
50: (4)                 """Test that gnp generators generate edges according to the their
probability `p`."""
51: (4)                 runs = 5000
52: (4)                 n = 5
53: (4)                 edge_counts = [[0] * n for _ in range(n)]
54: (4)                 for i in range(runs):
55: (8)                     G = generator(n, p, directed=directed)
56: (8)                     for v, w in G.edges:
57: (12)                        edge_counts[v][w] += 1
58: (12)                        if not directed:
59: (16)                            edge_counts[w][v] += 1
60: (4)                 for v in range(n):
61: (8)                     for w in range(n):
62: (12)                        if v == w:
63: (16)                            assert edge_counts[v][w] == 0
64: (12)                        else:
65: (16)                            assert abs(edge_counts[v][w] / float(runs) - p) <= 0.03
66: (0)             @pytest.mark.parametrize(
67: (4)                 "generator", [nx.gnp_random_graph, nx.binomial_graph,
nx.erdos_renyi_graph]
68: (0)             )
69: (0)             @pytest.mark.parametrize(
70: (4)                 ("seed", "directed", "expected_num_edges"),
71: (4)                 [(42, False, 1219), (42, True, 2454), (314, False, 1247), (314, True,
2476)],
72: (0)             )
73: (0)             def test_gnp_random_graph_aliases(generator, seed, directed,
expected_num_edges):
74: (4)                 """Test that aliases give the same result with the same seed."""
75: (4)                 G = generator(100, 0.25, seed=seed, directed=directed)
76: (4)                 assert len(G) == 100
77: (4)                 assert G.number_of_edges() == expected_num_edges
78: (4)                 assert G.is_directed() == directed
79: (0)             class TestGeneratorsRandom:
80: (4)                 def test_random_graph(self):
81: (8)                     seed = 42
82: (8)                     G = nx.gnm_random_graph(100, 20, seed)
83: (8)                     G = nx.gnm_random_graph(100, 20, seed, directed=True)
84: (8)                     G = nx.dense_gnm_random_graph(100, 20, seed)
85: (8)                     G = nx.barabasi_albert_graph(100, 1, seed)
86: (8)                     G = nx.barabasi_albert_graph(100, 3, seed)
87: (8)                     assert G.number_of_edges() == (97 * 3)
88: (8)                     G = nx.barabasi_albert_graph(100, 3, seed, nx.complete_graph(5))
89: (8)                     assert G.number_of_edges() == (10 + 95 * 3)
90: (8)                     G = nx.extended_barabasi_albert_graph(100, 1, 0, 0, seed)
```

```
 91: (8)                    assert G.number_of_edges() == 99
 92: (8)                    G = nx.extended_barabasi_albert_graph(100, 3, 0, 0, seed)
 93: (8)                    assert G.number_of_edges() == 97 * 3
 94: (8)                    G = nx.extended_barabasi_albert_graph(100, 1, 0, 0.5, seed)
 95: (8)                    assert G.number_of_edges() == 99
 96: (8)                    G = nx.extended_barabasi_albert_graph(100, 2, 0.5, 0, seed)
 97: (8)                    assert G.number_of_edges() > 100 * 3
 98: (8)                    assert G.number_of_edges() < 100 * 4
 99: (8)                    G = nx.extended_barabasi_albert_graph(100, 2, 0.3, 0.3, seed)
 100: (8)                   assert G.number_of_edges() > 100 * 2
 101: (8)                   assert G.number_of_edges() < 100 * 4
 102: (8)                   G = nx.powerlaw_cluster_graph(100, 1, 1.0, seed)
 103: (8)                   G = nx.powerlaw_cluster_graph(100, 3, 0.0, seed)
 104: (8)                   assert G.number_of_edges() == (97 * 3)
 105: (8)                   G = nx.random_regular_graph(10, 20, seed)
 106: (8)                   pytest.raises(nx.NetworkXError, nx.random_regular_graph, 3, 21)
 107: (8)                   pytest.raises(nx.NetworkXError, nx.random_regular_graph, 33, 21)
 108: (8)                   constructor = [(10, 20, 0.8), (20, 40, 0.8)]
 109: (8)                   G = nx.random_shell_graph(constructor, seed)
 110: (8)                   def is_caterpillar(g):
 111: (12)                      """
 112: (12)                      A tree is a caterpillar iff all nodes of degree >=3 are surrounded
 113: (12)                      by at most two nodes of degree two or greater.
 114: (12)                      ref: http://mathworld.wolfram.com/CaterpillarGraph.html
 115: (12)                      """
 116: (12)                      deg_over_3 = [n for n in g if g.degree(n) >= 3]
 117: (12)                      for n in deg_over_3:
 118: (16)                          nbh_deg_over_2 = [nbh for nbh in g.neighbors(n) if
g.degree(nbh) >= 2]
 119: (16)                          if not len(nbh_deg_over_2) <= 2:
 120: (20)                              return False
 121: (12)                      return True
 122: (8)                   def is_lobster(g):
 123: (12)                      """
 124: (12)                      A tree is a lobster if it has the property that the removal of
leaf
 125: (12)                      nodes leaves a caterpillar graph (Gallian 2007)
 126: (12)                      ref: http://mathworld.wolfram.com/LobsterGraph.html
 127: (12)                      """
 128: (12)                      non_leafs = [n for n in g if g.degree(n) > 1]
 129: (12)                      return is_caterpillar(g.subgraph(non_leafs))
 130: (8)                   G = nx.random_lobster(10, 0.1, 0.5, seed)
 131: (8)                   assert max(G.degree(n) for n in G.nodes()) > 3
 132: (8)                   assert is_lobster(G)
 133: (8)                   pytest.raises(nx.NetworkXError, nx.random_lobster, 10, 0.1, 1, seed)
 134: (8)                   pytest.raises(nx.NetworkXError, nx.random_lobster, 10, 1, 1, seed)
 135: (8)                   pytest.raises(nx.NetworkXError, nx.random_lobster, 10, 1, 0.5, seed)
 136: (8)                   G = nx.random_lobster(10, 0.1, 0.0, seed)
 137: (8)                   assert is_caterpillar(G)
 138: (8)                   seq = nx.random_powerlaw_tree_sequence(10, 3, seed=14, tries=1)
 139: (8)                   G = nx.random_powerlaw_tree(10, 3, seed=14, tries=1)
 140: (4)               def test_dual_barabasi_albert(self, m1=1, m2=4, p=0.5):
 141: (8)                   """
 142: (8)                   Tests that the dual BA random graph generated behaves consistently.
 143: (8)                   Tests the exceptions are raised as expected.
 144: (8)                   The graphs generation are repeated several times to prevent lucky
shots
 145: (8)                   """
 146: (8)                   seeds = [42, 314, 2718]
 147: (8)                   initial_graph = nx.complete_graph(10)
 148: (8)                   for seed in seeds:
 149: (12)                      BA1 = nx.barabasi_albert_graph(100, m1, seed)
 150: (12)                      DBA1 = nx.dual_barabasi_albert_graph(100, m1, m2, 1, seed)
 151: (12)                      assert BA1.edges() == DBA1.edges()
 152: (12)                      BA2 = nx.barabasi_albert_graph(100, m2, seed)
 153: (12)                      DBA2 = nx.dual_barabasi_albert_graph(100, m1, m2, 0, seed)
 154: (12)                      assert BA2.edges() == DBA2.edges()
 155: (12)                      BA3 = nx.barabasi_albert_graph(100, m1, seed)
 156: (12)                      DBA3 = nx.dual_barabasi_albert_graph(100, m1, m1, p, seed)
```

```
157: (12)                        assert BA3.size() == DBA3.size()
158: (12)                        DBA = nx.dual_barabasi_albert_graph(100, m1, m2, p, seed,
initial_graph)
159: (12)                        BA1 = nx.barabasi_albert_graph(100, m1, seed, initial_graph)
160: (12)                        BA2 = nx.barabasi_albert_graph(100, m2, seed, initial_graph)
161: (12)                        assert (
162: (16)                            min(BA1.size(), BA2.size()) <= DBA.size() <= max(BA1.size(),
BA2.size())
163: (12)                        )
164: (8)                    dbag = nx.dual_barabasi_albert_graph
165: (8)                    pytest.raises(nx.NetworkXError, dbag, m1, m1, m2, 0)
166: (8)                    pytest.raises(nx.NetworkXError, dbag, m2, m1, m2, 0)
167: (8)                    pytest.raises(nx.NetworkXError, dbag, 100, m1, m2, -0.5)
168: (8)                    pytest.raises(nx.NetworkXError, dbag, 100, m1, m2, 1.5)
169: (8)                    initial = nx.complete_graph(max(m1, m2) - 1)
170: (8)                    pytest.raises(nx.NetworkXError, dbag, 100, m1, m2, p,
initial_graph=initial)
171: (4)                def test_extended_barabasi_albert(self, m=2):
172: (8)                    """
173: (8)                    Tests that the extended BA random graph generated behaves
consistently.
174: (8)                    Tests the exceptions are raised as expected.
175: (8)                    The graphs generation are repeated several times to prevent lucky-
shots
176: (8)                    """
177: (8)                    seeds = [42, 314, 2718]
178: (8)                    for seed in seeds:
179: (12)                       BA_model = nx.barabasi_albert_graph(100, m, seed)
180: (12)                       BA_model_edges = BA_model.number_of_edges()
181: (12)                       G1 = nx.extended_barabasi_albert_graph(100, m, 0, 0, seed)
182: (12)                       assert G1.size() == BA_model_edges
183: (12)                       G1 = nx.extended_barabasi_albert_graph(100, m, 0.8, 0, seed)
184: (12)                       assert G1.size() > BA_model_edges * 2
185: (12)                       G2 = nx.extended_barabasi_albert_graph(100, m, 0, 0.8, seed)
186: (12)                       assert G2.size() == BA_model_edges
187: (12)                       G3 = nx.extended_barabasi_albert_graph(100, m, 0.3, 0.3, seed)
188: (12)                       assert G3.size() > G2.size()
189: (12)                       assert G3.size() < G1.size()
190: (8)                    ebag = nx.extended_barabasi_albert_graph
191: (8)                    pytest.raises(nx.NetworkXError, ebag, m, m, 0, 0)
192: (8)                    pytest.raises(nx.NetworkXError, ebag, 1, 0.5, 0, 0)
193: (8)                    pytest.raises(nx.NetworkXError, ebag, 100, 2, 0.5, 0.5)
194: (4)                def test_random_zero_regular_graph(self):
195: (8)                    """Tests that a 0-regular graph has the correct number of nodes and
196: (8)                    edges.
197: (8)                    """
198: (8)                    seed = 42
199: (8)                    G = nx.random_regular_graph(0, 10, seed)
200: (8)                    assert len(G) == 10
201: (8)                    assert G.number_of_edges() == 0
202: (4)                def test_gnm(self):
203: (8)                    G = nx.gnm_random_graph(10, 3)
204: (8)                    assert len(G) == 10
205: (8)                    assert G.number_of_edges() == 3
206: (8)                    G = nx.gnm_random_graph(10, 3, seed=42)
207: (8)                    assert len(G) == 10
208: (8)                    assert G.number_of_edges() == 3
209: (8)                    G = nx.gnm_random_graph(10, 100)
210: (8)                    assert len(G) == 10
211: (8)                    assert G.number_of_edges() == 45
212: (8)                    G = nx.gnm_random_graph(10, 100, directed=True)
213: (8)                    assert len(G) == 10
214: (8)                    assert G.number_of_edges() == 90
215: (8)                    G = nx.gnm_random_graph(10, -1.1)
216: (8)                    assert len(G) == 10
217: (8)                    assert G.number_of_edges() == 0
218: (4)                def test_watts_strogatz_big_k(self):
219: (8)                    pytest.raises(nx.NetworkXError, nx.watts_strogatz_graph, 10, 11, 0.25)
220: (8)                    pytest.raises(nx.NetworkXError, nx.newman_watts_strogatz_graph, 10,
```

```
11, 0.25)
221: (8)                    nx.watts_strogatz_graph(10, 9, 0.25, seed=0)
222: (8)                    nx.newman_watts_strogatz_graph(10, 9, 0.5, seed=0)
223: (8)                    nx.watts_strogatz_graph(10, 10, 0.25, seed=0)
224: (8)                    nx.newman_watts_strogatz_graph(10, 10, 0.25, seed=0)
225: (4)                def test_random_kernel_graph(self):
226: (8)                    def integral(u, w, z):
227: (12)                       return c * (z - w)
228: (8)                    def root(u, w, r):
229: (12)                       return r / c + w
230: (8)                    c = 1
231: (8)                    graph = nx.random_kernel_graph(1000, integral, root)
232: (8)                    graph = nx.random_kernel_graph(1000, integral, root, seed=42)
233: (8)                    assert len(graph) == 1000
234: (0)            @pytest.mark.parametrize(
235: (4)                ("k", "expected_num_nodes", "expected_num_edges"),
236: (4)                [
237: (8)                    (2, 10, 10),
238: (8)                    (4, 10, 20),
239: (4)                ],
240: (0)            )
241: (0)            def test_watts_strogatz(k, expected_num_nodes, expected_num_edges):
242: (4)                G = nx.watts_strogatz_graph(10, k, 0.25, seed=42)
243: (4)                assert len(G) == expected_num_nodes
244: (4)                assert G.number_of_edges() == expected_num_edges
245: (0)            def test_newman_watts_strogatz_zero_probability():
246: (4)                G = nx.newman_watts_strogatz_graph(10, 2, 0.0, seed=42)
247: (4)                assert len(G) == 10
248: (4)                assert G.number_of_edges() == 10
249: (0)            def test_newman_watts_strogatz_nonzero_probability():
250: (4)                G = nx.newman_watts_strogatz_graph(10, 4, 0.25, seed=42)
251: (4)                assert len(G) == 10
252: (4)                assert G.number_of_edges() >= 20
253: (0)            def test_connected_watts_strogatz():
254: (4)                G = nx.connected_watts_strogatz_graph(10, 2, 0.1, tries=10, seed=42)
255: (4)                assert len(G) == 10
256: (4)                assert G.number_of_edges() == 10
257: (0)            def test_connected_watts_strogatz_zero_tries():
258: (4)                with pytest.raises(nx.NetworkXError, match="Maximum number of tries
exceeded"):
259: (8)                    nx.connected_watts_strogatz_graph(10, 2, 0.1, tries=0)

----------------------------------------

File 478 - test_small.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.algorithms.isomorphism.isomorph import graph_could_be_isomorphic
4: (0)              is_isomorphic = graph_could_be_isomorphic
5: (0)              """Generators - Small
6: (0)              ====================
7: (0)              Some small graphs
8: (0)              """
9: (0)              null = nx.null_graph()
10: (0)             class TestGeneratorsSmall:
11: (4)                 def test__LCF_graph(self):
12: (8)                     G = nx.LCF_graph(-10, [1, 2], 100)
13: (8)                     assert is_isomorphic(G, null)
14: (8)                     G = nx.LCF_graph(0, [1, 2], 3)
15: (8)                     assert is_isomorphic(G, null)
16: (8)                     G = nx.LCF_graph(0, [1, 2], 10)
17: (8)                     assert is_isomorphic(G, null)
18: (8)                     for a, b, c in [(5, [], 0), (10, [], 0), (5, [], 1), (10, [], 10)]:
19: (12)                        G = nx.LCF_graph(a, b, c)
20: (12)                        assert is_isomorphic(G, nx.cycle_graph(a))
21: (8)                     G = nx.LCF_graph(6, [3, -3], 3)
22: (8)                     utility_graph = nx.complete_bipartite_graph(3, 3)
23: (8)                     assert is_isomorphic(G, utility_graph)
```

```
(8)                     with pytest.raises(nx.NetworkXError, match="Directed Graph not
supported"):
(12)                        G = nx.LCF_graph(6, [3, -3], 3, create_using=nx.DiGraph)
(4)                 def test_properties_named_small_graphs(self):
(8)                     G = nx.bull_graph()
(8)                     assert sorted(G) == list(range(5))
(8)                     assert G.number_of_edges() == 5
(8)                     assert sorted(d for n, d in G.degree()) == [1, 1, 2, 3, 3]
(8)                     assert nx.diameter(G) == 3
(8)                     assert nx.radius(G) == 2
(8)                     G = nx.chvatal_graph()
(8)                     assert sorted(G) == list(range(12))
(8)                     assert G.number_of_edges() == 24
(8)                     assert [d for n, d in G.degree()] == 12 * [4]
(8)                     assert nx.diameter(G) == 2
(8)                     assert nx.radius(G) == 2
(8)                     G = nx.cubical_graph()
(8)                     assert sorted(G) == list(range(8))
(8)                     assert G.number_of_edges() == 12
(8)                     assert [d for n, d in G.degree()] == 8 * [3]
(8)                     assert nx.diameter(G) == 3
(8)                     assert nx.radius(G) == 3
(8)                     G = nx.desargues_graph()
(8)                     assert sorted(G) == list(range(20))
(8)                     assert G.number_of_edges() == 30
(8)                     assert [d for n, d in G.degree()] == 20 * [3]
(8)                     G = nx.diamond_graph()
(8)                     assert sorted(G) == list(range(4))
(8)                     assert sorted(d for n, d in G.degree()) == [2, 2, 3, 3]
(8)                     assert nx.diameter(G) == 2
(8)                     assert nx.radius(G) == 1
(8)                     G = nx.dodecahedral_graph()
(8)                     assert sorted(G) == list(range(20))
(8)                     assert G.number_of_edges() == 30
(8)                     assert [d for n, d in G.degree()] == 20 * [3]
(8)                     assert nx.diameter(G) == 5
(8)                     assert nx.radius(G) == 5
(8)                     G = nx.frucht_graph()
(8)                     assert sorted(G) == list(range(12))
(8)                     assert G.number_of_edges() == 18
(8)                     assert [d for n, d in G.degree()] == 12 * [3]
(8)                     assert nx.diameter(G) == 4
(8)                     assert nx.radius(G) == 3
(8)                     G = nx.heawood_graph()
(8)                     assert sorted(G) == list(range(14))
(8)                     assert G.number_of_edges() == 21
(8)                     assert [d for n, d in G.degree()] == 14 * [3]
(8)                     assert nx.diameter(G) == 3
(8)                     assert nx.radius(G) == 3
(8)                     G = nx.hoffman_singleton_graph()
(8)                     assert sorted(G) == list(range(50))
(8)                     assert G.number_of_edges() == 175
(8)                     assert [d for n, d in G.degree()] == 50 * [7]
(8)                     assert nx.diameter(G) == 2
(8)                     assert nx.radius(G) == 2
(8)                     G = nx.house_graph()
(8)                     assert sorted(G) == list(range(5))
(8)                     assert G.number_of_edges() == 6
(8)                     assert sorted(d for n, d in G.degree()) == [2, 2, 2, 3, 3]
(8)                     assert nx.diameter(G) == 2
(8)                     assert nx.radius(G) == 2
(8)                     G = nx.house_x_graph()
(8)                     assert sorted(G) == list(range(5))
(8)                     assert G.number_of_edges() == 8
(8)                     assert sorted(d for n, d in G.degree()) == [2, 3, 3, 4, 4]
(8)                     assert nx.diameter(G) == 2
(8)                     assert nx.radius(G) == 1
(8)                     G = nx.icosahedral_graph()
(8)                     assert sorted(G) == list(range(12))
```

```
92: (8)                     assert G.number_of_edges() == 30
93: (8)                     assert [d for n, d in G.degree()] == [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5]
94: (8)                     assert nx.diameter(G) == 3
95: (8)                     assert nx.radius(G) == 3
96: (8)                     G = nx.krackhardt_kite_graph()
97: (8)                     assert sorted(G) == list(range(10))
98: (8)                     assert G.number_of_edges() == 18
99: (8)                     assert sorted(d for n, d in G.degree()) == [1, 2, 3, 3, 3, 4, 4, 5, 5,
6]
100: (8)                    G = nx.moebius_kantor_graph()
101: (8)                    assert sorted(G) == list(range(16))
102: (8)                    assert G.number_of_edges() == 24
103: (8)                    assert [d for n, d in G.degree()] == 16 * [3]
104: (8)                    assert nx.diameter(G) == 4
105: (8)                    G = nx.octahedral_graph()
106: (8)                    assert sorted(G) == list(range(6))
107: (8)                    assert G.number_of_edges() == 12
108: (8)                    assert [d for n, d in G.degree()] == 6 * [4]
109: (8)                    assert nx.diameter(G) == 2
110: (8)                    assert nx.radius(G) == 2
111: (8)                    G = nx.pappus_graph()
112: (8)                    assert sorted(G) == list(range(18))
113: (8)                    assert G.number_of_edges() == 27
114: (8)                    assert [d for n, d in G.degree()] == 18 * [3]
115: (8)                    assert nx.diameter(G) == 4
116: (8)                    G = nx.petersen_graph()
117: (8)                    assert sorted(G) == list(range(10))
118: (8)                    assert G.number_of_edges() == 15
119: (8)                    assert [d for n, d in G.degree()] == 10 * [3]
120: (8)                    assert nx.diameter(G) == 2
121: (8)                    assert nx.radius(G) == 2
122: (8)                    G = nx.sedgewick_maze_graph()
123: (8)                    assert sorted(G) == list(range(8))
124: (8)                    assert G.number_of_edges() == 10
125: (8)                    assert sorted(d for n, d in G.degree()) == [1, 2, 2, 2, 3, 3, 3, 4]
126: (8)                    G = nx.tetrahedral_graph()
127: (8)                    assert sorted(G) == list(range(4))
128: (8)                    assert G.number_of_edges() == 6
129: (8)                    assert [d for n, d in G.degree()] == [3, 3, 3, 3]
130: (8)                    assert nx.diameter(G) == 1
131: (8)                    assert nx.radius(G) == 1
132: (8)                    G = nx.truncated_cube_graph()
133: (8)                    assert sorted(G) == list(range(24))
134: (8)                    assert G.number_of_edges() == 36
135: (8)                    assert [d for n, d in G.degree()] == 24 * [3]
136: (8)                    G = nx.truncated_tetrahedron_graph()
137: (8)                    assert sorted(G) == list(range(12))
138: (8)                    assert G.number_of_edges() == 18
139: (8)                    assert [d for n, d in G.degree()] == 12 * [3]
140: (8)                    G = nx.tutte_graph()
141: (8)                    assert sorted(G) == list(range(46))
142: (8)                    assert G.number_of_edges() == 69
143: (8)                    assert [d for n, d in G.degree()] == 46 * [3]
144: (8)                    pytest.raises(nx.NetworkXError, nx.tutte_graph,
create_using=nx.DiGraph)
145: (8)                    MG = nx.tutte_graph(create_using=nx.MultiGraph)
146: (8)                    assert sorted(MG.edges()) == sorted(G.edges())
147: (0)            @pytest.mark.parametrize(
148: (4)                "fn",
149: (4)                (
150: (8)                    nx.bull_graph,
151: (8)                    nx.chvatal_graph,
152: (8)                    nx.cubical_graph,
153: (8)                    nx.diamond_graph,
154: (8)                    nx.house_graph,
155: (8)                    nx.house_x_graph,
156: (8)                    nx.icosahedral_graph,
157: (8)                    nx.krackhardt_kite_graph,
```

```
158: (8)                    nx.octahedral_graph,
159: (8)                    nx.petersen_graph,
160: (8)                    nx.truncated_cube_graph,
161: (8)                    nx.tutte_graph,
162: (4)                ),
163: (0)            )
164: (0)            @pytest.mark.parametrize(
165: (4)                "create_using", (nx.DiGraph, nx.MultiDiGraph, nx.DiGraph([(0, 1)]))
166: (0)            )
167: (0)            def tests_raises_with_directed_create_using(fn, create_using):
168: (4)                with pytest.raises(nx.NetworkXError, match="Directed Graph not
supported"):
169: (8)                    fn(create_using=create_using)
```

File 479 - test_spectral_graph_forge.py:

```
1: (0)              import pytest
2: (0)              pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              from networkx import is_isomorphic
5: (0)              from networkx.exception import NetworkXError
6: (0)              from networkx.generators import karate_club_graph
7: (0)              from networkx.generators.spectral_graph_forge import spectral_graph_forge
8: (0)              from networkx.utils import nodes_equal
9: (0)              def test_spectral_graph_forge():
10: (4)                 G = karate_club_graph()
11: (4)                 seed = 54321
12: (4)                 H = spectral_graph_forge(G, 0.1, transformation="identity", seed=seed)
13: (4)                 assert nodes_equal(G, H)
14: (4)                 I = spectral_graph_forge(G, 0.1, transformation="identity", seed=seed)
15: (4)                 assert nodes_equal(G, H)
16: (4)                 assert is_isomorphic(I, H)
17: (4)                 I = spectral_graph_forge(G, 0.1, transformation="modularity", seed=seed)
18: (4)                 assert nodes_equal(G, I)
19: (4)                 assert not is_isomorphic(I, H)
20: (4)                 H = spectral_graph_forge(G, 1, transformation="modularity", seed=seed)
21: (4)                 assert nodes_equal(G, H)
22: (4)                 assert is_isomorphic(G, H)
23: (4)                 H = spectral_graph_forge(G, -1, transformation="identity", seed=seed)
24: (4)                 assert nodes_equal(G, H)
25: (4)                 H = spectral_graph_forge(G, 10, transformation="identity", seed=seed)
26: (4)                 assert nodes_equal(G, H)
27: (4)                 assert is_isomorphic(G, H)
28: (4)                 pytest.raises(
29: (8)                     NetworkXError, spectral_graph_forge, G, 0.1, transformation="unknown",
seed=seed
30: (4)                 )
```

File 480 - test_stochastic.py:

```
1: (0)              """Unit tests for the :mod:`networkx.generators.stochastic` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              class TestStochasticGraph:
5: (4)                  """Unit tests for the :func:`~networkx.stochastic_graph` function."""
6: (4)                  def test_default_weights(self):
7: (8)                      G = nx.DiGraph()
8: (8)                      G.add_edge(0, 1)
9: (8)                      G.add_edge(0, 2)
10: (8)                     S = nx.stochastic_graph(G)
11: (8)                     assert nx.is_isomorphic(G, S)
12: (8)                     assert sorted(S.edges(data=True)) == [
13: (12)                        (0, 1, {"weight": 0.5}),
14: (12)                        (0, 2, {"weight": 0.5}),
15: (8)                     ]
```

```
16: (4)                 def test_in_place(self):
17: (8)                     """Tests for an in-place reweighting of the edges of the graph."""
18: (8)                     G = nx.DiGraph()
19: (8)                     G.add_edge(0, 1, weight=1)
20: (8)                     G.add_edge(0, 2, weight=1)
21: (8)                     nx.stochastic_graph(G, copy=False)
22: (8)                     assert sorted(G.edges(data=True)) == [
23: (12)                        (0, 1, {"weight": 0.5}),
24: (12)                        (0, 2, {"weight": 0.5}),
25: (8)                     ]
26: (4)                 def test_arbitrary_weights(self):
27: (8)                     G = nx.DiGraph()
28: (8)                     G.add_edge(0, 1, weight=1)
29: (8)                     G.add_edge(0, 2, weight=1)
30: (8)                     S = nx.stochastic_graph(G)
31: (8)                     assert sorted(S.edges(data=True)) == [
32: (12)                        (0, 1, {"weight": 0.5}),
33: (12)                        (0, 2, {"weight": 0.5}),
34: (8)                     ]
35: (4)                 def test_multidigraph(self):
36: (8)                     G = nx.MultiDiGraph()
37: (8)                     G.add_edges_from([(0, 1), (0, 1), (0, 2), (0, 2)])
38: (8)                     S = nx.stochastic_graph(G)
39: (8)                     d = {"weight": 0.25}
40: (8)                     assert sorted(S.edges(data=True)) == [
41: (12)                        (0, 1, d),
42: (12)                        (0, 1, d),
43: (12)                        (0, 2, d),
44: (12)                        (0, 2, d),
45: (8)                     ]
46: (4)                 def test_zero_weights(self):
47: (8)                     """Smoke test: ensure ZeroDivisionError is not raised."""
48: (8)                     G = nx.DiGraph()
49: (8)                     G.add_edge(0, 1, weight=0)
50: (8)                     G.add_edge(0, 2, weight=0)
51: (8)                     S = nx.stochastic_graph(G)
52: (8)                     assert sorted(S.edges(data=True)) == [
53: (12)                        (0, 1, {"weight": 0}),
54: (12)                        (0, 2, {"weight": 0}),
55: (8)                     ]
56: (4)                 def test_graph_disallowed(self):
57: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
58: (12)                        nx.stochastic_graph(nx.Graph())
59: (4)                 def test_multigraph_disallowed(self):
60: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
61: (12)                        nx.stochastic_graph(nx.MultiGraph())

----------------------------------------

File 481 - test_sudoku.py:

1: (0)              """Unit tests for the :mod:`networkx.generators.sudoku_graph` module."""
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              def test_sudoku_negative():
5: (4)                  """Raise an error when generating a Sudoku graph of order -1."""
6: (4)                  pytest.raises(nx.NetworkXError, nx.sudoku_graph, n=-1)
7: (0)              @pytest.mark.parametrize("n", [0, 1, 2, 3, 4])
8: (0)              def test_sudoku_generator(n):
9: (4)                  """Generate Sudoku graphs of various sizes and verify their properties."""
10: (4)                 G = nx.sudoku_graph(n)
11: (4)                 expected_nodes = n**4
12: (4)                 expected_degree = (n - 1) * (3 * n + 1)
13: (4)                 expected_edges = expected_nodes * expected_degree // 2
14: (4)                 assert not G.is_directed()
15: (4)                 assert not G.is_multigraph()
16: (4)                 assert G.number_of_nodes() == expected_nodes
17: (4)                 assert G.number_of_edges() == expected_edges
18: (4)                 assert all(d == expected_degree for _, d in G.degree)
```

```
(4)                 if n == 2:
(8)                     assert sorted(G.neighbors(6)) == [2, 3, 4, 5, 7, 10, 14]
(4)                 elif n == 3:
(8)                     assert sorted(G.neighbors(42)) == [
(12)                        6,
(12)                        15,
(12)                        24,
(12)                        33,
(12)                        34,
(12)                        35,
(12)                        36,
(12)                        37,
(12)                        38,
(12)                        39,
(12)                        40,
(12)                        41,
(12)                        43,
(12)                        44,
(12)                        51,
(12)                        52,
(12)                        53,
(12)                        60,
(12)                        69,
(12)                        78,
(8)                     ]
(4)                 elif n == 4:
(8)                     assert sorted(G.neighbors(0)) == [
(12)                        1,
(12)                        2,
(12)                        3,
(12)                        4,
(12)                        5,
(12)                        6,
(12)                        7,
(12)                        8,
(12)                        9,
(12)                        10,
(12)                        11,
(12)                        12,
(12)                        13,
(12)                        14,
(12)                        15,
(12)                        16,
(12)                        17,
(12)                        18,
(12)                        19,
(12)                        32,
(12)                        33,
(12)                        34,
(12)                        35,
(12)                        48,
(12)                        49,
(12)                        50,
(12)                        51,
(12)                        64,
(12)                        80,
(12)                        96,
(12)                        112,
(12)                        128,
(12)                        144,
(12)                        160,
(12)                        176,
(12)                        192,
(12)                        208,
(12)                        224,
(12)                        240,
(8)                     ]

----------------------------------------
```

File 482 - test_trees.py:

```
1: (0)              import random
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import arbitrary_element, graphs_equal
5: (0)              @pytest.mark.parametrize("prefix_tree_fn", (nx.prefix_tree,
nx.prefix_tree_recursive))
6: (0)              def test_basic_prefix_tree(prefix_tree_fn):
7: (4)                  strings = ["a", "to", "tea", "ted", "ten", "i", "in", "inn"]
8: (4)                  T = prefix_tree_fn(strings)
9: (4)                  root, NIL = 0, -1
10: (4)                 def source_label(v):
11: (8)                     return T.nodes[v]["source"]
12: (4)                 a, i, t = sorted(T[root], key=source_label)
13: (4)                 assert len(T[a]) == 1
14: (4)                 nil = arbitrary_element(T[a])
15: (4)                 assert len(T[nil]) == 0
16: (4)                 assert len(T[i]) == 2
17: (4)                 nil, in_ = sorted(T[i], key=source_label)
18: (4)                 assert len(T[nil]) == 0
19: (4)                 assert len(T[in_]) == 2
20: (4)                 nil, inn = sorted(T[in_], key=source_label)
21: (4)                 assert len(T[nil]) == 0
22: (4)                 assert len(T[inn]) == 1
23: (4)                 nil = arbitrary_element(T[inn])
24: (4)                 assert len(T[nil]) == 0
25: (4)                 te, to = sorted(T[t], key=source_label)
26: (4)                 assert len(T[to]) == 1
27: (4)                 nil = arbitrary_element(T[to])
28: (4)                 assert len(T[nil]) == 0
29: (4)                 tea, ted, ten = sorted(T[te], key=source_label)
30: (4)                 assert len(T[tea]) == 1
31: (4)                 assert len(T[ted]) == 1
32: (4)                 assert len(T[ten]) == 1
33: (4)                 nil = arbitrary_element(T[tea])
34: (4)                 assert len(T[nil]) == 0
35: (4)                 nil = arbitrary_element(T[ted])
36: (4)                 assert len(T[nil]) == 0
37: (4)                 nil = arbitrary_element(T[ten])
38: (4)                 assert len(T[nil]) == 0
39: (4)                 assert source_label(root) is None
40: (4)                 assert source_label(a) == "a"
41: (4)                 assert source_label(i) == "i"
42: (4)                 assert source_label(t) == "t"
43: (4)                 assert source_label(in_) == "n"
44: (4)                 assert source_label(inn) == "n"
45: (4)                 assert source_label(to) == "o"
46: (4)                 assert source_label(te) == "e"
47: (4)                 assert source_label(tea) == "a"
48: (4)                 assert source_label(ted) == "d"
49: (4)                 assert source_label(ten) == "n"
50: (4)                 assert source_label(NIL) == "NIL"
51: (0)             @pytest.mark.parametrize(
52: (4)                 "strings",
53: (4)                 (
54: (8)                     ["a", "to", "tea", "ted", "ten", "i", "in", "inn"],
55: (8)                     ["ab", "abs", "ad"],
56: (8)                     ["ab", "abs", "ad", ""],
57: (8)                     ["distant", "disparaging", "distant", "diamond", "ruby"],
58: (4)                 ),
59: (0)             )
60: (0)             def test_implementations_consistent(strings):
61: (4)                 """Ensure results are consistent between prefix_tree implementations."""
62: (4)                 assert graphs_equal(nx.prefix_tree(strings),
nx.prefix_tree_recursive(strings))
63: (0)             @pytest.mark.filterwarnings("ignore")
64: (0)             def test_random_tree():
```

```
65: (4)                 """Tests that a random tree is in fact a tree."""
66: (4)                 T = nx.random_tree(10, seed=1234)
67: (4)                 assert nx.is_tree(T)
68: (0)             @pytest.mark.filterwarnings("ignore")
69: (0)             def test_random_directed_tree():
70: (4)                 """Generates a directed tree."""
71: (4)                 T = nx.random_tree(10, seed=1234, create_using=nx.DiGraph())
72: (4)                 assert T.is_directed()
73: (0)             @pytest.mark.filterwarnings("ignore")
74: (0)             def test_random_tree_using_generator():
75: (4)                 """Tests that creating a random tree with a generator works"""
76: (4)                 G = nx.Graph()
77: (4)                 T = nx.random_tree(10, seed=1234, create_using=G)
78: (4)                 assert nx.is_tree(T)
79: (0)             def test_random_labeled_rooted_tree():
80: (4)                 for i in range(1, 10):
81: (8)                     t1 = nx.random_labeled_rooted_tree(i, seed=42)
82: (8)                     t2 = nx.random_labeled_rooted_tree(i, seed=42)
83: (8)                     assert nx.utils.misc.graphs_equal(t1, t2)
84: (8)                     assert nx.is_tree(t1)
85: (8)                     assert "root" in t1.graph
86: (8)                     assert "roots" not in t1.graph
87: (0)             def test_random_labeled_tree_n_zero():
88: (4)                 """Tests if n = 0 then the NetworkXPointlessConcept exception is
raised."""
89: (4)                 with pytest.raises(nx.NetworkXPointlessConcept):
90: (8)                     T = nx.random_labeled_tree(0, seed=1234)
91: (4)                 with pytest.raises(nx.NetworkXPointlessConcept):
92: (8)                     T = nx.random_labeled_rooted_tree(0, seed=1234)
93: (0)             def test_random_labeled_rooted_forest():
94: (4)                 for i in range(1, 10):
95: (8)                     t1 = nx.random_labeled_rooted_forest(i, seed=42)
96: (8)                     t2 = nx.random_labeled_rooted_forest(i, seed=42)
97: (8)                     assert nx.utils.misc.graphs_equal(t1, t2)
98: (8)                     for c in nx.connected_components(t1):
99: (12)                        assert nx.is_tree(t1.subgraph(c))
100: (8)                    assert "root" not in t1.graph
101: (8)                    assert "roots" in t1.graph
102: (0)            def test_random_labeled_rooted_forest_n_zero():
103: (4)                """Tests generation of empty labeled forests."""
104: (4)                F = nx.random_labeled_rooted_forest(0, seed=1234)
105: (4)                assert len(F) == 0
106: (4)                assert len(F.graph["roots"]) == 0
107: (0)            def test_random_unlabeled_rooted_tree():
108: (4)                for i in range(1, 10):
109: (8)                    t1 = nx.random_unlabeled_rooted_tree(i, seed=42)
110: (8)                    t2 = nx.random_unlabeled_rooted_tree(i, seed=42)
111: (8)                    assert nx.utils.misc.graphs_equal(t1, t2)
112: (8)                    assert nx.is_tree(t1)
113: (8)                    assert "root" in t1.graph
114: (8)                    assert "roots" not in t1.graph
115: (4)                t = nx.random_unlabeled_rooted_tree(15, number_of_trees=10, seed=43)
116: (4)                random.seed(43)
117: (4)                s = nx.random_unlabeled_rooted_tree(15, number_of_trees=10, seed=random)
118: (4)                for i in range(10):
119: (8)                    assert nx.utils.misc.graphs_equal(t[i], s[i])
120: (8)                    assert nx.is_tree(t[i])
121: (8)                    assert "root" in t[i].graph
122: (8)                    assert "roots" not in t[i].graph
123: (0)            def test_random_unlabeled_tree_n_zero():
124: (4)                """Tests if n = 0 then the NetworkXPointlessConcept exception is
raised."""
125: (4)                with pytest.raises(nx.NetworkXPointlessConcept):
126: (8)                    T = nx.random_unlabeled_tree(0, seed=1234)
127: (4)                with pytest.raises(nx.NetworkXPointlessConcept):
128: (8)                    T = nx.random_unlabeled_rooted_tree(0, seed=1234)
129: (0)            def test_random_unlabeled_rooted_forest():
130: (4)                with pytest.raises(ValueError):
131: (8)                    nx.random_unlabeled_rooted_forest(10, q=0, seed=42)
```

```
132: (4)                for i in range(1, 10):
133: (8)                    for q in range(1, i + 1):
134: (12)                       t1 = nx.random_unlabeled_rooted_forest(i, q=q, seed=42)
135: (12)                       t2 = nx.random_unlabeled_rooted_forest(i, q=q, seed=42)
136: (12)                       assert nx.utils.misc.graphs_equal(t1, t2)
137: (12)                       for c in nx.connected_components(t1):
138: (16)                           assert nx.is_tree(t1.subgraph(c))
139: (16)                           assert len(c) <= q
140: (12)                       assert "root" not in t1.graph
141: (12)                       assert "roots" in t1.graph
142: (4)                t = nx.random_unlabeled_rooted_forest(15, number_of_forests=10, seed=43)
143: (4)                random.seed(43)
144: (4)                s = nx.random_unlabeled_rooted_forest(15, number_of_forests=10,
seed=random)
145: (4)                for i in range(10):
146: (8)                    assert nx.utils.misc.graphs_equal(t[i], s[i])
147: (8)                    for c in nx.connected_components(t[i]):
148: (12)                       assert nx.is_tree(t[i].subgraph(c))
149: (8)                    assert "root" not in t[i].graph
150: (8)                    assert "roots" in t[i].graph
151: (0)            def test_random_unlabeled_forest_n_zero():
152: (4)                """Tests generation of empty unlabeled forests."""
153: (4)                F = nx.random_unlabeled_rooted_forest(0, seed=1234)
154: (4)                assert len(F) == 0
155: (4)                assert len(F.graph["roots"]) == 0
156: (0)            def test_random_unlabeled_tree():
157: (4)                for i in range(1, 10):
158: (8)                    t1 = nx.random_unlabeled_tree(i, seed=42)
159: (8)                    t2 = nx.random_unlabeled_tree(i, seed=42)
160: (8)                    assert nx.utils.misc.graphs_equal(t1, t2)
161: (8)                    assert nx.is_tree(t1)
162: (8)                    assert "root" not in t1.graph
163: (8)                    assert "roots" not in t1.graph
164: (4)                t = nx.random_unlabeled_tree(10, number_of_trees=10, seed=43)
165: (4)                random.seed(43)
166: (4)                s = nx.random_unlabeled_tree(10, number_of_trees=10, seed=random)
167: (4)                for i in range(10):
168: (8)                    assert nx.utils.misc.graphs_equal(t[i], s[i])
169: (8)                    assert nx.is_tree(t[i])
170: (8)                    assert "root" not in t[i].graph
171: (8)                    assert "roots" not in t[i].graph
```

File 483 - test_time_series.py:

```
1: (0)              """Unit tests for the :mod:`networkx.generators.time_series` module."""
2: (0)              import itertools
3: (0)              import networkx as nx
4: (0)              def test_visibility_graph__empty_series__empty_graph():
5: (4)                  null_graph = nx.visibility_graph([])  # move along nothing to see here
6: (4)                  assert nx.is_empty(null_graph)
7: (0)              def test_visibility_graph__single_value_ts__single_node_graph():
8: (4)                  node_graph = nx.visibility_graph([10])  # So Lonely
9: (4)                  assert node_graph.number_of_nodes() == 1
10: (4)                 assert node_graph.number_of_edges() == 0
11: (0)             def test_visibility_graph__two_values_ts__single_edge_graph():
12: (4)                 edge_graph = nx.visibility_graph([10, 20])  # Two of Us
13: (4)                 assert list(edge_graph.edges) == [(0, 1)]
14: (0)             def test_visibility_graph__convex_series__complete_graph():
15: (4)                 series = [i**2 for i in range(10)]  # no obstructions
16: (4)                 expected_series_length = len(series)
17: (4)                 actual_graph = nx.visibility_graph(series)
18: (4)                 assert actual_graph.number_of_nodes() == expected_series_length
19: (4)                 assert actual_graph.number_of_edges() == 45
20: (4)                 assert nx.is_isomorphic(actual_graph,
nx.complete_graph(expected_series_length))
21: (0)             def test_visibility_graph__concave_series__path_graph():
22: (4)                 series = [-(i**2) for i in range(10)]  # Slip Slidin' Away
```

```
23: (4)                 expected_node_count = len(series)
24: (4)                 actual_graph = nx.visibility_graph(series)
25: (4)                 assert actual_graph.number_of_nodes() == expected_node_count
26: (4)                 assert actual_graph.number_of_edges() == expected_node_count - 1
27: (4)                 assert nx.is_isomorphic(actual_graph, nx.path_graph(expected_node_count))
28: (0)             def test_visibility_graph__flat_series__path_graph():
29: (4)                 series = [0] * 10  # living in 1D flatland
30: (4)                 expected_node_count = len(series)
31: (4)                 actual_graph = nx.visibility_graph(series)
32: (4)                 assert actual_graph.number_of_nodes() == expected_node_count
33: (4)                 assert actual_graph.number_of_edges() == expected_node_count - 1
34: (4)                 assert nx.is_isomorphic(actual_graph, nx.path_graph(expected_node_count))
35: (0)             def test_visibility_graph_cyclic_series():
36: (4)                 series = list(itertools.islice(itertools.cycle((2, 1, 3)), 17))  # It's so
bumpy!
37: (4)                 expected_node_count = len(series)
38: (4)                 actual_graph = nx.visibility_graph(series)
39: (4)                 assert actual_graph.number_of_nodes() == expected_node_count
40: (4)                 assert actual_graph.number_of_edges() == 25

----------------------------------------

File 484 - test_triads.py:

1: (0)              """Unit tests for the :mod:`networkx.generators.triads` module."""
2: (0)              import pytest
3: (0)              from networkx import triad_graph
4: (0)              def test_triad_graph():
5: (4)                  G = triad_graph("030T")
6: (4)                  assert [tuple(e) for e in ("ab", "ac", "cb")] == sorted(G.edges())
7: (0)              def test_invalid_name():
8: (4)                  with pytest.raises(ValueError):
9: (8)                      triad_graph("bogus")

----------------------------------------

File 485 - __init__.py:

1: (0)              from networkx.linalg.attrmatrix import *
2: (0)              from networkx.linalg import attrmatrix
3: (0)              from networkx.linalg.spectrum import *
4: (0)              from networkx.linalg import spectrum
5: (0)              from networkx.linalg.graphmatrix import *
6: (0)              from networkx.linalg import graphmatrix
7: (0)              from networkx.linalg.laplacianmatrix import *
8: (0)              from networkx.linalg import laplacianmatrix
9: (0)              from networkx.linalg.algebraicconnectivity import *
10: (0)             from networkx.linalg.modularitymatrix import *
11: (0)             from networkx.linalg import modularitymatrix
12: (0)             from networkx.linalg.bethehessianmatrix import *
13: (0)             from networkx.linalg import bethehessianmatrix

----------------------------------------

File 486 - algebraicconnectivity.py:

1: (0)              """
2: (0)              Algebraic connectivity and Fiedler vectors of undirected graphs.
3: (0)              """
4: (0)              from functools import partial
5: (0)              import networkx as nx
6: (0)              from networkx.utils import (
7: (4)                  not_implemented_for,
8: (4)                  np_random_state,
9: (4)                  reverse_cuthill_mckee_ordering,
10: (0)             )
11: (0)             __all__ = [
12: (4)                 "algebraic_connectivity",
13: (4)                 "fiedler_vector",
```

```
(4)                 "spectral_ordering",
(4)                 "spectral_bisection",
(0)             ]
(0)             class _PCGSolver:
(4)                 """Preconditioned conjugate gradient method.
(4)                 To solve Ax = b:
(8)                     M = A.diagonal() # or some other preconditioner
(8)                     solver = _PCGSolver(lambda x: A * x, lambda x: M * x)
(8)                     x = solver.solve(b)
(4)                 The inputs A and M are functions which compute
(4)                 matrix multiplication on the argument.
(4)                 A - multiply by the matrix A in Ax=b
(4)                 M - multiply by M, the preconditioner surrogate for A
(4)                 Warning: There is no limit on number of iterations.
(4)                 """
(4)                 def __init__(self, A, M):
(8)                     self._A = A
(8)                     self._M = M
(4)                 def solve(self, B, tol):
(8)                     import numpy as np
(8)                     B = np.asarray(B)
(8)                     X = np.ndarray(B.shape, order="F")
(8)                     for j in range(B.shape[1]):
(12)                        X[:, j] = self._solve(B[:, j], tol)
(8)                     return X
(4)                 def _solve(self, b, tol):
(8)                     import numpy as np
(8)                     import scipy as sp
(8)                     A = self._A
(8)                     M = self._M
(8)                     tol *= sp.linalg.blas.dasum(b)
(8)                     x = np.zeros(b.shape)
(8)                     r = b.copy()
(8)                     z = M(r)
(8)                     rz = sp.linalg.blas.ddot(r, z)
(8)                     p = z.copy()
(8)                     while True:
(12)                        Ap = A(p)
(12)                        alpha = rz / sp.linalg.blas.ddot(p, Ap)
(12)                        x = sp.linalg.blas.daxpy(p, x, a=alpha)
(12)                        r = sp.linalg.blas.daxpy(Ap, r, a=-alpha)
(12)                        if sp.linalg.blas.dasum(r) < tol:
(16)                            return x
(12)                        z = M(r)
(12)                        beta = sp.linalg.blas.ddot(r, z)
(12)                        beta, rz = beta / rz, beta
(12)                        p = sp.linalg.blas.daxpy(p, z, a=beta)
(0)             class _LUSolver:
(4)                 """LU factorization.
(4)                 To solve Ax = b:
(8)                     solver = _LUSolver(A)
(8)                     x = solver.solve(b)
(4)                 optional argument `tol` on solve method is ignored but included
(4)                 to match _PCGsolver API.
(4)                 """
(4)                 def __init__(self, A):
(8)                     import scipy as sp
(8)                     self._LU = sp.sparse.linalg.splu(
(12)                        A,
(12)                        permc_spec="MMD_AT_PLUS_A",
(12)                        diag_pivot_thresh=0.0,
(12)                        options={"Equil": True, "SymmetricMode": True},
(8)                     )
(4)                 def solve(self, B, tol=None):
(8)                     import numpy as np
(8)                     B = np.asarray(B)
(8)                     X = np.ndarray(B.shape, order="F")
(8)                     for j in range(B.shape[1]):
(12)                        X[:, j] = self._LU.solve(B[:, j])
```

```
 83: (8)                     return X
 84: (0)             def _preprocess_graph(G, weight):
 85: (4)                 """Compute edge weights and eliminate zero-weight edges."""
 86: (4)                 if G.is_directed():
 87: (8)                     H = nx.MultiGraph()
 88: (8)                     H.add_nodes_from(G)
 89: (8)                     H.add_weighted_edges_from(
 90: (12)                        ((u, v, e.get(weight, 1.0)) for u, v, e in G.edges(data=True) if u
 != v),
 91: (12)                        weight=weight,
 92: (8)                     )
 93: (8)                     G = H
 94: (4)                 if not G.is_multigraph():
 95: (8)                     edges = (
 96: (12)                        (u, v, abs(e.get(weight, 1.0))) for u, v, e in G.edges(data=True)
 if u != v
 97: (8)                     )
 98: (4)                 else:
 99: (8)                     edges = (
 100: (12)                       (u, v, sum(abs(e.get(weight, 1.0)) for e in G[u][v].values()))
 101: (12)                       for u, v in G.edges()
 102: (12)                       if u != v
 103: (8)                    )
 104: (4)                H = nx.Graph()
 105: (4)                H.add_nodes_from(G)
 106: (4)                H.add_weighted_edges_from((u, v, e) for u, v, e in edges if e != 0)
 107: (4)                return H
 108: (0)            def _rcm_estimate(G, nodelist):
 109: (4)                """Estimate the Fiedler vector using the reverse Cuthill-McKee
 ordering."""
 110: (4)                import numpy as np
 111: (4)                G = G.subgraph(nodelist)
 112: (4)                order = reverse_cuthill_mckee_ordering(G)
 113: (4)                n = len(nodelist)
 114: (4)                index = dict(zip(nodelist, range(n)))
 115: (4)                x = np.ndarray(n, dtype=float)
 116: (4)                for i, u in enumerate(order):
 117: (8)                    x[index[u]] = i
 118: (4)                x -= (n - 1) / 2.0
 119: (4)                return x
 120: (0)            def _tracemin_fiedler(L, X, normalized, tol, method):
 121: (4)                """Compute the Fiedler vector of L using the TraceMIN-Fiedler algorithm.
 122: (4)                The Fiedler vector of a connected undirected graph is the eigenvector
 123: (4)                corresponding to the second smallest eigenvalue of the Laplacian matrix
 124: (4)                of the graph. This function starts with the Laplacian L, not the Graph.
 125: (4)                Parameters
 126: (4)                ----------
 127: (4)                L : Laplacian of a possibly weighted or normalized, but undirected graph
 128: (4)                X : Initial guess for a solution. Usually a matrix of random numbers.
 129: (8)                    This function allows more than one column in X to identify more than
 130: (8)                    one eigenvector if desired.
 131: (4)                normalized : bool
 132: (8)                    Whether the normalized Laplacian matrix is used.
 133: (4)                tol : float
 134: (8)                    Tolerance of relative residual in eigenvalue computation.
 135: (8)                    Warning: There is no limit on number of iterations.
 136: (4)                method : string
 137: (8)                    Should be 'tracemin_pcg' or 'tracemin_lu'.
 138: (8)                    Otherwise exception is raised.
 139: (4)                Returns
 140: (4)                -------
 141: (4)                sigma, X : Two NumPy arrays of floats.
 142: (8)                    The lowest eigenvalues and corresponding eigenvectors of L.
 143: (8)                    The size of input X determines the size of these outputs.
 144: (8)                    As this is for Fiedler vectors, the zero eigenvalue (and
 145: (8)                    constant eigenvector) are avoided.
 146: (4)                """
 147: (4)                import numpy as np
 148: (4)                import scipy as sp
```

```
149: (4)                n = X.shape[0]
150: (4)                if normalized:
151: (8)                    e = np.sqrt(L.diagonal())
152: (8)                    D = sp.sparse.csr_array(sp.sparse.spdiags(1 / e, 0, n, n,
format="csr"))
153: (8)                    L = D @ L @ D
154: (8)                    e *= 1.0 / np.linalg.norm(e, 2)
155: (4)                if normalized:
156: (8)                    def project(X):
157: (12)                       """Make X orthogonal to the nullspace of L."""
158: (12)                       X = np.asarray(X)
159: (12)                       for j in range(X.shape[1]):
160: (16)                           X[:, j] -= (X[:, j] @ e) * e
161: (4)                else:
162: (8)                    def project(X):
163: (12)                       """Make X orthogonal to the nullspace of L."""
164: (12)                       X = np.asarray(X)
165: (12)                       for j in range(X.shape[1]):
166: (16)                           X[:, j] -= X[:, j].sum() / n
167: (4)                if method == "tracemin_pcg":
168: (8)                    D = L.diagonal().astype(float)
169: (8)                    solver = _PCGSolver(lambda x: L @ x, lambda x: D * x)
170: (4)                elif method == "tracemin_lu":
171: (8)                    A = sp.sparse.csc_array(L, dtype=float, copy=True)
172: (8)                    i = (A.indptr[1:] - A.indptr[:-1]).argmax()
173: (8)                    A[i, i] = np.inf
174: (8)                    solver = _LUSolver(A)
175: (4)                else:
176: (8)                    raise nx.NetworkXError(f"Unknown linear system solver: {method}")
177: (4)                Lnorm = abs(L).sum(axis=1).flatten().max()
178: (4)                project(X)
179: (4)                W = np.ndarray(X.shape, order="F")
180: (4)                while True:
181: (8)                    X = np.linalg.qr(X)[0]
182: (8)                    W[:, :] = L @ X
183: (8)                    H = X.T @ W
184: (8)                    sigma, Y = sp.linalg.eigh(H, overwrite_a=True)
185: (8)                    X = X @ Y
186: (8)                    res = sp.linalg.blas.dasum(W @ Y[:, 0] - sigma[0] * X[:, 0]) / Lnorm
187: (8)                    if res < tol:
188: (12)                       break
189: (8)                    W[:, :] = solver.solve(X, tol)
190: (8)                    X = (sp.linalg.inv(W.T @ X) @ W.T).T  # Preserves Fortran storage
order.
191: (8)                    project(X)
192: (4)                return sigma, np.asarray(X)
193: (0)            def _get_fiedler_func(method):
194: (4)                """Returns a function that solves the Fiedler eigenvalue problem."""
195: (4)                import numpy as np
196: (4)                if method == "tracemin":  # old style keyword <v2.1
197: (8)                    method = "tracemin_pcg"
198: (4)                if method in ("tracemin_pcg", "tracemin_lu"):
199: (8)                    def find_fiedler(L, x, normalized, tol, seed):
200: (12)                       q = 1 if method == "tracemin_pcg" else min(4, L.shape[0] - 1)
201: (12)                       X = np.asarray(seed.normal(size=(q, L.shape[0]))).T
202: (12)                       sigma, X = _tracemin_fiedler(L, X, normalized, tol, method)
203: (12)                       return sigma[0], X[:, 0]
204: (4)                elif method == "lanczos" or method == "lobpcg":
205: (8)                    def find_fiedler(L, x, normalized, tol, seed):
206: (12)                       import scipy as sp
207: (12)                       L = sp.sparse.csc_array(L, dtype=float)
208: (12)                       n = L.shape[0]
209: (12)                       if normalized:
210: (16)                           D = sp.sparse.csc_array(
211: (20)                               sp.sparse.spdiags(
212: (24)                                   1.0 / np.sqrt(L.diagonal()), [0], n, n, format="csc"
213: (20)                               )
214: (16)                           )
215: (16)                           L = D @ L @ D
```

```
216: (12)                        if method == "lanczos" or n < 10:
217: (16)                            sigma, X = sp.sparse.linalg.eigsh(
218: (20)                                L, 2, which="SM", tol=tol, return_eigenvectors=True
219: (16)                            )
220: (16)                            return sigma[1], X[:, 1]
221: (12)                        else:
222: (16)                            X = np.asarray(np.atleast_2d(x).T)
223: (16)                            M = sp.sparse.csr_array(sp.sparse.spdiags(1.0 / L.diagonal(),
0, n, n))
224: (16)                            Y = np.ones(n)
225: (16)                            if normalized:
226: (20)                                Y /= D.diagonal()
227: (16)                            sigma, X = sp.sparse.linalg.lobpcg(
228: (20)                                L, X, M=M, Y=np.atleast_2d(Y).T, tol=tol, maxiter=n,
largest=False
229: (16)                            )
230: (16)                            return sigma[0], X[:, 0]
231: (4)                else:
232: (8)                    raise nx.NetworkXError(f"unknown method {method!r}.")
233: (4)                return find_fiedler
234: (0)            @not_implemented_for("directed")
235: (0)            @np_random_state(5)
236: (0)            @nx._dispatchable(edge_attrs="weight")
237: (0)            def algebraic_connectivity(
238: (4)                G, weight="weight", normalized=False, tol=1e-8, method="tracemin_pcg",
seed=None
239: (0)            ):
240: (4)                r"""Returns the algebraic connectivity of an undirected graph.
241: (4)                The algebraic connectivity of a connected undirected graph is the second
242: (4)                smallest eigenvalue of its Laplacian matrix.
243: (4)                Parameters
244: (4)                ----------
245: (4)                G : NetworkX graph
246: (8)                    An undirected graph.
247: (4)                weight : object, optional (default: None)
248: (8)                    The data key used to determine the weight of each edge. If None, then
249: (8)                    each edge has unit weight.
250: (4)                normalized : bool, optional (default: False)
251: (8)                    Whether the normalized Laplacian matrix is used.
252: (4)                tol : float, optional (default: 1e-8)
253: (8)                    Tolerance of relative residual in eigenvalue computation.
254: (4)                method : string, optional (default: 'tracemin_pcg')
255: (8)                    Method of eigenvalue computation. It must be one of the tracemin
256: (8)                    options shown below (TraceMIN), 'lanczos' (Lanczos iteration)
257: (8)                    or 'lobpcg' (LOBPCG).
258: (8)                    The TraceMIN algorithm uses a linear system solver. The following
259: (8)                    values allow specifying the solver to be used.
260: (8)                    =============== ========================================
261: (8)                    Value           Solver
262: (8)                    =============== ========================================
263: (8)                    'tracemin_pcg'  Preconditioned conjugate gradient method
264: (8)                    'tracemin_lu'   LU factorization
265: (8)                    =============== ========================================
266: (4)                seed : integer, random_state, or None (default)
267: (8)                    Indicator of random number generation state.
268: (8)                    See :ref:`Randomness<randomness>`.
269: (4)                Returns
270: (4)                -------
271: (4)                algebraic_connectivity : float
272: (8)                    Algebraic connectivity.
273: (4)                Raises
274: (4)                ------
275: (4)                NetworkXNotImplemented
276: (8)                    If G is directed.
277: (4)                NetworkXError
278: (8)                    If G has less than two nodes.
279: (4)                Notes
280: (4)                -----
281: (4)                Edge weights are interpreted by their absolute values. For MultiGraph's,
```

```
(4)                weights of parallel edges are summed. Zero-weighted edges are ignored.
(4)                See Also
(4)                --------
(4)                laplacian_matrix
(4)                Examples
(4)                --------
(4)                For undirected graphs algebraic connectivity can tell us if a graph is connected or not
(4)                `G` is connected iff  ``algebraic_connectivity(G) > 0``:
(4)                >>> G = nx.complete_graph(5)
(4)                >>> nx.algebraic_connectivity(G) > 0
(4)                True
(4)                >>> G.add_node(10)  # G is no longer connected
(4)                >>> nx.algebraic_connectivity(G) > 0
(4)                False
(4)                """
(4)                if len(G) < 2:
(8)                    raise nx.NetworkXError("graph has less than two nodes.")
(4)                G = _preprocess_graph(G, weight)
(4)                if not nx.is_connected(G):
(8)                    return 0.0
(4)                L = nx.laplacian_matrix(G)
(4)                if L.shape[0] == 2:
(8)                    return 2.0 * float(L[0, 0]) if not normalized else 2.0
(4)                find_fiedler = _get_fiedler_func(method)
(4)                x = None if method != "lobpcg" else _rcm_estimate(G, G)
(4)                sigma, fiedler = find_fiedler(L, x, normalized, tol, seed)
(4)                return float(sigma)
(0)            @not_implemented_for("directed")
(0)            @np_random_state(5)
(0)            @nx._dispatchable(edge_attrs="weight")
(0)            def fiedler_vector(
(4)                G, weight="weight", normalized=False, tol=1e-8, method="tracemin_pcg", seed=None
(0)            ):
(4)                """Returns the Fiedler vector of a connected undirected graph.
(4)                The Fiedler vector of a connected undirected graph is the eigenvector
(4)                corresponding to the second smallest eigenvalue of the Laplacian matrix
(4)                of the graph.
(4)                Parameters
(4)                ----------
(4)                G : NetworkX graph
(8)                    An undirected graph.
(4)                weight : object, optional (default: None)
(8)                    The data key used to determine the weight of each edge. If None, then
(8)                    each edge has unit weight.
(4)                normalized : bool, optional (default: False)
(8)                    Whether the normalized Laplacian matrix is used.
(4)                tol : float, optional (default: 1e-8)
(8)                    Tolerance of relative residual in eigenvalue computation.
(4)                method : string, optional (default: 'tracemin_pcg')
(8)                    Method of eigenvalue computation. It must be one of the tracemin
(8)                    options shown below (TraceMIN), 'lanczos' (Lanczos iteration)
(8)                    or 'lobpcg' (LOBPCG).
(8)                    The TraceMIN algorithm uses a linear system solver. The following
(8)                    values allow specifying the solver to be used.
(8)                    =============== ========================================
(8)                    Value           Solver
(8)                    =============== ========================================
(8)                    'tracemin_pcg'  Preconditioned conjugate gradient method
(8)                    'tracemin_lu'   LU factorization
(8)                    =============== ========================================
(4)                seed : integer, random_state, or None (default)
(8)                    Indicator of random number generation state.
(8)                    See :ref:`Randomness<randomness>`.
(4)                Returns
(4)                -------
(4)                fiedler_vector : NumPy array of floats.
(8)                    Fiedler vector.
```

```
    Raises
    ------
    NetworkXNotImplemented
        If G is directed.
    NetworkXError
        If G has less than two nodes or is not connected.
    Notes
    -----
    Edge weights are interpreted by their absolute values. For MultiGraph's,
    weights of parallel edges are summed. Zero-weighted edges are ignored.
    See Also
    --------
    laplacian_matrix
    Examples
    --------
    Given a connected graph the signs of the values in the Fiedler vector can be
    used to partition the graph into two components.
    >>> G = nx.barbell_graph(5, 0)
    >>> nx.fiedler_vector(G, normalized=True, seed=1)
    array([-0.32864129, -0.32864129, -0.32864129, -0.32864129, -0.26072899,
            0.26072899,  0.32864129,  0.32864129,  0.32864129,  0.32864129])
    The connected components are the two 5-node cliques of the barbell graph.
    """
    import numpy as np
    if len(G) < 2:
        raise nx.NetworkXError("graph has less than two nodes.")
    G = _preprocess_graph(G, weight)
    if not nx.is_connected(G):
        raise nx.NetworkXError("graph is not connected.")
    if len(G) == 2:
        return np.array([1.0, -1.0])
    find_fiedler = _get_fiedler_func(method)
    L = nx.laplacian_matrix(G)
    x = None if method != "lobpcg" else _rcm_estimate(G, G)
    sigma, fiedler = find_fiedler(L, x, normalized, tol, seed)
    return fiedler
@np_random_state(5)
@nx._dispatchable(edge_attrs="weight")
def spectral_ordering(
    G, weight="weight", normalized=False, tol=1e-8, method="tracemin_pcg", seed=None
):
    """Compute the spectral_ordering of a graph.
    The spectral ordering of a graph is an ordering of its nodes where nodes
    in the same weakly connected components appear contiguous and ordered by
    their corresponding elements in the Fiedler vector of the component.
    Parameters
    ----------
    G : NetworkX graph
        A graph.
    weight : object, optional (default: None)
        The data key used to determine the weight of each edge. If None, then
        each edge has unit weight.
    normalized : bool, optional (default: False)
        Whether the normalized Laplacian matrix is used.
    tol : float, optional (default: 1e-8)
        Tolerance of relative residual in eigenvalue computation.
    method : string, optional (default: 'tracemin_pcg')
        Method of eigenvalue computation. It must be one of the tracemin
        options shown below (TraceMIN), 'lanczos' (Lanczos iteration)
        or 'lobpcg' (LOBPCG).
        The TraceMIN algorithm uses a linear system solver. The following
        values allow specifying the solver to be used.
        =============== ========================================
        Value           Solver
        =============== ========================================
        'tracemin_pcg'  Preconditioned conjugate gradient method
        'tracemin_lu'   LU factorization
```

```
416: (8)                    ============== ========================================
417: (4)                seed : integer, random_state, or None (default)
418: (8)                    Indicator of random number generation state.
419: (8)                    See :ref:`Randomness<randomness>`.
420: (4)                Returns
421: (4)                -------
422: (4)                spectral_ordering : NumPy array of floats.
423: (8)                    Spectral ordering of nodes.
424: (4)                Raises
425: (4)                ------
426: (4)                NetworkXError
427: (8)                    If G is empty.
428: (4)                Notes
429: (4)                -----
430: (4)                Edge weights are interpreted by their absolute values. For MultiGraph's,
431: (4)                weights of parallel edges are summed. Zero-weighted edges are ignored.
432: (4)                See Also
433: (4)                --------
434: (4)                laplacian_matrix
435: (4)                """
436: (4)                if len(G) == 0:
437: (8)                    raise nx.NetworkXError("graph is empty.")
438: (4)                G = _preprocess_graph(G, weight)
439: (4)                find_fiedler = _get_fiedler_func(method)
440: (4)                order = []
441: (4)                for component in nx.connected_components(G):
442: (8)                    size = len(component)
443: (8)                    if size > 2:
444: (12)                       L = nx.laplacian_matrix(G, component)
445: (12)                       x = None if method != "lobpcg" else _rcm_estimate(G, component)
446: (12)                       sigma, fiedler = find_fiedler(L, x, normalized, tol, seed)
447: (12)                       sort_info = zip(fiedler, range(size), component)
448: (12)                       order.extend(u for x, c, u in sorted(sort_info))
449: (8)                    else:
450: (12)                       order.extend(component)
451: (4)                return order
452: (0)            @nx._dispatchable(edge_attrs="weight")
453: (0)            def spectral_bisection(
454: (4)                G, weight="weight", normalized=False, tol=1e-8, method="tracemin_pcg",
seed=None
455: (0)            ):
456: (4)                """Bisect the graph using the Fiedler vector.
457: (4)                This method uses the Fiedler vector to bisect a graph.
458: (4)                The partition is defined by the nodes which are associated with
459: (4)                either positive or negative values in the vector.
460: (4)                Parameters
461: (4)                ----------
462: (4)                G : NetworkX Graph
463: (4)                weight : str, optional (default: weight)
464: (8)                    The data key used to determine the weight of each edge. If None, then
465: (8)                    each edge has unit weight.
466: (4)                normalized : bool, optional (default: False)
467: (8)                    Whether the normalized Laplacian matrix is used.
468: (4)                tol : float, optional (default: 1e-8)
469: (8)                    Tolerance of relative residual in eigenvalue computation.
470: (4)                method : string, optional (default: 'tracemin_pcg')
471: (8)                    Method of eigenvalue computation. It must be one of the tracemin
472: (8)                    options shown below (TraceMIN), 'lanczos' (Lanczos iteration)
473: (8)                    or 'lobpcg' (LOBPCG).
474: (8)                    The TraceMIN algorithm uses a linear system solver. The following
475: (8)                    values allow specifying the solver to be used.
476: (8)                    ============== ========================================
477: (8)                    Value          Solver
478: (8)                    ============== ========================================
479: (8)                    'tracemin_pcg'  Preconditioned conjugate gradient method
480: (8)                    'tracemin_lu'   LU factorization
481: (8)                    ============== ========================================
482: (4)                seed : integer, random_state, or None (default)
483: (8)                    Indicator of random number generation state.
```

```
484: (8)                    See :ref:`Randomness<randomness>`.
485: (4)                Returns
486: (4)                -------
487: (4)                bisection : tuple of sets
488: (8)                    Sets with the bisection of nodes
489: (4)                Examples
490: (4)                --------
491: (4)                >>> G = nx.barbell_graph(3, 0)
492: (4)                >>> nx.spectral_bisection(G)
493: (4)                ({0, 1, 2}, {3, 4, 5})
494: (4)                References
495: (4)                ----------
496: (4)                .. [1] M. E. J Newman 'Networks: An Introduction', pages 364-370
497: (7)                   Oxford University Press 2011.
498: (4)                """
499: (4)                import numpy as np
500: (4)                v = nx.fiedler_vector(G, weight, normalized, tol, method, seed)
501: (4)                nodes = np.array(list(G))
502: (4)                pos_vals = v >= 0
503: (4)                return set(nodes[~pos_vals].tolist()), set(nodes[pos_vals].tolist())
```

File 487 - attrmatrix.py:

```
1: (0)              """
2: (4)                  Functions for constructing matrix-like objects from graph attributes.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = ["attr_matrix", "attr_sparse_matrix"]
6: (0)              def _node_value(G, node_attr):
7: (4)                  """Returns a function that returns a value from G.nodes[u].
8: (4)                  We return a function expecting a node as its sole argument. Then, in the
9: (4)                  simplest scenario, the returned function will return G.nodes[u]
[node_attr].
10: (4)                 However, we also handle the case when `node_attr` is None or when it is a
11: (4)                 function itself.
12: (4)                 Parameters
13: (4)                 ----------
14: (4)                 G : graph
15: (8)                     A NetworkX graph
16: (4)                 node_attr : {None, str, callable}
17: (8)                     Specification of how the value of the node attribute should be
obtained
18: (8)                     from the node attribute dictionary.
19: (4)                 Returns
20: (4)                 -------
21: (4)                 value : function
22: (8)                     A function expecting a node as its sole argument. The function will
23: (8)                     returns a value from G.nodes[u] that depends on `edge_attr`.
24: (4)                 """
25: (4)                 if node_attr is None:
26: (8)                     def value(u):
27: (12)                        return u
28: (4)                 elif not callable(node_attr):
29: (8)                     def value(u):
30: (12)                        return G.nodes[u][node_attr]
31: (4)                 else:
32: (8)                     value = node_attr
33: (4)                 return value
34: (0)             def _edge_value(G, edge_attr):
35: (4)                 """Returns a function that returns a value from G[u][v].
36: (4)                 Suppose there exists an edge between u and v.  Then we return a function
37: (4)                 expecting u and v as arguments.  For Graph and DiGraph, G[u][v] is
38: (4)                 the edge attribute dictionary, and the function (essentially) returns
39: (4)                 G[u][v][edge_attr].  However, we also handle cases when `edge_attr` is
None
40: (4)                 and when it is a function itself. For MultiGraph and MultiDiGraph, G[u][v]
41: (4)                 is a dictionary of all edges between u and v.  In this case, the returned
```

```
    function sums the value of `edge_attr` for every edge between u and v.
    Parameters
    ----------
    G : graph
       A NetworkX graph
    edge_attr : {None, str, callable}
        Specification of how the value of the edge attribute should be obtained

        from the edge attribute dictionary, G[u][v].  For multigraphs, G[u][v]
        is a dictionary of all the edges between u and v.  This allows for
        special treatment of multiedges.
    Returns
    -------
    value : function
        A function expecting two nodes as parameters. The nodes should
        represent the from- and to- node of an edge. The function will
        return a value from G[u][v] that depends on `edge_attr`.
    """
    if edge_attr is None:
        if G.is_multigraph():
            def value(u, v):
                return len(G[u][v])
        else:
            def value(u, v):
                return 1
    elif not callable(edge_attr):
        if edge_attr == "weight":
            if G.is_multigraph():
                def value(u, v):
                    return sum(d.get(edge_attr, 1) for d in G[u][v].values())
            else:
                def value(u, v):
                    return G[u][v].get(edge_attr, 1)
        else:
            if G.is_multigraph():
                def value(u, v):
                    return sum(d[edge_attr] for d in G[u][v].values())
            else:
                def value(u, v):
                    return G[u][v][edge_attr]
    else:
        value = edge_attr
    return value
@nx._dispatchable(edge_attrs={"edge_attr": None}, node_attrs="node_attr")
def attr_matrix(
    G,
    edge_attr=None,
    node_attr=None,
    normalized=False,
    rc_order=None,
    dtype=None,
    order=None,
):
    """Returns the attribute matrix using attributes from `G` as a numpy array.
    If only `G` is passed in, then the adjacency matrix is constructed.
    Let A be a discrete set of values for the node attribute `node_attr`. Then
    the elements of A represent the rows and columns of the constructed matrix.
    Now, iterate through every edge e=(u,v) in `G` and consider the value
    of the edge attribute `edge_attr`.  If ua and va are the values of the
    node attribute `node_attr` for u and v, respectively, then the value of
    the edge attribute is added to the matrix element at (ua, va).
    Parameters
    ----------
    G : graph
        The NetworkX graph used to construct the attribute matrix.
    edge_attr : str, optional
        Each element of the matrix represents a running total of the
```

```
    specified edge attribute for edges whose node attributes correspond
    to the rows/cols of the matrix. The attribute must be present for
    all edges in the graph. If no attribute is specified, then we
    just count the number of edges whose node attributes correspond
    to the matrix element.
node_attr : str, optional
    Each row and column in the matrix represents a particular value
    of the node attribute.  The attribute must be present for all nodes
    in the graph. Note, the values of this attribute should be reliably
    hashable. So, float values are not recommended. If no attribute is
    specified, then the rows and columns will be the nodes of the graph.
normalized : bool, optional
    If True, then each row is normalized by the summation of its values.
rc_order : list, optional
    A list of the node attribute values. This list specifies the ordering
    of rows and columns of the array. If no ordering is provided, then
    the ordering will be random (and also, a return value).
Other Parameters
----------------
dtype : NumPy data-type, optional
    A valid NumPy dtype used to initialize the array. Keep in mind certain
    dtypes can yield unexpected results if the array is to be normalized.
    The parameter is passed to numpy.zeros(). If unspecified, the NumPy
    default is used.
order : {'C', 'F'}, optional
    Whether to store multidimensional data in C- or Fortran-contiguous
    (row- or column-wise) order in memory. This parameter is passed to
    numpy.zeros(). If unspecified, the NumPy default is used.
Returns
-------
M : 2D NumPy ndarray
    The attribute matrix.
ordering : list
    If `rc_order` was specified, then only the attribute matrix is returned.

    However, if `rc_order` was None, then the ordering used to construct
    the matrix is returned as well.
Examples
--------
Construct an adjacency matrix:
>>> G = nx.Graph()
>>> G.add_edge(0, 1, thickness=1, weight=3)
>>> G.add_edge(0, 2, thickness=2)
>>> G.add_edge(1, 2, thickness=3)
>>> nx.attr_matrix(G, rc_order=[0, 1, 2])
array([[0., 1., 1.],
       [1., 0., 1.],
       [1., 1., 0.]])
Alternatively, we can obtain the matrix describing edge thickness.
>>> nx.attr_matrix(G, edge_attr="thickness", rc_order=[0, 1, 2])
array([[0., 1., 2.],
       [1., 0., 3.],
       [2., 3., 0.]])
We can also color the nodes and ask for the probability distribution over
all edges (u,v) describing:
    Pr(v has color Y | u has color X)
>>> G.nodes[0]["color"] = "red"
>>> G.nodes[1]["color"] = "red"
>>> G.nodes[2]["color"] = "blue"
>>> rc = ["red", "blue"]
>>> nx.attr_matrix(G, node_attr="color", normalized=True, rc_order=rc)
array([[0.33333333, 0.66666667],
       [1.        , 0.        ]])
For example, the above tells us that for all edges (u,v):
    Pr( v is red  | u is red)  = 1/3
    Pr( v is blue | u is red)  = 2/3
    Pr( v is red  | u is blue) = 1
    Pr( v is blue | u is blue) = 0
Finally, we can obtain the total weights listed by the node colors.
```

```
(4)                 >>> nx.attr_matrix(G, edge_attr="weight", node_attr="color", rc_order=rc)
(4)                 array([[3., 2.],
(11)                       [2., 0.]])
(4)                 Thus, the total weight over all edges (u,v) with u and v having colors:
(8)                     (red, red)   is 3   # the sole contribution is from edge (0,1)
(8)                     (red, blue)  is 2   # contributions from edges (0,2) and (1,2)
(8)                     (blue, red)  is 2   # same as (red, blue) since graph is undirected
(8)                     (blue, blue) is 0   # there are no edges with blue endpoints
(4)                 """
(4)                 import numpy as np
(4)                 edge_value = _edge_value(G, edge_attr)
(4)                 node_value = _node_value(G, node_attr)
(4)                 if rc_order is None:
(8)                     ordering = list({node_value(n) for n in G})
(4)                 else:
(8)                     ordering = rc_order
(4)                 N = len(ordering)
(4)                 undirected = not G.is_directed()
(4)                 index = dict(zip(ordering, range(N)))
(4)                 M = np.zeros((N, N), dtype=dtype, order=order)
(4)                 seen = set()
(4)                 for u, nbrdict in G.adjacency():
(8)                     for v in nbrdict:
(12)                        i, j = index[node_value(u)], index[node_value(v)]
(12)                        if v not in seen:
(16)                            M[i, j] += edge_value(u, v)
(16)                            if undirected:
(20)                                M[j, i] = M[i, j]
(8)                     if undirected:
(12)                        seen.add(u)
(4)                 if normalized:
(8)                     M /= M.sum(axis=1).reshape((N, 1))
(4)                 if rc_order is None:
(8)                     return M, ordering
(4)                 else:
(8)                     return M
(0)             @nx._dispatchable(edge_attrs={"edge_attr": None}, node_attrs="node_attr")
(0)             def attr_sparse_matrix(
(4)                 G, edge_attr=None, node_attr=None, normalized=False, rc_order=None,
dtype=None
(0)             ):
(4)                 """Returns a SciPy sparse array using attributes from G.
(4)                 If only `G` is passed in, then the adjacency matrix is constructed.
(4)                 Let A be a discrete set of values for the node attribute `node_attr`. Then
(4)                 the elements of A represent the rows and columns of the constructed
matrix.
(4)                 Now, iterate through every edge e=(u,v) in `G` and consider the value
(4)                 of the edge attribute `edge_attr`.  If ua and va are the values of the
(4)                 node attribute `node_attr` for u and v, respectively, then the value of
(4)                 the edge attribute is added to the matrix element at (ua, va).
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(8)                     The NetworkX graph used to construct the NumPy matrix.
(4)                 edge_attr : str, optional
(8)                     Each element of the matrix represents a running total of the
(8)                     specified edge attribute for edges whose node attributes correspond
(8)                     to the rows/cols of the matrix. The attribute must be present for
(8)                     all edges in the graph. If no attribute is specified, then we
(8)                     just count the number of edges whose node attributes correspond
(8)                     to the matrix element.
(4)                 node_attr : str, optional
(8)                     Each row and column in the matrix represents a particular value
(8)                     of the node attribute.  The attribute must be present for all nodes
(8)                     in the graph. Note, the values of this attribute should be reliably
(8)                     hashable. So, float values are not recommended. If no attribute is
(8)                     specified, then the rows and columns will be the nodes of the graph.
(4)                 normalized : bool, optional
(8)                     If True, then each row is normalized by the summation of its values.
```

```
    rc_order : list, optional
        A list of the node attribute values. This list specifies the ordering
        of rows and columns of the array. If no ordering is provided, then
        the ordering will be random (and also, a return value).
    Other Parameters
    ----------------
    dtype : NumPy data-type, optional
        A valid NumPy dtype used to initialize the array. Keep in mind certain
        dtypes can yield unexpected results if the array is to be normalized.
        The parameter is passed to numpy.zeros(). If unspecified, the NumPy
        default is used.
    Returns
    -------
    M : SciPy sparse array
        The attribute matrix.
    ordering : list
        If `rc_order` was specified, then only the matrix is returned.
        However, if `rc_order` was None, then the ordering used to construct
        the matrix is returned as well.
    Examples
    --------
    Construct an adjacency matrix:
    >>> G = nx.Graph()
    >>> G.add_edge(0, 1, thickness=1, weight=3)
    >>> G.add_edge(0, 2, thickness=2)
    >>> G.add_edge(1, 2, thickness=3)
    >>> M = nx.attr_sparse_matrix(G, rc_order=[0, 1, 2])
    >>> M.toarray()
    array([[0., 1., 1.],
           [1., 0., 1.],
           [1., 1., 0.]])
    Alternatively, we can obtain the matrix describing edge thickness.
    >>> M = nx.attr_sparse_matrix(G, edge_attr="thickness", rc_order=[0, 1, 2])
    >>> M.toarray()
    array([[0., 1., 2.],
           [1., 0., 3.],
           [2., 3., 0.]])
    We can also color the nodes and ask for the probability distribution over
    all edges (u,v) describing:
        Pr(v has color Y | u has color X)
    >>> G.nodes[0]["color"] = "red"
    >>> G.nodes[1]["color"] = "red"
    >>> G.nodes[2]["color"] = "blue"
    >>> rc = ["red", "blue"]
    >>> M = nx.attr_sparse_matrix(G, node_attr="color", normalized=True, rc_order=rc)
    >>> M.toarray()
    array([[0.33333333, 0.66666667],
           [1.        , 0.        ]])
    For example, the above tells us that for all edges (u,v):
        Pr( v is red  | u is red)  = 1/3
        Pr( v is blue | u is red)  = 2/3
        Pr( v is red  | u is blue) = 1
        Pr( v is blue | u is blue) = 0
    Finally, we can obtain the total weights listed by the node colors.
    >>> M = nx.attr_sparse_matrix(G, edge_attr="weight", node_attr="color", rc_order=rc)
    >>> M.toarray()
    array([[3., 2.],
           [2., 0.]])
    Thus, the total weight over all edges (u,v) with u and v having colors:
        (red, red)   is 3   # the sole contribution is from edge (0,1)
        (red, blue)  is 2   # contributions from edges (0,2) and (1,2)
        (blue, red)  is 2   # same as (red, blue) since graph is undirected
        (blue, blue) is 0   # there are no edges with blue endpoints
    """
    import numpy as np
    import scipy as sp
```

```
309: (4)                edge_value = _edge_value(G, edge_attr)
310: (4)                node_value = _node_value(G, node_attr)
311: (4)                if rc_order is None:
312: (8)                    ordering = list({node_value(n) for n in G})
313: (4)                else:
314: (8)                    ordering = rc_order
315: (4)                N = len(ordering)
316: (4)                undirected = not G.is_directed()
317: (4)                index = dict(zip(ordering, range(N)))
318: (4)                M = sp.sparse.lil_array((N, N), dtype=dtype)
319: (4)                seen = set()
320: (4)                for u, nbrdict in G.adjacency():
321: (8)                    for v in nbrdict:
322: (12)                       i, j = index[node_value(u)], index[node_value(v)]
323: (12)                       if v not in seen:
324: (16)                           M[i, j] += edge_value(u, v)
325: (16)                           if undirected:
326: (20)                               M[j, i] = M[i, j]
327: (8)                    if undirected:
328: (12)                       seen.add(u)
329: (4)                if normalized:
330: (8)                    M *= 1 / M.sum(axis=1)[:, np.newaxis]  # in-place mult preserves
sparse
331: (4)                if rc_order is None:
332: (8)                    return M, ordering
333: (4)                else:
334: (8)                    return M
```

File 488 - bethehessianmatrix.py:

```
1: (0)              """Bethe Hessian or deformed Laplacian matrix of graphs."""
2: (0)              import networkx as nx
3: (0)              from networkx.utils import not_implemented_for
4: (0)              __all__ = ["bethe_hessian_matrix"]
5: (0)              @not_implemented_for("directed")
6: (0)              @not_implemented_for("multigraph")
7: (0)              @nx._dispatchable
8: (0)              def bethe_hessian_matrix(G, r=None, nodelist=None):
9: (4)                  r"""Returns the Bethe Hessian matrix of G.
10: (4)                 The Bethe Hessian is a family of matrices parametrized by r, defined as
11: (4)                 H(r) = (r^2 - 1) I - r A + D where A is the adjacency matrix, D is the
12: (4)                 diagonal matrix of node degrees, and I is the identify matrix. It is equal
13: (4)                 to the graph laplacian when the regularizer r = 1.
14: (4)                 The default choice of regularizer should be the ratio [2]_
15: (4)                 .. math::
16: (6)                   r_m = \left(\sum k_i \right)^{-1}\left(\sum k_i^2 \right) - 1
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : Graph
20: (7)                    A NetworkX graph
21: (4)                 r : float
22: (7)                    Regularizer parameter
23: (4)                 nodelist : list, optional
24: (7)                    The rows and columns are ordered according to the nodes in nodelist.
25: (7)                    If nodelist is None, then the ordering is produced by ``G.nodes()``.
26: (4)                 Returns
27: (4)                 -------
28: (4)                 H : scipy.sparse.csr_array
29: (6)                   The Bethe Hessian matrix of `G`, with parameter `r`.
30: (4)                 Examples
31: (4)                 --------
32: (4)                 >>> k = [3, 2, 2, 1, 0]
33: (4)                 >>> G = nx.havel_hakimi_graph(k)
34: (4)                 >>> H = nx.bethe_hessian_matrix(G)
35: (4)                 >>> H.toarray()
36: (4)                 array([[ 3.5625, -1.25  , -1.25  , -1.25  ,  0.    ],
37: (11)                       [-1.25  ,  2.5625, -1.25  ,  0.    ,  0.    ],
```

```
38: (11)                        [-1.25  , -1.25  ,  2.5625,  0.    ,  0.    ],
39: (11)                        [-1.25  ,  0.    ,  0.    ,  1.5625,  0.    ],
40: (11)                        [ 0.    ,  0.    ,  0.    ,  0.    ,  0.5625]])
41: (4)                 See Also
42: (4)                 --------
43: (4)                 bethe_hessian_spectrum
44: (4)                 adjacency_matrix
45: (4)                 laplacian_matrix
46: (4)                 References
47: (4)                 ----------
48: (4)                 .. [1] A. Saade, F. Krzakala and L. Zdeborová
49: (7)                    "Spectral Clustering of Graphs with the Bethe Hessian",
50: (7)                    Advances in Neural Information Processing Systems, 2014.
51: (4)                 .. [2] C. M. Le, E. Levina
52: (7)                    "Estimating the number of communities in networks by spectral methods"
53: (7)                    arXiv:1507.00827, 2015.
54: (4)                 """
55: (4)                 import scipy as sp
56: (4)                 if nodelist is None:
57: (8)                     nodelist = list(G)
58: (4)                 if r is None:
59: (8)                     r = sum(d**2 for v, d in nx.degree(G)) / sum(d for v, d in
nx.degree(G)) - 1
60: (4)                 A = nx.to_scipy_sparse_array(G, nodelist=nodelist, format="csr")
61: (4)                 n, m = A.shape
62: (4)                 D = sp.sparse.csr_array(sp.sparse.spdiags(A.sum(axis=1), 0, m, n,
format="csr"))
63: (4)                 I = sp.sparse.csr_array(sp.sparse.eye(m, n, format="csr"))
64: (4)                 return (r**2 - 1) * I - r * A + D

----------------------------------------

File 489 - graphmatrix.py:

1: (0)              """
2: (0)              Adjacency matrix and incidence matrix of graphs.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = ["incidence_matrix", "adjacency_matrix"]
6: (0)              @nx._dispatchable(edge_attrs="weight")
7: (0)              def incidence_matrix(
8: (4)                  G, nodelist=None, edgelist=None, oriented=False, weight=None, *,
dtype=None
9: (0)              ):
10: (4)                 """Returns incidence matrix of G.
11: (4)                 The incidence matrix assigns each row to a node and each column to an
edge.
12: (4)                 For a standard incidence matrix a 1 appears wherever a row's node is
13: (4)                 incident on the column's edge.  For an oriented incidence matrix each
14: (4)                 edge is assigned an orientation (arbitrarily for undirected and aligning
to
15: (4)                 direction for directed).  A -1 appears for the source (tail) of an edge
and
16: (4)                 1 for the destination (head) of the edge.  The elements are zero
otherwise.
17: (4)                 Parameters
18: (4)                 ----------
19: (4)                 G : graph
20: (7)                    A NetworkX graph
21: (4)                 nodelist : list, optional   (default= all nodes in G)
22: (7)                    The rows are ordered according to the nodes in nodelist.
23: (7)                    If nodelist is None, then the ordering is produced by G.nodes().
24: (4)                 edgelist : list, optional (default= all edges in G)
25: (7)                    The columns are ordered according to the edges in edgelist.
26: (7)                    If edgelist is None, then the ordering is produced by G.edges().
27: (4)                 oriented: bool, optional (default=False)
28: (7)                    If True, matrix elements are +1 or -1 for the head or tail node
29: (7)                    respectively of each edge.  If False, +1 occurs at both nodes.
30: (4)                 weight : string or None, optional (default=None)
```

```
(7)                 The edge data key used to provide each value in the matrix.
(7)                 If None, then each edge has weight 1.  Edge weights, if used,
(7)                 should be positive so that the orientation can provide the sign.
(4)             dtype : a NumPy dtype or None (default=None)
(8)                 The dtype of the output sparse array. This type should be a compatible
(8)                 type of the weight argument, eg. if weight would return a float this
(8)                 argument should also be a float.
(8)                 If None, then the default for SciPy is used.
(4)             Returns
(4)             -------
(4)             A : SciPy sparse array
(6)               The incidence matrix of G.
(4)             Notes
(4)             -----
(4)             For MultiGraph/MultiDiGraph, the edges in edgelist should be
(4)             (u,v,key) 3-tuples.
(4)             "Networks are the best discrete model for so many problems in
(4)             applied mathematics" [1]_.
(4)             References
(4)             ----------
(4)             .. [1] Gil Strang, Network applications: A = incidence matrix,
(7)                http://videolectures.net/mit18085f07_strang_lec03/
(4)             """
(4)             import scipy as sp
(4)             if nodelist is None:
(8)                 nodelist = list(G)
(4)             if edgelist is None:
(8)                 if G.is_multigraph():
(12)                    edgelist = list(G.edges(keys=True))
(8)                 else:
(12)                    edgelist = list(G.edges())
(4)             A = sp.sparse.lil_array((len(nodelist), len(edgelist)), dtype=dtype)
(4)             node_index = {node: i for i, node in enumerate(nodelist)}
(4)             for ei, e in enumerate(edgelist):
(8)                 (u, v) = e[:2]
(8)                 if u == v:
(12)                    continue  # self loops give zero column
(8)                 try:
(12)                    ui = node_index[u]
(12)                    vi = node_index[v]
(8)                 except KeyError as err:
(12)                    raise nx.NetworkXError(
(16)                        f"node {u} or {v} in edgelist but not in nodelist"
(12)                    ) from err
(8)                 if weight is None:
(12)                    wt = 1
(8)                 else:
(12)                    if G.is_multigraph():
(16)                        ekey = e[2]
(16)                        wt = G[u][v][ekey].get(weight, 1)
(12)                    else:
(16)                        wt = G[u][v].get(weight, 1)
(8)                 if oriented:
(12)                    A[ui, ei] = -wt
(12)                    A[vi, ei] = wt
(8)                 else:
(12)                    A[ui, ei] = wt
(12)                    A[vi, ei] = wt
(4)             return A.asformat("csc")
(0)         @nx._dispatchable(edge_attrs="weight")
(0)         def adjacency_matrix(G, nodelist=None, dtype=None, weight="weight"):
(4)             """Returns adjacency matrix of G.
(4)             Parameters
(4)             ----------
(4)             G : graph
(7)                A NetworkX graph
(4)             nodelist : list, optional
(7)                The rows and columns are ordered according to the nodes in nodelist.
(7)                If nodelist is None, then the ordering is produced by G.nodes().
```

```
100: (4)                dtype : NumPy data-type, optional
101: (8)                    The desired data-type for the array.
102: (8)                    If None, then the NumPy default is used.
103: (4)                weight : string or None, optional (default='weight')
104: (7)                   The edge data key used to provide each value in the matrix.
105: (7)                   If None, then each edge has weight 1.
106: (4)                Returns
107: (4)                -------
108: (4)                A : SciPy sparse array
109: (6)                  Adjacency matrix representation of G.
110: (4)                Notes
111: (4)                -----
112: (4)                For directed graphs, entry i,j corresponds to an edge from i to j.
113: (4)                If you want a pure Python adjacency matrix representation try
114: (4)                networkx.convert.to_dict_of_dicts which will return a
115: (4)                dictionary-of-dictionaries format that can be addressed as a
116: (4)                sparse matrix.
117: (4)                For MultiGraph/MultiDiGraph with parallel edges the weights are summed.
118: (4)                See `to_numpy_array` for other options.
119: (4)                The convention used for self-loop edges in graphs is to assign the
120: (4)                diagonal matrix entry value to the edge weight attribute
121: (4)                (or the number 1 if the edge has no weight attribute).  If the
122: (4)                alternate convention of doubling the edge weight is desired the
123: (4)                resulting SciPy sparse array can be modified as follows:
124: (4)                >>> G = nx.Graph([(1, 1)])
125: (4)                >>> A = nx.adjacency_matrix(G)
126: (4)                >>> print(A.todense())
127: (4)                [[1]]
128: (4)                >>> A.setdiag(A.diagonal() * 2)
129: (4)                >>> print(A.todense())
130: (4)                [[2]]
131: (4)                See Also
132: (4)                --------
133: (4)                to_numpy_array
134: (4)                to_scipy_sparse_array
135: (4)                to_dict_of_dicts
136: (4)                adjacency_spectrum
137: (4)                """
138: (4)                return nx.to_scipy_sparse_array(G, nodelist=nodelist, dtype=dtype,
weight=weight)

----------------------------------------

File 490 - laplacianmatrix.py:

1: (0)              """Laplacian matrix of graphs.
2: (0)              All calculations here are done using the out-degree. For Laplacians using
3: (0)              in-degree, use `G.reverse(copy=False)` instead of `G` and take the transpose.
4: (0)              The `laplacian_matrix` function provides an unnormalized matrix,
5: (0)              while `normalized_laplacian_matrix`, `directed_laplacian_matrix`,
6: (0)              and `directed_combinatorial_laplacian_matrix` are all normalized.
7: (0)              """
8: (0)              import networkx as nx
9: (0)              from networkx.utils import not_implemented_for
10: (0)             __all__ = [
11: (4)                 "laplacian_matrix",
12: (4)                 "normalized_laplacian_matrix",
13: (4)                 "total_spanning_tree_weight",
14: (4)                 "directed_laplacian_matrix",
15: (4)                 "directed_combinatorial_laplacian_matrix",
16: (0)             ]
17: (0)             @nx._dispatchable(edge_attrs="weight")
18: (0)             def laplacian_matrix(G, nodelist=None, weight="weight"):
19: (4)                 """Returns the Laplacian matrix of G.
20: (4)                 The graph Laplacian is the matrix L = D - A, where
21: (4)                 A is the adjacency matrix and D is the diagonal matrix of node degrees.
22: (4)                 Parameters
23: (4)                 ----------
24: (4)                 G : graph
```

```
 25: (7)                    A NetworkX graph
 26: (4)                 nodelist : list, optional
 27: (7)                    The rows and columns are ordered according to the nodes in nodelist.
 28: (7)                    If nodelist is None, then the ordering is produced by G.nodes().
 29: (4)                 weight : string or None, optional (default='weight')
 30: (7)                    The edge data key used to compute each value in the matrix.
 31: (7)                    If None, then each edge has weight 1.
 32: (4)                 Returns
 33: (4)                 -------
 34: (4)                 L : SciPy sparse array
 35: (6)                   The Laplacian matrix of G.
 36: (4)                 Notes
 37: (4)                 -----
 38: (4)                 For MultiGraph, the edges weights are summed.
 39: (4)                 This returns an unnormalized matrix. For a normalized output,
 40: (4)                 use `normalized_laplacian_matrix`, `directed_laplacian_matrix`,
 41: (4)                 or `directed_combinatorial_laplacian_matrix`.
 42: (4)                 This calculation uses the out-degree of the graph `G`. To use the
 43: (4)                 in-degree for calculations instead, use `G.reverse(copy=False)` and
 44: (4)                 take the transpose.
 45: (4)                 See Also
 46: (4)                 --------
 47: (4)                 :func:`~networkx.convert_matrix.to_numpy_array`
 48: (4)                 normalized_laplacian_matrix
 49: (4)                 directed_laplacian_matrix
 50: (4)                 directed_combinatorial_laplacian_matrix
 51: (4)                 :func:`~networkx.linalg.spectrum.laplacian_spectrum`
 52: (4)                 Examples
 53: (4)                 --------
 54: (4)                 For graphs with multiple connected components, L is permutation-similar
 55: (4)                 to a block diagonal matrix where each block is the respective Laplacian
 56: (4)                 matrix for each component.
 57: (4)                 >>> G = nx.Graph([(1, 2), (2, 3), (4, 5)])
 58: (4)                 >>> print(nx.laplacian_matrix(G).toarray())
 59: (4)                 [[ 1 -1  0  0  0]
 60: (5)                  [-1  2 -1  0  0]
 61: (5)                  [ 0 -1  1  0  0]
 62: (5)                  [ 0  0  0  1 -1]
 63: (5)                  [ 0  0  0 -1  1]]
 64: (4)                 >>> edges = [
 65: (4)                 ...     (1, 2),
 66: (4)                 ...     (2, 1),
 67: (4)                 ...     (2, 4),
 68: (4)                 ...     (4, 3),
 69: (4)                 ...     (3, 4),
 70: (4)                 ... ]
 71: (4)                 >>> DiG = nx.DiGraph(edges)
 72: (4)                 >>> print(nx.laplacian_matrix(DiG).toarray())
 73: (4)                 [[ 1 -1  0  0]
 74: (5)                  [-1  2 -1  0]
 75: (5)                  [ 0  0  1 -1]
 76: (5)                  [ 0  0 -1  1]]
 77: (4)                 Notice that node 4 is represented by the third column and row. This is
because
 78: (4)                 by default the row/column order is the order of `G.nodes` (i.e. the node
added
 79: (4)                 order -- in the edgelist, 4 first appears in (2, 4), before node 3 in edge
(4, 3).)
 80: (4)                 To control the node order of the matrix, use the `nodelist` argument.
 81: (4)                 >>> print(nx.laplacian_matrix(DiG, nodelist=[1, 2, 3, 4]).toarray())
 82: (4)                 [[ 1 -1  0  0]
 83: (5)                  [-1  2  0 -1]
 84: (5)                  [ 0  0  1 -1]
 85: (5)                  [ 0  0 -1  1]]
 86: (4)                 This calculation uses the out-degree of the graph `G`. To use the
 87: (4)                 in-degree for calculations instead, use `G.reverse(copy=False)` and
 88: (4)                 take the transpose.
 89: (4)                 >>> print(nx.laplacian_matrix(DiG.reverse(copy=False)).toarray().T)
 90: (4)                 [[ 1 -1  0  0]
```

```
 91: (5)                 [-1  1 -1  0]
 92: (5)                 [ 0  0  2 -1]
 93: (5)                 [ 0  0 -1  1]]
 94: (4)                References
 95: (4)                ----------
 96: (4)                .. [1] Langville, Amy N., and Carl D. Meyer. Google’s PageRank and Beyond:
 97: (7)                   The Science of Search Engine Rankings. Princeton University Press,
 2006.
 98: (4)                """
 99: (4)                import scipy as sp
 100: (4)               if nodelist is None:
 101: (8)                   nodelist = list(G)
 102: (4)               A = nx.to_scipy_sparse_array(G, nodelist=nodelist, weight=weight,
 format="csr")
 103: (4)               n, m = A.shape
 104: (4)               D = sp.sparse.csr_array(sp.sparse.spdiags(A.sum(axis=1), 0, m, n,
 format="csr"))
 105: (4)               return D - A
 106: (0)            @nx._dispatchable(edge_attrs="weight")
 107: (0)            def normalized_laplacian_matrix(G, nodelist=None, weight="weight"):
 108: (4)               r"""Returns the normalized Laplacian matrix of G.
 109: (4)               The normalized graph Laplacian is the matrix
 110: (4)               .. math::
 111: (8)                   N = D^{-1/2} L D^{-1/2}
 112: (4)               where `L` is the graph Laplacian and `D` is the diagonal matrix of
 113: (4)               node degrees [1]_.
 114: (4)               Parameters
 115: (4)               ----------
 116: (4)               G : graph
 117: (7)                  A NetworkX graph
 118: (4)               nodelist : list, optional
 119: (7)                  The rows and columns are ordered according to the nodes in nodelist.
 120: (7)                  If nodelist is None, then the ordering is produced by G.nodes().
 121: (4)               weight : string or None, optional (default='weight')
 122: (7)                  The edge data key used to compute each value in the matrix.
 123: (7)                  If None, then each edge has weight 1.
 124: (4)               Returns
 125: (4)               -------
 126: (4)               N : SciPy sparse array
 127: (6)                 The normalized Laplacian matrix of G.
 128: (4)               Notes
 129: (4)               -----
 130: (4)               For MultiGraph, the edges weights are summed.
 131: (4)               See :func:`to_numpy_array` for other options.
 132: (4)               If the Graph contains selfloops, D is defined as ``diag(sum(A, 1))``,
 where A is
 133: (4)               the adjacency matrix [2]_.
 134: (4)               This calculation uses the out-degree of the graph `G`. To use the
 135: (4)               in-degree for calculations instead, use `G.reverse(copy=False)` and
 136: (4)               take the transpose.
 137: (4)               For an unnormalized output, use `laplacian_matrix`.
 138: (4)               Examples
 139: (4)               --------
 140: (4)               >>> import numpy as np
 141: (4)               >>> edges = [
 142: (4)               ...     (1, 2),
 143: (4)               ...     (2, 1),
 144: (4)               ...     (2, 4),
 145: (4)               ...     (4, 3),
 146: (4)               ...     (3, 4),
 147: (4)               ... ]
 148: (4)               >>> DiG = nx.DiGraph(edges)
 149: (4)               >>> print(nx.normalized_laplacian_matrix(DiG).toarray())
 150: (4)               [[ 1.         -0.70710678  0.          0.        ]
 151: (5)                [-0.70710678  1.         -0.70710678  0.        ]
 152: (5)                [ 0.          0.          1.         -1.        ]
 153: (5)                [ 0.          0.         -1.          1.        ]]
 154: (4)               Notice that node 4 is represented by the third column and row. This is
 because
```

```
155: (4)                by default the row/column order is the order of `G.nodes` (i.e. the node
added
156: (4)                order -- in the edgelist, 4 first appears in (2, 4), before node 3 in edge
(4, 3).)
157: (4)                To control the node order of the matrix, use the `nodelist` argument.
158: (4)                >>> print(nx.normalized_laplacian_matrix(DiG, nodelist=[1, 2, 3,
4]).toarray())
159: (4)                [[ 1.         -0.70710678  0.          0.        ]
160: (5)                 [-0.70710678  1.          0.         -0.70710678]
161: (5)                 [ 0.          0.          1.         -1.        ]
162: (5)                 [ 0.          0.         -1.          1.        ]]
163: (4)                >>> G = nx.Graph(edges)
164: (4)                >>> print(nx.normalized_laplacian_matrix(G).toarray())
165: (4)                [[ 1.         -0.70710678  0.          0.        ]
166: (5)                 [-0.70710678  1.         -0.5         0.        ]
167: (5)                 [ 0.         -0.5         1.         -0.70710678]
168: (5)                 [ 0.          0.         -0.70710678  1.        ]]
169: (4)                See Also
170: (4)                --------
171: (4)                laplacian_matrix
172: (4)                normalized_laplacian_spectrum
173: (4)                directed_laplacian_matrix
174: (4)                directed_combinatorial_laplacian_matrix
175: (4)                References
176: (4)                ----------
177: (4)                .. [1] Fan Chung-Graham, Spectral Graph Theory,
178: (7)                   CBMS Regional Conference Series in Mathematics, Number 92, 1997.
179: (4)                .. [2] Steve Butler, Interlacing For Weighted Graphs Using The Normalized
180: (7)                   Laplacian, Electronic Journal of Linear Algebra, Volume 16, pp. 90-98,
181: (7)                   March 2007.
182: (4)                .. [3] Langville, Amy N., and Carl D. Meyer. Google’s PageRank and Beyond:
183: (7)                   The Science of Search Engine Rankings. Princeton University Press,
2006.
184: (4)                """
185: (4)                import numpy as np
186: (4)                import scipy as sp
187: (4)                if nodelist is None:
188: (8)                    nodelist = list(G)
189: (4)                A = nx.to_scipy_sparse_array(G, nodelist=nodelist, weight=weight,
format="csr")
190: (4)                n, _ = A.shape
191: (4)                diags = A.sum(axis=1)
192: (4)                D = sp.sparse.csr_array(sp.sparse.spdiags(diags, 0, n, n, format="csr"))
193: (4)                L = D - A
194: (4)                with np.errstate(divide="ignore"):
195: (8)                    diags_sqrt = 1.0 / np.sqrt(diags)
196: (4)                diags_sqrt[np.isinf(diags_sqrt)] = 0
197: (4)                DH = sp.sparse.csr_array(sp.sparse.spdiags(diags_sqrt, 0, n, n,
format="csr"))
198: (4)                return DH @ (L @ DH)
199: (0)            @nx._dispatchable(edge_attrs="weight")
200: (0)            def total_spanning_tree_weight(G, weight=None, root=None):
201: (4)                """
202: (4)                Returns the total weight of all spanning trees of `G`.
203: (4)                Kirchoff's Tree Matrix Theorem [1]_, [2]_ states that the determinant of
any
204: (4)                cofactor of the Laplacian matrix of a graph is the number of spanning
trees
205: (4)                in the graph. For a weighted Laplacian matrix, it is the sum across all
206: (4)                spanning trees of the multiplicative weight of each tree. That is, the
207: (4)                weight of each tree is the product of its edge weights.
208: (4)                For unweighted graphs, the total weight equals the number of spanning
trees in `G`.
209: (4)                For directed graphs, the total weight follows by summing over all directed
210: (4)                spanning trees in `G` that start in the `root` node [3]_.
211: (4)                .. deprecated:: 3.3
212: (7)                   ``total_spanning_tree_weight`` is deprecated and will be removed in
v3.5.
213: (7)                   Use ``nx.number_of_spanning_trees(G)`` instead.
```

```
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX Graph
(4)                 weight : string or None, optional (default=None)
(8)                     The key for the edge attribute holding the edge weight.
(8)                     If None, then each edge has weight 1.
(4)                 root : node (only required for directed graphs)
(7)                    A node in the directed graph `G`.
(4)                 Returns
(4)                 -------
(4)                 total_weight : float
(8)                     Undirected graphs:
(12)                        The sum of the total multiplicative weights for all spanning trees in `G`.
(8)                     Directed graphs:
(12)                        The sum of the total multiplicative weights for all spanning trees of `G`,
(12)                        rooted at node `root`.
(4)                 Raises
(4)                 ------
(4)                 NetworkXPointlessConcept
(8)                     If `G` does not contain any nodes.
(4)                 NetworkXError
(8)                     If the graph `G` is not (weakly) connected,
(8)                     or if `G` is directed and the root node is not specified or not in G.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.complete_graph(5)
(4)                 >>> round(nx.total_spanning_tree_weight(G))
(4)                 125
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edge(1, 2, weight=2)
(4)                 >>> G.add_edge(1, 3, weight=1)
(4)                 >>> G.add_edge(2, 3, weight=1)
(4)                 >>> round(nx.total_spanning_tree_weight(G, "weight"))
(4)                 5
(4)                 Notes
(4)                 -----
(4)                 Self-loops are excluded. Multi-edges are contracted in one edge
(4)                 equal to the sum of the weights.
(4)                 References
(4)                 ----------
(4)                 .. [1] Wikipedia
(7)                    "Kirchhoff's theorem."
(7)                    https://en.wikipedia.org/wiki/Kirchhoff%27s_theorem
(4)                 .. [2] Kirchhoff, G. R.
(8)                     Über die Auflösung der Gleichungen, auf welche man
(8)                     bei der Untersuchung der linearen Vertheilung
(8)                     Galvanischer Ströme geführt wird
(8)                     Annalen der Physik und Chemie, vol. 72, pp. 497-508, 1847.
(4)                 .. [3] Margoliash, J.
(8)                     "Matrix-Tree Theorem for Directed Graphs"
(8)                 https://www.math.uchicago.edu/~may/VIGRE/VIGRE2010/REUPapers/Margoliash.pdf
(4)                 """
(4)                 import warnings
(4)                 warnings.warn(
(8)                     (
(12)                        "\n\ntotal_spanning_tree_weight is deprecated and will be removed in v3.5.\n"
(12)                        "Use `nx.number_of_spanning_trees(G)` instead."
(8)                     ),
(8)                     category=DeprecationWarning,
(8)                     stacklevel=3,
(4)                 )
(4)                 return nx.number_of_spanning_trees(G, weight=weight, root=root)
(0)             @not_implemented_for("undirected")
(0)             @not_implemented_for("multigraph")
(0)             @nx._dispatchable(edge_attrs="weight")
```

```
(0)            def directed_laplacian_matrix(
(4)                G, nodelist=None, weight="weight", walk_type=None, alpha=0.95
(0)            ):
(4)                r"""Returns the directed Laplacian matrix of G.
(4)                The graph directed Laplacian is the matrix
(4)                .. math::
(8)                    L = I - \frac{1}{2} \left (\Phi^{1/2} P \Phi^{-1/2} + \Phi^{-1/2} P^T \Phi^{1/2} \right )
(4)                where `I` is the identity matrix, `P` is the transition matrix of the
(4)                graph, and `\Phi` a matrix with the Perron vector of `P` in the diagonal and
(4)                zeros elsewhere [1]_.
(4)                Depending on the value of walk_type, `P` can be the transition matrix
(4)                induced by a random walk, a lazy random walk, or a random walk with
(4)                teleportation (PageRank).
(4)                Parameters
(4)                ----------
(4)                G : DiGraph
(7)                   A NetworkX graph
(4)                nodelist : list, optional
(7)                   The rows and columns are ordered according to the nodes in nodelist.
(7)                   If nodelist is None, then the ordering is produced by G.nodes().
(4)                weight : string or None, optional (default='weight')
(7)                   The edge data key used to compute each value in the matrix.
(7)                   If None, then each edge has weight 1.
(4)                walk_type : string or None, optional (default=None)
(7)                   One of ``"random"``, ``"lazy"``, or ``"pagerank"``. If ``walk_type=None``
(7)                   (the default), then a value is selected according to the properties of `G`:
(7)                   - ``walk_type="random"`` if `G` is strongly connected and aperiodic
(7)                   - ``walk_type="lazy"`` if `G` is strongly connected but not aperiodic
(7)                   - ``walk_type="pagerank"`` for all other cases.
(4)                alpha : real
(7)                   (1 - alpha) is the teleportation probability used with pagerank
(4)                Returns
(4)                -------
(4)                L : NumPy matrix
(6)                  Normalized Laplacian of G.
(4)                Notes
(4)                -----
(4)                Only implemented for DiGraphs
(4)                The result is always a symmetric matrix.
(4)                This calculation uses the out-degree of the graph `G`. To use the
(4)                in-degree for calculations instead, use `G.reverse(copy=False)` and
(4)                take the transpose.
(4)                See Also
(4)                --------
(4)                laplacian_matrix
(4)                normalized_laplacian_matrix
(4)                directed_combinatorial_laplacian_matrix
(4)                References
(4)                ----------
(4)                .. [1] Fan Chung (2005).
(7)                   Laplacians and the Cheeger inequality for directed graphs.
(7)                   Annals of Combinatorics, 9(1), 2005
(4)                """
(4)                import numpy as np
(4)                import scipy as sp
(4)                P = _transition_matrix(
(8)                    G, nodelist=nodelist, weight=weight, walk_type=walk_type, alpha=alpha
(4)                )
(4)                n, m = P.shape
(4)                evals, evecs = sp.sparse.linalg.eigs(P.T, k=1)
(4)                v = evecs.flatten().real
(4)                p = v / v.sum()
(4)                sqrtp = np.sqrt(np.abs(p))
(4)                Q = (
(8)                    sp.sparse.csr_array(sp.sparse.spdiags(sqrtp, 0, n, n))
```

```
344: (8)                    @ P
345: (8)                    @ sp.sparse.csr_array(sp.sparse.spdiags(1.0 / sqrtp, 0, n, n))
346: (4)                )
347: (4)                I = np.identity(len(G))
348: (4)                return I - (Q + Q.T) / 2.0
349: (0)            @not_implemented_for("undirected")
350: (0)            @not_implemented_for("multigraph")
351: (0)            @nx._dispatchable(edge_attrs="weight")
352: (0)            def directed_combinatorial_laplacian_matrix(
353: (4)                G, nodelist=None, weight="weight", walk_type=None, alpha=0.95
354: (0)            ):
355: (4)                r"""Return the directed combinatorial Laplacian matrix of G.
356: (4)                The graph directed combinatorial Laplacian is the matrix
357: (4)                .. math::
358: (8)                    L = \Phi - \frac{1}{2} \left (\Phi P + P^T \Phi \right)
359: (4)                where `P` is the transition matrix of the graph and `\Phi` a matrix
360: (4)                with the Perron vector of `P` in the diagonal and zeros elsewhere [1]_.
361: (4)                Depending on the value of walk_type, `P` can be the transition matrix
362: (4)                induced by a random walk, a lazy random walk, or a random walk with
363: (4)                teleportation (PageRank).
364: (4)                Parameters
365: (4)                ----------
366: (4)                G : DiGraph
367: (7)                   A NetworkX graph
368: (4)                nodelist : list, optional
369: (7)                   The rows and columns are ordered according to the nodes in nodelist.
370: (7)                   If nodelist is None, then the ordering is produced by G.nodes().
371: (4)                weight : string or None, optional (default='weight')
372: (7)                   The edge data key used to compute each value in the matrix.
373: (7)                   If None, then each edge has weight 1.
374: (4)                walk_type : string or None, optional (default=None)
375: (8)                    One of ``"random"``, ``"lazy"``, or ``"pagerank"``. If
``walk_type=None``
376: (8)                    (the default), then a value is selected according to the properties of
`G`:
377: (8)                    - ``walk_type="random"`` if `G` is strongly connected and aperiodic
378: (8)                    - ``walk_type="lazy"`` if `G` is strongly connected but not aperiodic
379: (8)                    - ``walk_type="pagerank"`` for all other cases.
380: (4)                alpha : real
381: (7)                   (1 - alpha) is the teleportation probability used with pagerank
382: (4)                Returns
383: (4)                -------
384: (4)                L : NumPy matrix
385: (6)                  Combinatorial Laplacian of G.
386: (4)                Notes
387: (4)                -----
388: (4)                Only implemented for DiGraphs
389: (4)                The result is always a symmetric matrix.
390: (4)                This calculation uses the out-degree of the graph `G`. To use the
391: (4)                in-degree for calculations instead, use `G.reverse(copy=False)` and
392: (4)                take the transpose.
393: (4)                See Also
394: (4)                --------
395: (4)                laplacian_matrix
396: (4)                normalized_laplacian_matrix
397: (4)                directed_laplacian_matrix
398: (4)                References
399: (4)                ----------
400: (4)                .. [1] Fan Chung (2005).
401: (7)                   Laplacians and the Cheeger inequality for directed graphs.
402: (7)                   Annals of Combinatorics, 9(1), 2005
403: (4)                """
404: (4)                import scipy as sp
405: (4)                P = _transition_matrix(
406: (8)                    G, nodelist=nodelist, weight=weight, walk_type=walk_type, alpha=alpha
407: (4)                )
408: (4)                n, m = P.shape
409: (4)                evals, evecs = sp.sparse.linalg.eigs(P.T, k=1)
410: (4)                v = evecs.flatten().real
```

```
(4)                 p = v / v.sum()
(4)                 Phi = sp.sparse.csr_array(sp.sparse.spdiags(p, 0, n, n)).toarray()
(4)                 return Phi - (Phi @ P + P.T @ Phi) / 2.0
(0)             def _transition_matrix(G, nodelist=None, weight="weight", walk_type=None,
alpha=0.95):
(4)                 """Returns the transition matrix of G.
(4)                 This is a row stochastic giving the transition probabilities while
(4)                 performing a random walk on the graph. Depending on the value of
walk_type,
(4)                 P can be the transition matrix induced by a random walk, a lazy random
walk,
(4)                 or a random walk with teleportation (PageRank).
(4)                 Parameters
(4)                 ----------
(4)                 G : DiGraph
(7)                    A NetworkX graph
(4)                 nodelist : list, optional
(7)                    The rows and columns are ordered according to the nodes in nodelist.
(7)                    If nodelist is None, then the ordering is produced by G.nodes().
(4)                 weight : string or None, optional (default='weight')
(7)                    The edge data key used to compute each value in the matrix.
(7)                    If None, then each edge has weight 1.
(4)                 walk_type : string or None, optional (default=None)
(7)                    One of ``"random"``, ``"lazy"``, or ``"pagerank"``. If
``walk_type=None``
(7)                    (the default), then a value is selected according to the properties of
`G`:
(8)                     - ``walk_type="random"`` if `G` is strongly connected and aperiodic
(8)                     - ``walk_type="lazy"`` if `G` is strongly connected but not aperiodic
(8)                     - ``walk_type="pagerank"`` for all other cases.
(4)                 alpha : real
(7)                    (1 - alpha) is the teleportation probability used with pagerank
(4)                 Returns
(4)                 -------
(4)                 P : numpy.ndarray
(6)                   transition matrix of G.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If walk_type not specified or alpha not in valid range
(4)                 """
(4)                 import numpy as np
(4)                 import scipy as sp
(4)                 if walk_type is None:
(8)                     if nx.is_strongly_connected(G):
(12)                        if nx.is_aperiodic(G):
(16)                            walk_type = "random"
(12)                        else:
(16)                            walk_type = "lazy"
(8)                     else:
(12)                        walk_type = "pagerank"
(4)                 A = nx.to_scipy_sparse_array(G, nodelist=nodelist, weight=weight,
dtype=float)
(4)                 n, m = A.shape
(4)                 if walk_type in ["random", "lazy"]:
(8)                     DI = sp.sparse.csr_array(sp.sparse.spdiags(1.0 / A.sum(axis=1), 0, n,
n))
(8)                     if walk_type == "random":
(12)                        P = DI @ A
(8)                     else:
(12)                        I = sp.sparse.csr_array(sp.sparse.identity(n))
(12)                        P = (I + DI @ A) / 2.0
(4)                 elif walk_type == "pagerank":
(8)                     if not (0 < alpha < 1):
(12)                        raise nx.NetworkXError("alpha must be between 0 and 1")
(8)                     A = A.toarray()
(8)                     A[A.sum(axis=1) == 0, :] = 1 / n
(8)                     A = A / A.sum(axis=1)[np.newaxis, :].T
(8)                     P = alpha * A + (1 - alpha) / n
```

```
473: (4)                else:
474: (8)                    raise nx.NetworkXError("walk_type must be random, lazy, or pagerank")
475: (4)                return P
```

File 491 - spectrum.py:

```
1: (0)              """
2: (0)              Eigenvalue spectrum of graphs.
3: (0)              """
4: (0)              import networkx as nx
5: (0)              __all__ = [
6: (4)                  "laplacian_spectrum",
7: (4)                  "adjacency_spectrum",
8: (4)                  "modularity_spectrum",
9: (4)                  "normalized_laplacian_spectrum",
10: (4)                 "bethe_hessian_spectrum",
11: (0)             ]
12: (0)             @nx._dispatchable(edge_attrs="weight")
13: (0)             def laplacian_spectrum(G, weight="weight"):
14: (4)                 """Returns eigenvalues of the Laplacian of G
15: (4)                 Parameters
16: (4)                 ----------
17: (4)                 G : graph
18: (7)                    A NetworkX graph
19: (4)                 weight : string or None, optional (default='weight')
20: (7)                    The edge data key used to compute each value in the matrix.
21: (7)                    If None, then each edge has weight 1.
22: (4)                 Returns
23: (4)                 -------
24: (4)                 evals : NumPy array
25: (6)                   Eigenvalues
26: (4)                 Notes
27: (4)                 -----
28: (4)                 For MultiGraph/MultiDiGraph, the edges weights are summed.
29: (4)                 See :func:`~networkx.convert_matrix.to_numpy_array` for other options.
30: (4)                 See Also
31: (4)                 --------
32: (4)                 laplacian_matrix
33: (4)                 Examples
34: (4)                 --------
35: (4)                 The multiplicity of 0 as an eigenvalue of the laplacian matrix is equal
36: (4)                 to the number of connected components of G.
37: (4)                 >>> G = nx.Graph()  # Create a graph with 5 nodes and 3 connected
components
38: (4)                 >>> G.add_nodes_from(range(5))
39: (4)                 >>> G.add_edges_from([(0, 2), (3, 4)])
40: (4)                 >>> nx.laplacian_spectrum(G)
41: (4)                 array([0., 0., 0., 2., 2.])
42: (4)                 """
43: (4)                 import scipy as sp
44: (4)                 return sp.linalg.eigvalsh(nx.laplacian_matrix(G, weight=weight).todense())
45: (0)             @nx._dispatchable(edge_attrs="weight")
46: (0)             def normalized_laplacian_spectrum(G, weight="weight"):
47: (4)                 """Return eigenvalues of the normalized Laplacian of G
48: (4)                 Parameters
49: (4)                 ----------
50: (4)                 G : graph
51: (7)                    A NetworkX graph
52: (4)                 weight : string or None, optional (default='weight')
53: (7)                    The edge data key used to compute each value in the matrix.
54: (7)                    If None, then each edge has weight 1.
55: (4)                 Returns
56: (4)                 -------
57: (4)                 evals : NumPy array
58: (6)                   Eigenvalues
59: (4)                 Notes
60: (4)                 -----
```

```
(4)                 For MultiGraph/MultiDiGraph, the edges weights are summed.
(4)                 See to_numpy_array for other options.
(4)                 See Also
(4)                 --------
(4)                 normalized_laplacian_matrix
(4)                 """
(4)                 import scipy as sp
(4)                 return sp.linalg.eigvalsh(
(8)                     nx.normalized_laplacian_matrix(G, weight=weight).todense()
(4)                 )
(0)             @nx._dispatchable(edge_attrs="weight")
(0)             def adjacency_spectrum(G, weight="weight"):
(4)                 """Returns eigenvalues of the adjacency matrix of G.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph
(4)                 weight : string or None, optional (default='weight')
(7)                    The edge data key used to compute each value in the matrix.
(7)                    If None, then each edge has weight 1.
(4)                 Returns
(4)                 -------
(4)                 evals : NumPy array
(6)                   Eigenvalues
(4)                 Notes
(4)                 -----
(4)                 For MultiGraph/MultiDiGraph, the edges weights are summed.
(4)                 See to_numpy_array for other options.
(4)                 See Also
(4)                 --------
(4)                 adjacency_matrix
(4)                 """
(4)                 import scipy as sp
(4)                 return sp.linalg.eigvals(nx.adjacency_matrix(G, weight=weight).todense())
(0)             @nx._dispatchable
(0)             def modularity_spectrum(G):
(4)                 """Returns eigenvalues of the modularity matrix of G.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph
(7)                    A NetworkX Graph or DiGraph
(4)                 Returns
(4)                 -------
(4)                 evals : NumPy array
(6)                   Eigenvalues
(4)                 See Also
(4)                 --------
(4)                 modularity_matrix
(4)                 References
(4)                 ----------
(4)                 .. [1] M. E. J. Newman, "Modularity and community structure in networks",
(7)                    Proc. Natl. Acad. Sci. USA, vol. 103, pp. 8577-8582, 2006.
(4)                 """
(4)                 import scipy as sp
(4)                 if G.is_directed():
(8)                     return sp.linalg.eigvals(nx.directed_modularity_matrix(G))
(4)                 else:
(8)                     return sp.linalg.eigvals(nx.modularity_matrix(G))
(0)             @nx._dispatchable
(0)             def bethe_hessian_spectrum(G, r=None):
(4)                 """Returns eigenvalues of the Bethe Hessian matrix of G.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph
(7)                    A NetworkX Graph or DiGraph
(4)                 r : float
(7)                    Regularizer parameter
(4)                 Returns
(4)                 -------
```

```
130: (4)                evals : NumPy array
131: (6)                  Eigenvalues
132: (4)                See Also
133: (4)                --------
134: (4)                bethe_hessian_matrix
135: (4)                References
136: (4)                ----------
137: (4)                .. [1] A. Saade, F. Krzakala and L. Zdeborová
138: (7)                   "Spectral clustering of graphs with the bethe hessian",
139: (7)                   Advances in Neural Information Processing Systems. 2014.
140: (4)                """
141: (4)                import scipy as sp
142: (4)                return sp.linalg.eigvalsh(nx.bethe_hessian_matrix(G, r).todense())

----------------------------------------

File 492 - modularitymatrix.py:

1: (0)              """Modularity matrix of graphs.
2: (0)              """
3: (0)              import networkx as nx
4: (0)              from networkx.utils import not_implemented_for
5: (0)              __all__ = ["modularity_matrix", "directed_modularity_matrix"]
6: (0)              @not_implemented_for("directed")
7: (0)              @not_implemented_for("multigraph")
8: (0)              @nx._dispatchable(edge_attrs="weight")
9: (0)              def modularity_matrix(G, nodelist=None, weight=None):
10: (4)                 r"""Returns the modularity matrix of G.
11: (4)                 The modularity matrix is the matrix B = A - <A>, where A is the adjacency
12: (4)                 matrix and <A> is the average adjacency matrix, assuming that the graph
13: (4)                 is described by the configuration model.
14: (4)                 More specifically, the element B_ij of B is defined as
15: (4)                 .. math::
16: (8)                     A_{ij} - {k_i k_j \over 2 m}
17: (4)                 where k_i is the degree of node i, and where m is the number of edges
18: (4)                 in the graph. When weight is set to a name of an attribute edge, Aij, k_i,
19: (4)                 k_j and m are computed using its value.
20: (4)                 Parameters
21: (4)                 ----------
22: (4)                 G : Graph
23: (7)                    A NetworkX graph
24: (4)                 nodelist : list, optional
25: (7)                    The rows and columns are ordered according to the nodes in nodelist.
26: (7)                    If nodelist is None, then the ordering is produced by G.nodes().
27: (4)                 weight : string or None, optional (default=None)
28: (7)                    The edge attribute that holds the numerical value used for
29: (7)                    the edge weight.  If None then all edge weights are 1.
30: (4)                 Returns
31: (4)                 -------
32: (4)                 B : Numpy array
33: (6)                   The modularity matrix of G.
34: (4)                 Examples
35: (4)                 --------
36: (4)                 >>> k = [3, 2, 2, 1, 0]
37: (4)                 >>> G = nx.havel_hakimi_graph(k)
38: (4)                 >>> B = nx.modularity_matrix(G)
39: (4)                 See Also
40: (4)                 --------
41: (4)                 to_numpy_array
42: (4)                 modularity_spectrum
43: (4)                 adjacency_matrix
44: (4)                 directed_modularity_matrix
45: (4)                 References
46: (4)                 ----------
47: (4)                 .. [1] M. E. J. Newman, "Modularity and community structure in networks",
48: (11)                       Proc. Natl. Acad. Sci. USA, vol. 103, pp. 8577-8582, 2006.
49: (4)                 """
50: (4)                 import numpy as np
51: (4)                 if nodelist is None:
```

```
52: (8)                     nodelist = list(G)
53: (4)                 A = nx.to_scipy_sparse_array(G, nodelist=nodelist, weight=weight,
format="csr")
54: (4)                 k = A.sum(axis=1)
55: (4)                 m = k.sum() * 0.5
56: (4)                 X = np.outer(k, k) / (2 * m)
57: (4)                 return A - X
58: (0)             @not_implemented_for("undirected")
59: (0)             @not_implemented_for("multigraph")
60: (0)             @nx._dispatchable(edge_attrs="weight")
61: (0)             def directed_modularity_matrix(G, nodelist=None, weight=None):
62: (4)                 """Returns the directed modularity matrix of G.
63: (4)                 The modularity matrix is the matrix B = A - <A>, where A is the adjacency
64: (4)                 matrix and <A> is the expected adjacency matrix, assuming that the graph
65: (4)                 is described by the configuration model.
66: (4)                 More specifically, the element B_ij of B is defined as
67: (4)                 .. math::
68: (8)                     B_{ij} = A_{ij} - k_i^{out} k_j^{in} / m
69: (4)                 where :math:`k_i^{in}` is the in degree of node i, and :math:`k_j^{out}`
is the out degree
70: (4)                 of node j, with m the number of edges in the graph. When weight is set
71: (4)                 to a name of an attribute edge, Aij, k_i, k_j and m are computed using
72: (4)                 its value.
73: (4)                 Parameters
74: (4)                 ----------
75: (4)                 G : DiGraph
76: (7)                    A NetworkX DiGraph
77: (4)                 nodelist : list, optional
78: (7)                    The rows and columns are ordered according to the nodes in nodelist.
79: (7)                    If nodelist is None, then the ordering is produced by G.nodes().
80: (4)                 weight : string or None, optional (default=None)
81: (7)                    The edge attribute that holds the numerical value used for
82: (7)                    the edge weight.  If None then all edge weights are 1.
83: (4)                 Returns
84: (4)                 -------
85: (4)                 B : Numpy array
86: (6)                   The modularity matrix of G.
87: (4)                 Examples
88: (4)                 --------
89: (4)                 >>> G = nx.DiGraph()
90: (4)                 >>> G.add_edges_from(
91: (4)                 ...     (
92: (4)                 ...         (1, 2),
93: (4)                 ...         (1, 3),
94: (4)                 ...         (3, 1),
95: (4)                 ...         (3, 2),
96: (4)                 ...         (3, 5),
97: (4)                 ...         (4, 5),
98: (4)                 ...         (4, 6),
99: (4)                 ...         (5, 4),
100: (4)                ...         (5, 6),
101: (4)                ...         (6, 4),
102: (4)                ...     )
103: (4)                ... )
104: (4)                >>> B = nx.directed_modularity_matrix(G)
105: (4)                Notes
106: (4)                -----
107: (4)                NetworkX defines the element A_ij of the adjacency matrix as 1 if there
108: (4)                is a link going from node i to node j. Leicht and Newman use the opposite
109: (4)                definition. This explains the different expression for B_ij.
110: (4)                See Also
111: (4)                --------
112: (4)                to_numpy_array
113: (4)                modularity_spectrum
114: (4)                adjacency_matrix
115: (4)                modularity_matrix
116: (4)                References
117: (4)                ----------
118: (4)                .. [1] E. A. Leicht, M. E. J. Newman,
```

```
119: (8)                    "Community structure in directed networks",
120: (8)                    Phys. Rev Lett., vol. 100, no. 11, p. 118703, 2008.
121: (4)                """
122: (4)                import numpy as np
123: (4)                if nodelist is None:
124: (8)                    nodelist = list(G)
125: (4)                A = nx.to_scipy_sparse_array(G, nodelist=nodelist, weight=weight,
format="csr")
126: (4)                k_in = A.sum(axis=0)
127: (4)                k_out = A.sum(axis=1)
128: (4)                m = k_in.sum()
129: (4)                X = np.outer(k_out, k_in) / m
130: (4)                return A - X
```

File 493 - __init__.py:

```
1: (0)
```

File 494 - test_algebraic_connectivity.py:

```
1: (0)              from math import sqrt
2: (0)              import pytest
3: (0)              np = pytest.importorskip("numpy")
4: (0)              import networkx as nx
5: (0)              methods = ("tracemin_pcg", "tracemin_lu", "lanczos", "lobpcg")
6: (0)              def test_algebraic_connectivity_tracemin_chol():
7: (4)                  """Test that "tracemin_chol" raises an exception."""
8: (4)                  pytest.importorskip("scipy")
9: (4)                  G = nx.barbell_graph(5, 4)
10: (4)                 with pytest.raises(nx.NetworkXError):
11: (8)                     nx.algebraic_connectivity(G, method="tracemin_chol")
12: (0)             def test_fiedler_vector_tracemin_chol():
13: (4)                 """Test that "tracemin_chol" raises an exception."""
14: (4)                 pytest.importorskip("scipy")
15: (4)                 G = nx.barbell_graph(5, 4)
16: (4)                 with pytest.raises(nx.NetworkXError):
17: (8)                     nx.fiedler_vector(G, method="tracemin_chol")
18: (0)             def test_spectral_ordering_tracemin_chol():
19: (4)                 """Test that "tracemin_chol" raises an exception."""
20: (4)                 pytest.importorskip("scipy")
21: (4)                 G = nx.barbell_graph(5, 4)
22: (4)                 with pytest.raises(nx.NetworkXError):
23: (8)                     nx.spectral_ordering(G, method="tracemin_chol")
24: (0)             def test_fiedler_vector_tracemin_unknown():
25: (4)                 """Test that "tracemin_unknown" raises an exception."""
26: (4)                 pytest.importorskip("scipy")
27: (4)                 G = nx.barbell_graph(5, 4)
28: (4)                 L = nx.laplacian_matrix(G)
29: (4)                 X = np.asarray(np.random.normal(size=(1, L.shape[0]))).T
30: (4)                 with pytest.raises(nx.NetworkXError, match="Unknown linear system
solver"):
31: (8)                     nx.linalg.algebraicconnectivity._tracemin_fiedler(
32: (12)                        L, X, normalized=False, tol=1e-8, method="tracemin_unknown"
33: (8)                     )
34: (0)             def test_spectral_bisection():
35: (4)                 pytest.importorskip("scipy")
36: (4)                 G = nx.barbell_graph(3, 0)
37: (4)                 C = nx.spectral_bisection(G)
38: (4)                 assert C == ({0, 1, 2}, {3, 4, 5})
39: (4)                 mapping = dict(enumerate("badfec"))
40: (4)                 G = nx.relabel_nodes(G, mapping)
41: (4)                 C = nx.spectral_bisection(G)
42: (4)                 assert C == (
43: (8)                     {mapping[0], mapping[1], mapping[2]},
44: (8)                     {mapping[3], mapping[4], mapping[5]},
```

```
45: (4)                 )
46: (0)             def check_eigenvector(A, l, x):
47: (4)                 nx = np.linalg.norm(x)
48: (4)                 assert nx != pytest.approx(0, abs=1e-07)
49: (4)                 y = A @ x
50: (4)                 ny = np.linalg.norm(y)
51: (4)                 assert x @ y == pytest.approx(nx * ny, abs=1e-7)
52: (4)                 assert ny == pytest.approx(l * nx, abs=1e-7)
53: (0)             class TestAlgebraicConnectivity:
54: (4)                 @pytest.mark.parametrize("method", methods)
55: (4)                 def test_directed(self, method):
56: (8)                     G = nx.DiGraph()
57: (8)                     pytest.raises(
58: (12)                        nx.NetworkXNotImplemented, nx.algebraic_connectivity, G,
method=method
59: (8)                     )
60: (8)                     pytest.raises(nx.NetworkXNotImplemented, nx.fiedler_vector, G,
method=method)
61: (4)                 @pytest.mark.parametrize("method", methods)
62: (4)                 def test_null_and_singleton(self, method):
63: (8)                     G = nx.Graph()
64: (8)                     pytest.raises(nx.NetworkXError, nx.algebraic_connectivity, G,
method=method)
65: (8)                     pytest.raises(nx.NetworkXError, nx.fiedler_vector, G, method=method)
66: (8)                     G.add_edge(0, 0)
67: (8)                     pytest.raises(nx.NetworkXError, nx.algebraic_connectivity, G,
method=method)
68: (8)                     pytest.raises(nx.NetworkXError, nx.fiedler_vector, G, method=method)
69: (4)                 @pytest.mark.parametrize("method", methods)
70: (4)                 def test_disconnected(self, method):
71: (8)                     G = nx.Graph()
72: (8)                     G.add_nodes_from(range(2))
73: (8)                     assert nx.algebraic_connectivity(G) == 0
74: (8)                     pytest.raises(nx.NetworkXError, nx.fiedler_vector, G, method=method)
75: (8)                     G.add_edge(0, 1, weight=0)
76: (8)                     assert nx.algebraic_connectivity(G) == 0
77: (8)                     pytest.raises(nx.NetworkXError, nx.fiedler_vector, G, method=method)
78: (4)                 def test_unrecognized_method(self):
79: (8)                     pytest.importorskip("scipy")
80: (8)                     G = nx.path_graph(4)
81: (8)                     pytest.raises(nx.NetworkXError, nx.algebraic_connectivity, G,
method="unknown")
82: (8)                     pytest.raises(nx.NetworkXError, nx.fiedler_vector, G,
method="unknown")
83: (4)                 @pytest.mark.parametrize("method", methods)
84: (4)                 def test_two_nodes(self, method):
85: (8)                     pytest.importorskip("scipy")
86: (8)                     G = nx.Graph()
87: (8)                     G.add_edge(0, 1, weight=1)
88: (8)                     A = nx.laplacian_matrix(G)
89: (8)                     assert nx.algebraic_connectivity(G, tol=1e-12, method=method) ==
pytest.approx(
90: (12)                        2, abs=1e-7
91: (8)                     )
92: (8)                     x = nx.fiedler_vector(G, tol=1e-12, method=method)
93: (8)                     check_eigenvector(A, 2, x)
94: (4)                 @pytest.mark.parametrize("method", methods)
95: (4)                 def test_two_nodes_multigraph(self, method):
96: (8)                     pytest.importorskip("scipy")
97: (8)                     G = nx.MultiGraph()
98: (8)                     G.add_edge(0, 0, spam=1e8)
99: (8)                     G.add_edge(0, 1, spam=1)
100: (8)                    G.add_edge(0, 1, spam=-2)
101: (8)                    A = -3 * nx.laplacian_matrix(G, weight="spam")
102: (8)                    assert nx.algebraic_connectivity(
103: (12)                       G, weight="spam", tol=1e-12, method=method
104: (8)                    ) == pytest.approx(6, abs=1e-7)
105: (8)                    x = nx.fiedler_vector(G, weight="spam", tol=1e-12, method=method)
106: (8)                    check_eigenvector(A, 6, x)
```

```
(4)     def test_abbreviation_of_method(self):
(8)         pytest.importorskip("scipy")
(8)         G = nx.path_graph(8)
(8)         A = nx.laplacian_matrix(G)
(8)         sigma = 2 - sqrt(2 + sqrt(2))
(8)         ac = nx.algebraic_connectivity(G, tol=1e-12, method="tracemin")
(8)         assert ac == pytest.approx(sigma, abs=1e-7)
(8)         x = nx.fiedler_vector(G, tol=1e-12, method="tracemin")
(8)         check_eigenvector(A, sigma, x)
(4)     @pytest.mark.parametrize("method", methods)
(4)     def test_path(self, method):
(8)         pytest.importorskip("scipy")
(8)         G = nx.path_graph(8)
(8)         A = nx.laplacian_matrix(G)
(8)         sigma = 2 - sqrt(2 + sqrt(2))
(8)         ac = nx.algebraic_connectivity(G, tol=1e-12, method=method)
(8)         assert ac == pytest.approx(sigma, abs=1e-7)
(8)         x = nx.fiedler_vector(G, tol=1e-12, method=method)
(8)         check_eigenvector(A, sigma, x)
(4)     @pytest.mark.parametrize("method", methods)
(4)     def test_problematic_graph_issue_2381(self, method):
(8)         pytest.importorskip("scipy")
(8)         G = nx.path_graph(4)
(8)         G.add_edges_from([(4, 2), (5, 1)])
(8)         A = nx.laplacian_matrix(G)
(8)         sigma = 0.438447187191
(8)         ac = nx.algebraic_connectivity(G, tol=1e-12, method=method)
(8)         assert ac == pytest.approx(sigma, abs=1e-7)
(8)         x = nx.fiedler_vector(G, tol=1e-12, method=method)
(8)         check_eigenvector(A, sigma, x)
(4)     @pytest.mark.parametrize("method", methods)
(4)     def test_cycle(self, method):
(8)         pytest.importorskip("scipy")
(8)         G = nx.cycle_graph(8)
(8)         A = nx.laplacian_matrix(G)
(8)         sigma = 2 - sqrt(2)
(8)         ac = nx.algebraic_connectivity(G, tol=1e-12, method=method)
(8)         assert ac == pytest.approx(sigma, abs=1e-7)
(8)         x = nx.fiedler_vector(G, tol=1e-12, method=method)
(8)         check_eigenvector(A, sigma, x)
(4)     @pytest.mark.parametrize("method", methods)
(4)     def test_seed_argument(self, method):
(8)         pytest.importorskip("scipy")
(8)         G = nx.cycle_graph(8)
(8)         A = nx.laplacian_matrix(G)
(8)         sigma = 2 - sqrt(2)
(8)         ac = nx.algebraic_connectivity(G, tol=1e-12, method=method, seed=1)
(8)         assert ac == pytest.approx(sigma, abs=1e-7)
(8)         x = nx.fiedler_vector(G, tol=1e-12, method=method, seed=1)
(8)         check_eigenvector(A, sigma, x)
(4)     @pytest.mark.parametrize(
(8)         ("normalized", "sigma", "laplacian_fn"),
(8)         (
(12)            (False, 0.2434017461399311, nx.laplacian_matrix),
(12)            (True, 0.08113391537997749, nx.normalized_laplacian_matrix),
(8)         ),
(4)     )
(4)     @pytest.mark.parametrize("method", methods)
(4)     def test_buckminsterfullerene(self, normalized, sigma, laplacian_fn,
method):
(8)         pytest.importorskip("scipy")
(8)         G = nx.Graph(
(12)            [
(16)                (1, 10),
(16)                (1, 41),
(16)                (1, 59),
(16)                (2, 12),
(16)                (2, 42),
(16)                (2, 60),
```

```
175: (16)                           (3, 6),
176: (16)                           (3, 43),
177: (16)                           (3, 57),
178: (16)                           (4, 8),
179: (16)                           (4, 44),
180: (16)                           (4, 58),
181: (16)                           (5, 13),
182: (16)                           (5, 56),
183: (16)                           (5, 57),
184: (16)                           (6, 10),
185: (16)                           (6, 31),
186: (16)                           (7, 14),
187: (16)                           (7, 56),
188: (16)                           (7, 58),
189: (16)                           (8, 12),
190: (16)                           (8, 32),
191: (16)                           (9, 23),
192: (16)                           (9, 53),
193: (16)                           (9, 59),
194: (16)                           (10, 15),
195: (16)                           (11, 24),
196: (16)                           (11, 53),
197: (16)                           (11, 60),
198: (16)                           (12, 16),
199: (16)                           (13, 14),
200: (16)                           (13, 25),
201: (16)                           (14, 26),
202: (16)                           (15, 27),
203: (16)                           (15, 49),
204: (16)                           (16, 28),
205: (16)                           (16, 50),
206: (16)                           (17, 18),
207: (16)                           (17, 19),
208: (16)                           (17, 54),
209: (16)                           (18, 20),
210: (16)                           (18, 55),
211: (16)                           (19, 23),
212: (16)                           (19, 41),
213: (16)                           (20, 24),
214: (16)                           (20, 42),
215: (16)                           (21, 31),
216: (16)                           (21, 33),
217: (16)                           (21, 57),
218: (16)                           (22, 32),
219: (16)                           (22, 34),
220: (16)                           (22, 58),
221: (16)                           (23, 24),
222: (16)                           (25, 35),
223: (16)                           (25, 43),
224: (16)                           (26, 36),
225: (16)                           (26, 44),
226: (16)                           (27, 51),
227: (16)                           (27, 59),
228: (16)                           (28, 52),
229: (16)                           (28, 60),
230: (16)                           (29, 33),
231: (16)                           (29, 34),
232: (16)                           (29, 56),
233: (16)                           (30, 51),
234: (16)                           (30, 52),
235: (16)                           (30, 53),
236: (16)                           (31, 47),
237: (16)                           (32, 48),
238: (16)                           (33, 45),
239: (16)                           (34, 46),
240: (16)                           (35, 36),
241: (16)                           (35, 37),
242: (16)                           (36, 38),
243: (16)                           (37, 39),
```

```
244: (16)                            (37, 49),
245: (16)                            (38, 40),
246: (16)                            (38, 50),
247: (16)                            (39, 40),
248: (16)                            (39, 51),
249: (16)                            (40, 52),
250: (16)                            (41, 47),
251: (16)                            (42, 48),
252: (16)                            (43, 49),
253: (16)                            (44, 50),
254: (16)                            (45, 46),
255: (16)                            (45, 54),
256: (16)                            (46, 55),
257: (16)                            (47, 54),
258: (16)                            (48, 55),
259: (12)                        ]
260: (8)                    )
261: (8)                    A = laplacian_fn(G)
262: (8)                    try:
263: (12)                        assert nx.algebraic_connectivity(
264: (16)                            G, normalized=normalized, tol=1e-12, method=method
265: (12)                        ) == pytest.approx(sigma, abs=1e-7)
266: (12)                        x = nx.fiedler_vector(G, normalized=normalized, tol=1e-12,
method=method)
267: (12)                        check_eigenvector(A, sigma, x)
268: (8)                    except nx.NetworkXError as err:
269: (12)                        if err.args not in (
270: (16)                            ("Cholesky solver unavailable.",),
271: (16)                            ("LU solver unavailable.",),
272: (12)                        ):
273: (16)                            raise
274: (0)            class TestSpectralOrdering:
275: (4)                _graphs = (nx.Graph, nx.DiGraph, nx.MultiGraph, nx.MultiDiGraph)
276: (4)                @pytest.mark.parametrize("graph", _graphs)
277: (4)                def test_nullgraph(self, graph):
278: (8)                    G = graph()
279: (8)                    pytest.raises(nx.NetworkXError, nx.spectral_ordering, G)
280: (4)                @pytest.mark.parametrize("graph", _graphs)
281: (4)                def test_singleton(self, graph):
282: (8)                    G = graph()
283: (8)                    G.add_node("x")
284: (8)                    assert nx.spectral_ordering(G) == ["x"]
285: (8)                    G.add_edge("x", "x", weight=33)
286: (8)                    G.add_edge("x", "x", weight=33)
287: (8)                    assert nx.spectral_ordering(G) == ["x"]
288: (4)                def test_unrecognized_method(self):
289: (8)                    G = nx.path_graph(4)
290: (8)                    pytest.raises(nx.NetworkXError, nx.spectral_ordering, G,
method="unknown")
291: (4)                @pytest.mark.parametrize("method", methods)
292: (4)                def test_three_nodes(self, method):
293: (8)                    pytest.importorskip("scipy")
294: (8)                    G = nx.Graph()
295: (8)                    G.add_weighted_edges_from([(1, 2, 1), (1, 3, 2), (2, 3, 1)],
weight="spam")
296: (8)                    order = nx.spectral_ordering(G, weight="spam", method=method)
297: (8)                    assert set(order) == set(G)
298: (8)                    assert {1, 3} in (set(order[:-1]), set(order[1:]))
299: (4)                @pytest.mark.parametrize("method", methods)
300: (4)                def test_three_nodes_multigraph(self, method):
301: (8)                    pytest.importorskip("scipy")
302: (8)                    G = nx.MultiDiGraph()
303: (8)                    G.add_weighted_edges_from([(1, 2, 1), (1, 3, 2), (2, 3, 1), (2, 3,
2)])
304: (8)                    order = nx.spectral_ordering(G, method=method)
305: (8)                    assert set(order) == set(G)
306: (8)                    assert {2, 3} in (set(order[:-1]), set(order[1:]))
307: (4)                @pytest.mark.parametrize("method", methods)
308: (4)                def test_path(self, method):
```

```
309: (8)                    pytest.importorskip("scipy")
310: (8)                    path = list(range(10))
311: (8)                    np.random.shuffle(path)
312: (8)                    G = nx.Graph()
313: (8)                    nx.add_path(G, path)
314: (8)                    order = nx.spectral_ordering(G, method=method)
315: (8)                    assert order in [path, list(reversed(path))]
316: (4)                @pytest.mark.parametrize("method", methods)
317: (4)                def test_seed_argument(self, method):
318: (8)                    pytest.importorskip("scipy")
319: (8)                    path = list(range(10))
320: (8)                    np.random.shuffle(path)
321: (8)                    G = nx.Graph()
322: (8)                    nx.add_path(G, path)
323: (8)                    order = nx.spectral_ordering(G, method=method, seed=1)
324: (8)                    assert order in [path, list(reversed(path))]
325: (4)                @pytest.mark.parametrize("method", methods)
326: (4)                def test_disconnected(self, method):
327: (8)                    pytest.importorskip("scipy")
328: (8)                    G = nx.Graph()
329: (8)                    nx.add_path(G, range(0, 10, 2))
330: (8)                    nx.add_path(G, range(1, 10, 2))
331: (8)                    order = nx.spectral_ordering(G, method=method)
332: (8)                    assert set(order) == set(G)
333: (8)                    seqs = [
334: (12)                       list(range(0, 10, 2)),
335: (12)                       list(range(8, -1, -2)),
336: (12)                       list(range(1, 10, 2)),
337: (12)                       list(range(9, -1, -2)),
338: (8)                    ]
339: (8)                    assert order[:5] in seqs
340: (8)                    assert order[5:] in seqs
341: (4)                @pytest.mark.parametrize(
342: (8)                    ("normalized", "expected_order"),
343: (8)                    (
344: (12)                       (False, [[1, 2, 0, 3, 4, 5, 6, 9, 7, 8], [8, 7, 9, 6, 5, 4, 3, 0,
2, 1]]),
345: (12)                       (True, [[1, 2, 3, 0, 4, 5, 9, 6, 7, 8], [8, 7, 6, 9, 5, 4, 0, 3,
2, 1]]),
346: (8)                    ),
347: (4)                )
348: (4)                @pytest.mark.parametrize("method", methods)
349: (4)                def test_cycle(self, normalized, expected_order, method):
350: (8)                    pytest.importorskip("scipy")
351: (8)                    path = list(range(10))
352: (8)                    G = nx.Graph()
353: (8)                    nx.add_path(G, path, weight=5)
354: (8)                    G.add_edge(path[-1], path[0], weight=1)
355: (8)                    A = nx.laplacian_matrix(G).todense()
356: (8)                    order = nx.spectral_ordering(G, normalized=normalized, method=method)
357: (8)                    assert order in expected_order
```

File 495 - test_attrmatrix.py:

```
1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              import networkx as nx
4: (0)              def test_attr_matrix():
5: (4)                  G = nx.Graph()
6: (4)                  G.add_edge(0, 1, thickness=1, weight=3)
7: (4)                  G.add_edge(0, 1, thickness=1, weight=3)
8: (4)                  G.add_edge(0, 2, thickness=2)
9: (4)                  G.add_edge(1, 2, thickness=3)
10: (4)                 def node_attr(u):
11: (8)                     return G.nodes[u].get("size", 0.5) * 3
12: (4)                 def edge_attr(u, v):
13: (8)                     return G[u][v].get("thickness", 0.5)
```

```
 14: (4)                 M = nx.attr_matrix(G, edge_attr=edge_attr, node_attr=node_attr)
 15: (4)                 np.testing.assert_equal(M[0], np.array([[6.0]]))
 16: (4)                 assert M[1] == [1.5]
 17: (0)             def test_attr_matrix_directed():
 18: (4)                 G = nx.DiGraph()
 19: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 20: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 21: (4)                 G.add_edge(0, 2, thickness=2)
 22: (4)                 G.add_edge(1, 2, thickness=3)
 23: (4)                 M = nx.attr_matrix(G, rc_order=[0, 1, 2])
 24: (4)                 data = np.array(
 25: (8)                     [[0., 1., 1.],
 26: (9)                      [0., 0., 1.],
 27: (9)                      [0., 0., 0.]]
 28: (4)                 )
 29: (4)                 np.testing.assert_equal(M, np.array(data))
 30: (0)             def test_attr_matrix_multigraph():
 31: (4)                 G = nx.MultiGraph()
 32: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 33: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 34: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 35: (4)                 G.add_edge(0, 2, thickness=2)
 36: (4)                 G.add_edge(1, 2, thickness=3)
 37: (4)                 M = nx.attr_matrix(G, rc_order=[0, 1, 2])
 38: (4)                 data = np.array(
 39: (8)                     [[0., 3., 1.],
 40: (9)                      [3., 0., 1.],
 41: (9)                      [1., 1., 0.]]
 42: (4)                 )
 43: (4)                 np.testing.assert_equal(M, np.array(data))
 44: (4)                 M = nx.attr_matrix(G, edge_attr="weight", rc_order=[0, 1, 2])
 45: (4)                 data = np.array(
 46: (8)                     [[0., 9., 1.],
 47: (9)                      [9., 0., 1.],
 48: (9)                      [1., 1., 0.]]
 49: (4)                 )
 50: (4)                 np.testing.assert_equal(M, np.array(data))
 51: (4)                 M = nx.attr_matrix(G, edge_attr="thickness", rc_order=[0, 1, 2])
 52: (4)                 data = np.array(
 53: (8)                     [[0., 3., 2.],
 54: (9)                      [3., 0., 3.],
 55: (9)                      [2., 3., 0.]]
 56: (4)                 )
 57: (4)                 np.testing.assert_equal(M, np.array(data))
 58: (0)             def test_attr_sparse_matrix():
 59: (4)                 pytest.importorskip("scipy")
 60: (4)                 G = nx.Graph()
 61: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 62: (4)                 G.add_edge(0, 2, thickness=2)
 63: (4)                 G.add_edge(1, 2, thickness=3)
 64: (4)                 M = nx.attr_sparse_matrix(G)
 65: (4)                 mtx = M[0]
 66: (4)                 data = np.ones((3, 3), float)
 67: (4)                 np.fill_diagonal(data, 0)
 68: (4)                 np.testing.assert_equal(mtx.todense(), np.array(data))
 69: (4)                 assert M[1] == [0, 1, 2]
 70: (0)             def test_attr_sparse_matrix_directed():
 71: (4)                 pytest.importorskip("scipy")
 72: (4)                 G = nx.DiGraph()
 73: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 74: (4)                 G.add_edge(0, 1, thickness=1, weight=3)
 75: (4)                 G.add_edge(0, 2, thickness=2)
 76: (4)                 G.add_edge(1, 2, thickness=3)
 77: (4)                 M = nx.attr_sparse_matrix(G, rc_order=[0, 1, 2])
 78: (4)                 data = np.array(
 79: (8)                     [[0., 1., 1.],
 80: (9)                      [0., 0., 1.],
 81: (9)                      [0., 0., 0.]]
 82: (4)                 )
```

```
83: (4)                 np.testing.assert_equal(M.todense(), np.array(data))

----------------------------------------

File 496 - test_bethehessian.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.generators.degree_seq import havel_hakimi_graph
6: (0)              class TestBetheHessian:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      deg = [3, 2, 2, 1, 0]
10: (8)                     cls.G = havel_hakimi_graph(deg)
11: (8)                     cls.P = nx.path_graph(3)
12: (4)                 def test_bethe_hessian(self):
13: (8)                     "Bethe Hessian matrix"
14: (8)                     H = np.array([[4, -2, 0],
15: (22)                                  [-2, 5, -2],
16: (22)                                  [0, -2, 4]])
17: (8)                     permutation = [2, 0, 1]
18: (8)                     np.testing.assert_equal(nx.bethe_hessian_matrix(self.P,
r=2).todense(), H)
19: (8)                     np.testing.assert_equal(
20: (12)                        nx.bethe_hessian_matrix(self.P, r=2,
nodelist=permutation).todense(),
21: (12)                        H[np.ix_(permutation, permutation)],
22: (8)                     )
23: (8)                     np.testing.assert_equal(
24: (12)                        nx.bethe_hessian_matrix(self.G, r=1).todense(),
25: (12)                        nx.laplacian_matrix(self.G).todense(),
26: (8)                     )
27: (8)                     np.testing.assert_equal(
28: (12)                        nx.bethe_hessian_matrix(self.G).todense(),
29: (12)                        nx.bethe_hessian_matrix(self.G, r=1.25).todense(),
30: (8)                     )

----------------------------------------

File 497 - test_laplacian.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.generators.degree_seq import havel_hakimi_graph
6: (0)              from networkx.generators.expanders import margulis_gabber_galil_graph
7: (0)              class TestLaplacian:
8: (4)                  @classmethod
9: (4)                  def setup_class(cls):
10: (8)                     deg = [3, 2, 2, 1, 0]
11: (8)                     cls.G = havel_hakimi_graph(deg)
12: (8)                     cls.WG = nx.Graph(
13: (12)                        (u, v, {"weight": 0.5, "other": 0.3}) for (u, v) in cls.G.edges()
14: (8)                     )
15: (8)                     cls.WG.add_node(4)
16: (8)                     cls.MG = nx.MultiGraph(cls.G)
17: (8)                     cls.Gsl = cls.G.copy()
18: (8)                     for node in cls.Gsl.nodes():
19: (12)                        cls.Gsl.add_edge(node, node)
20: (8)                     cls.DiG = nx.DiGraph()
21: (8)                     cls.DiG.add_edges_from(
22: (12)                        (
23: (16)                            (1, 2),
24: (16)                            (1, 3),
25: (16)                            (3, 1),
26: (16)                            (3, 2),
```

```
 27: (16)                            (3, 5),
 28: (16)                            (4, 5),
 29: (16)                            (4, 6),
 30: (16)                            (5, 4),
 31: (16)                            (5, 6),
 32: (16)                            (6, 4),
 33: (12)                        )
 34: (8)                     )
 35: (8)                     cls.DiMG = nx.MultiDiGraph(cls.DiG)
 36: (8)                     cls.DiWG = nx.DiGraph(
 37: (12)                        (u, v, {"weight": 0.5, "other": 0.3}) for (u, v) in
cls.DiG.edges()
 38: (8)                     )
 39: (8)                     cls.DiGsl = cls.DiG.copy()
 40: (8)                     for node in cls.DiGsl.nodes():
 41: (12)                        cls.DiGsl.add_edge(node, node)
 42: (4)                 def test_laplacian(self):
 43: (8)                     "Graph Laplacian"
 44: (8)                     NL = np.array([[ 3, -1, -1, -1,  0],
 45: (23)                                   [-1,  2, -1,  0,  0],
 46: (23)                                   [-1, -1,  2,  0,  0],
 47: (23)                                   [-1,  0,  0,  1,  0],
 48: (23)                                   [ 0,  0,  0,  0,  0]])
 49: (8)                     WL = 0.5 * NL
 50: (8)                     OL = 0.3 * NL
 51: (8)                     DiNL = np.array([[ 2, -1, -1,  0,  0,  0],
 52: (25)                                     [ 0,  0,  0,  0,  0,  0],
 53: (25)                                     [-1, -1,  3, -1,  0,  0],
 54: (25)                                     [ 0,  0,  0,  2, -1, -1],
 55: (25)                                     [ 0,  0,  0, -1,  2, -1],
 56: (25)                                     [ 0,  0,  0,  0, -1,  1]])
 57: (8)                     DiWL = 0.5 * DiNL
 58: (8)                     DiOL = 0.3 * DiNL
 59: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.G).todense(), NL)
 60: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.MG).todense(), NL)
 61: (8)                     np.testing.assert_equal(
 62: (12)                        nx.laplacian_matrix(self.G, nodelist=[0, 1]).todense(),
 63: (12)                        np.array([[1, -1], [-1, 1]]),
 64: (8)                     )
 65: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.WG).todense(), WL)
 66: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.WG,
weight=None).todense(), NL)
 67: (8)                     np.testing.assert_equal(
 68: (12)                        nx.laplacian_matrix(self.WG, weight="other").todense(), OL
 69: (8)                     )
 70: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.DiG).todense(), DiNL)
 71: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.DiMG).todense(),
DiNL)
 72: (8)                     np.testing.assert_equal(
 73: (12)                        nx.laplacian_matrix(self.DiG, nodelist=[1, 2]).todense(),
 74: (12)                        np.array([[1, -1], [0, 0]]),
 75: (8)                     )
 76: (8)                     np.testing.assert_equal(nx.laplacian_matrix(self.DiWG).todense(),
DiWL)
 77: (8)                     np.testing.assert_equal(
 78: (12)                        nx.laplacian_matrix(self.DiWG, weight=None).todense(), DiNL
 79: (8)                     )
 80: (8)                     np.testing.assert_equal(
 81: (12)                        nx.laplacian_matrix(self.DiWG, weight="other").todense(), DiOL
 82: (8)                     )
 83: (4)                 def test_normalized_laplacian(self):
 84: (8)                     "Generalized Graph Laplacian"
 85: (8)                     G = np.array([[ 1.   , -0.408, -0.408, -0.577,  0.],
 86: (22)                                  [-0.408,  1.   , -0.5  ,  0.   ,  0.],
 87: (22)                                  [-0.408, -0.5  ,  1.   ,  0.   ,  0.],
 88: (22)                                  [-0.577,  0.   ,  0.   ,  1.   ,  0.],
 89: (22)                                  [ 0.   ,  0.   ,  0.   ,  0.   ,  0.]])
 90: (8)                     GL = np.array([[ 1.   , -0.408, -0.408, -0.577,  0.   ],
 91: (23)                                   [-0.408,  1.   , -0.5  ,  0.   ,  0.   ],
```

```
92: (23)                                    [-0.408, -0.5  ,  1.   ,  0.   ,  0.   ],
93: (23)                                    [-0.577,  0.   ,  0.   ,  1.   ,  0.   ],
94: (23)                                    [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
95: (8)                     Lsl = np.array([[ 0.75  , -0.2887, -0.2887, -0.3536,  0.    ],
96: (24)                                     [-0.2887,  0.6667, -0.3333,  0.    ,  0.    ],
97: (24)                                     [-0.2887, -0.3333,  0.6667,  0.    ,  0.    ],
98: (24)                                     [-0.3536,  0.    ,  0.    ,  0.5   ,  0.    ],
99: (24)                                     [ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ]])
100: (8)                    DiG = np.array([[ 1.    ,  0.    , -0.4082,  0.    ,  0.    ,  0.
],
101: (24)                                    [ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.
],
102: (24)                                    [-0.4082,  0.    ,  1.    ,  0.    , -0.4082,  0.
],
103: (24)                                    [ 0.    ,  0.    ,  0.    ,  1.    , -0.5   ,
-0.7071],
104: (24)                                    [ 0.    ,  0.    ,  0.    , -0.5   ,  1.    ,
-0.7071],
105: (24)                                    [ 0.    ,  0.    ,  0.    , -0.7071,  0.     , 1.
]])
106: (8)                    DiGL = np.array([[ 1.    ,  0.    , -0.4082,  0.    ,  0.    ,  0.
],
107: (25)                                     [ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.
],
108: (25)                                     [-0.4082,  0.    ,  1.    , -0.4082,  0.    ,  0.
],
109: (25)                                     [ 0.    ,  0.    ,  0.    ,  1.    , -0.5   ,
-0.7071],
110: (25)                                     [ 0.    ,  0.    ,  0.    , -0.5   ,  1.    ,
-0.7071],
111: (25)                                     [ 0.    ,  0.    ,  0.    ,  0.    , -0.7071,  1.
]])
112: (8)                    DiLsl = np.array([[ 0.6667, -0.5774, -0.2887,  0.    ,  0.    ,  0.
],
113: (26)                                      [ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.
],
114: (26)                                      [-0.2887, -0.5   ,  0.75  , -0.2887,  0.    ,  0.
],
115: (26)                                      [ 0.    ,  0.    ,  0.    ,  0.6667, -0.3333,
-0.4082],
116: (26)                                      [ 0.    ,  0.    ,  0.    , -0.3333,  0.6667,
-0.4082],
117: (26)                                      [ 0.    ,  0.    ,  0.    ,  0.    , -0.4082,  0.5
]])
118: (8)                    np.testing.assert_almost_equal(
119: (12)                       nx.normalized_laplacian_matrix(self.G,
nodelist=range(5)).todense(),
120: (12)                       G,
121: (12)                       decimal=3,
122: (8)                    )
123: (8)                    np.testing.assert_almost_equal(
124: (12)                       nx.normalized_laplacian_matrix(self.G).todense(), GL, decimal=3
125: (8)                    )
126: (8)                    np.testing.assert_almost_equal(
127: (12)                       nx.normalized_laplacian_matrix(self.MG).todense(), GL, decimal=3
128: (8)                    )
129: (8)                    np.testing.assert_almost_equal(
130: (12)                       nx.normalized_laplacian_matrix(self.WG).todense(), GL, decimal=3
131: (8)                    )
132: (8)                    np.testing.assert_almost_equal(
133: (12)                       nx.normalized_laplacian_matrix(self.WG, weight="other").todense(),
134: (12)                       GL,
135: (12)                       decimal=3,
136: (8)                    )
137: (8)                    np.testing.assert_almost_equal(
138: (12)                       nx.normalized_laplacian_matrix(self.Gsl).todense(), Lsl, decimal=3
139: (8)                    )
140: (8)                    np.testing.assert_almost_equal(
141: (12)                       nx.normalized_laplacian_matrix(
```

```
142: (16)                            self.DiG,
143: (16)                            nodelist=range(1, 1 + 6),
144: (12)                        ).todense(),
145: (12)                        DiG,
146: (12)                        decimal=3,
147: (8)                    )
148: (8)                    np.testing.assert_almost_equal(
149: (12)                        nx.normalized_laplacian_matrix(self.DiG).todense(), DiGL,
decimal=3
150: (8)                    )
151: (8)                    np.testing.assert_almost_equal(
152: (12)                        nx.normalized_laplacian_matrix(self.DiMG).todense(), DiGL,
decimal=3
153: (8)                    )
154: (8)                    np.testing.assert_almost_equal(
155: (12)                        nx.normalized_laplacian_matrix(self.DiWG).todense(), DiGL,
decimal=3
156: (8)                    )
157: (8)                    np.testing.assert_almost_equal(
158: (12)                        nx.normalized_laplacian_matrix(self.DiWG,
weight="other").todense(),
159: (12)                        DiGL,
160: (12)                        decimal=3,
161: (8)                    )
162: (8)                    np.testing.assert_almost_equal(
163: (12)                        nx.normalized_laplacian_matrix(self.DiGsl).todense(), DiLsl,
decimal=3
164: (8)                    )
165: (0)            def test_directed_laplacian():
166: (4)                "Directed Laplacian"
167: (4)                G = nx.DiGraph()
168: (4)                G.add_edges_from(
169: (8)                    (
170: (12)                        (1, 2),
171: (12)                        (1, 3),
172: (12)                        (3, 1),
173: (12)                        (3, 2),
174: (12)                        (3, 5),
175: (12)                        (4, 5),
176: (12)                        (4, 6),
177: (12)                        (5, 4),
178: (12)                        (5, 6),
179: (12)                        (6, 4),
180: (8)                    )
181: (4)                )
182: (4)                GL = np.array([[ 0.9833, -0.2941, -0.3882, -0.0291, -0.0231, -0.0261],
183: (19)                               [-0.2941,  0.8333, -0.2339, -0.0536, -0.0589, -0.0554],
184: (19)                               [-0.3882, -0.2339,  0.9833, -0.0278, -0.0896, -0.0251],
185: (19)                               [-0.0291, -0.0536, -0.0278,  0.9833, -0.4878, -0.6675],
186: (19)                               [-0.0231, -0.0589, -0.0896, -0.4878,  0.9833, -0.2078],
187: (19)                               [-0.0261, -0.0554, -0.0251, -0.6675, -0.2078,  0.9833]])
188: (4)                L = nx.directed_laplacian_matrix(G, alpha=0.9, nodelist=sorted(G))
189: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
190: (4)                G.add_edges_from(((2, 5), (6, 1)))
191: (4)                GL = np.array([[ 1.    , -0.3062, -0.4714,  0.    ,  0.    , -0.3227],
192: (19)                               [-0.3062,  1.    , -0.1443,  0.    , -0.3162,  0.    ],
193: (19)                               [-0.4714, -0.1443,  1.    ,  0.    , -0.0913,  0.    ],
194: (19)                               [ 0.    ,  0.    ,  0.    ,  1.    , -0.5   , -0.5   ],
195: (19)                               [ 0.    , -0.3162, -0.0913, -0.5   ,  1.    , -0.25  ],
196: (19)                               [-0.3227,  0.    ,  0.    , -0.5   , -0.25  ,  1.    ]])
197: (4)                L = nx.directed_laplacian_matrix(
198: (8)                    G, alpha=0.9, nodelist=sorted(G), walk_type="random"
199: (4)                )
200: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
201: (4)                GL = np.array([[ 0.5   , -0.1531, -0.2357,  0.    ,  0.    , -0.1614],
202: (19)                               [-0.1531,  0.5   , -0.0722,  0.    , -0.1581,  0.    ],
203: (19)                               [-0.2357, -0.0722,  0.5   ,  0.    , -0.0456,  0.    ],
204: (19)                               [ 0.    ,  0.    ,  0.    ,  0.5   , -0.25  , -0.25  ],
205: (19)                               [ 0.    , -0.1581, -0.0456, -0.25  ,  0.5   , -0.125 ],
```

```
206: (19)                               [-0.1614,  0.    ,  0.    , -0.25  , -0.125 ,  0.5   ]])
207: (4)                L = nx.directed_laplacian_matrix(G, alpha=0.9, nodelist=sorted(G),
walk_type="lazy")
208: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
209: (4)                G = nx.DiGraph()
210: (4)                G.add_edges_from(((1, 2), (2, 4), (4, 1), (1, 3), (3, 4)))
211: (4)                GL = np.array([[ 0.5  , -0.176, -0.176, -0.25 ],
212: (19)                               [-0.176,  0.5  ,  0.   , -0.176],
213: (19)                               [-0.176,  0.   ,  0.5  , -0.176],
214: (19)                               [-0.25 , -0.176, -0.176,  0.5  ]])
215: (4)                L = nx.directed_laplacian_matrix(G, alpha=0.9, nodelist=sorted(G))
216: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
217: (0)            def test_directed_combinatorial_laplacian():
218: (4)                "Directed combinatorial Laplacian"
219: (4)                G = nx.DiGraph()
220: (4)                G.add_edges_from(
221: (8)                    (
222: (12)                       (1, 2),
223: (12)                       (1, 3),
224: (12)                       (3, 1),
225: (12)                       (3, 2),
226: (12)                       (3, 5),
227: (12)                       (4, 5),
228: (12)                       (4, 6),
229: (12)                       (5, 4),
230: (12)                       (5, 6),
231: (12)                       (6, 4),
232: (8)                    )
233: (4)                )
234: (4)                GL = np.array([[ 0.0366, -0.0132, -0.0153, -0.0034, -0.0020, -0.0027],
235: (19)                               [-0.0132,  0.0450, -0.0111, -0.0076, -0.0062, -0.0069],
236: (19)                               [-0.0153, -0.0111,  0.0408, -0.0035, -0.0083, -0.0027],
237: (19)                               [-0.0034, -0.0076, -0.0035,  0.3688, -0.1356, -0.2187],
238: (19)                               [-0.0020, -0.0062, -0.0083, -0.1356,  0.2026, -0.0505],
239: (19)                               [-0.0027, -0.0069, -0.0027, -0.2187, -0.0505,  0.2815]])
240: (4)                L = nx.directed_combinatorial_laplacian_matrix(G, alpha=0.9,
nodelist=sorted(G))
241: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
242: (4)                G.add_edges_from(((2, 5), (6, 1)))
243: (4)                GL = np.array([[ 0.1395, -0.0349, -0.0465,  0.    ,  0.    , -0.0581],
244: (19)                               [-0.0349,  0.093 , -0.0116,  0.    , -0.0465,  0.    ],
245: (19)                               [-0.0465, -0.0116,  0.0698,  0.    , -0.0116,  0.    ],
246: (19)                               [ 0.    ,  0.    ,  0.    ,  0.2326, -0.1163, -0.1163],
247: (19)                               [ 0.    , -0.0465, -0.0116, -0.1163,  0.2326, -0.0581],
248: (19)                               [-0.0581,  0.    ,  0.    , -0.1163, -0.0581,  0.2326]])
249: (4)                L = nx.directed_combinatorial_laplacian_matrix(
250: (8)                    G, alpha=0.9, nodelist=sorted(G), walk_type="random"
251: (4)                )
252: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
253: (4)                GL = np.array([[ 0.0698, -0.0174, -0.0233,  0.    ,  0.    , -0.0291],
254: (19)                               [-0.0174,  0.0465, -0.0058,  0.    , -0.0233,  0.    ],
255: (19)                               [-0.0233, -0.0058,  0.0349,  0.    , -0.0058,  0.    ],
256: (19)                               [ 0.    ,  0.    ,  0.    ,  0.1163, -0.0581, -0.0581],
257: (19)                               [ 0.    , -0.0233, -0.0058, -0.0581,  0.1163, -0.0291],
258: (19)                               [-0.0291,  0.    ,  0.    , -0.0581, -0.0291,  0.1163]])
259: (4)                L = nx.directed_combinatorial_laplacian_matrix(
260: (8)                    G, alpha=0.9, nodelist=sorted(G), walk_type="lazy"
261: (4)                )
262: (4)                np.testing.assert_almost_equal(L, GL, decimal=3)
263: (4)                E = nx.DiGraph(margulis_gabber_galil_graph(2))
264: (4)                L = nx.directed_combinatorial_laplacian_matrix(E)
265: (4)                expected = np.array(
266: (8)                    [[ 0.16666667, -0.08333333, -0.08333333,  0.        ],
267: (9)                     [-0.08333333,  0.16666667,  0.        , -0.08333333],
268: (9)                     [-0.08333333,  0.        ,  0.16666667, -0.08333333],
269: (9)                     [ 0.        , -0.08333333, -0.08333333,  0.16666667]]
270: (4)                )
271: (4)                np.testing.assert_almost_equal(L, expected, decimal=6)
272: (4)                with pytest.raises(nx.NetworkXError):
```

```
273: (8)                    nx.directed_combinatorial_laplacian_matrix(G, walk_type="pagerank",
alpha=100)
274: (4)                with pytest.raises(nx.NetworkXError):
275: (8)                    nx.directed_combinatorial_laplacian_matrix(G, walk_type="silly")
```

File 498 - test_graphmatrix.py:

```
1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.exception import NetworkXError
6: (0)              from networkx.generators.degree_seq import havel_hakimi_graph
7: (0)              def test_incidence_matrix_simple():
8: (4)                  deg = [3, 2, 2, 1, 0]
9: (4)                  G = havel_hakimi_graph(deg)
10: (4)                 deg = [(1, 0), (1, 0), (1, 0), (2, 0), (1, 0), (2, 1), (0, 1), (0, 1)]
11: (4)                 MG = nx.random_clustered_graph(deg, seed=42)
12: (4)                 I = nx.incidence_matrix(G, dtype=int).todense()
13: (4)                 expected = np.array(
14: (8)                     [[1, 1, 1, 0],
15: (9)                      [0, 1, 0, 1],
16: (9)                      [1, 0, 0, 1],
17: (9)                      [0, 0, 1, 0],
18: (9)                      [0, 0, 0, 0]]
19: (4)                 )
20: (4)                 np.testing.assert_equal(I, expected)
21: (4)                 I = nx.incidence_matrix(MG, dtype=int).todense()
22: (4)                 expected = np.array(
23: (8)                     [[1, 0, 0, 0, 0, 0, 0],
24: (9)                      [1, 0, 0, 0, 0, 0, 0],
25: (9)                      [0, 1, 0, 0, 0, 0, 0],
26: (9)                      [0, 0, 0, 0, 0, 0, 0],
27: (9)                      [0, 1, 0, 0, 0, 0, 0],
28: (9)                      [0, 0, 0, 0, 1, 1, 0],
29: (9)                      [0, 0, 0, 0, 0, 1, 1],
30: (9)                      [0, 0, 0, 0, 1, 0, 1]]
31: (4)                 )
32: (4)                 np.testing.assert_equal(I, expected)
33: (4)                 with pytest.raises(NetworkXError):
34: (8)                     nx.incidence_matrix(G, nodelist=[0, 1])
35: (0)             class TestGraphMatrix:
36: (4)                 @classmethod
37: (4)                 def setup_class(cls):
38: (8)                     deg = [3, 2, 2, 1, 0]
39: (8)                     cls.G = havel_hakimi_graph(deg)
40: (8)                     cls.OI = np.array(
41: (12)                        [[-1, -1, -1, 0],
42: (13)                         [1, 0, 0, -1],
43: (13)                         [0, 1, 0, 1],
44: (13)                         [0, 0, 1, 0],
45: (13)                         [0, 0, 0, 0]]
46: (8)                     )
47: (8)                     cls.A = np.array(
48: (12)                        [[0, 1, 1, 1, 0],
49: (13)                         [1, 0, 1, 0, 0],
50: (13)                         [1, 1, 0, 0, 0],
51: (13)                         [1, 0, 0, 0, 0],
52: (13)                         [0, 0, 0, 0, 0]]
53: (8)                     )
54: (8)                     cls.WG = havel_hakimi_graph(deg)
55: (8)                     cls.WG.add_edges_from(
56: (12)                        (u, v, {"weight": 0.5, "other": 0.3}) for (u, v) in cls.G.edges()
57: (8)                     )
58: (8)                     cls.WA = np.array(
59: (12)                        [[0, 0.5, 0.5, 0.5, 0],
60: (13)                         [0.5, 0, 0.5, 0, 0],
```

```
 61: (13)                         [0.5, 0.5, 0, 0, 0],
 62: (13)                         [0.5, 0, 0, 0, 0],
 63: (13)                         [0, 0, 0, 0, 0]]
 64: (8)                     )
 65: (8)                     cls.MG = nx.MultiGraph(cls.G)
 66: (8)                     cls.MG2 = cls.MG.copy()
 67: (8)                     cls.MG2.add_edge(0, 1)
 68: (8)                     cls.MG2A = np.array(
 69: (12)                        [[0, 2, 1, 1, 0],
 70: (13)                         [2, 0, 1, 0, 0],
 71: (13)                         [1, 1, 0, 0, 0],
 72: (13)                         [1, 0, 0, 0, 0],
 73: (13)                         [0, 0, 0, 0, 0]]
 74: (8)                     )
 75: (8)                     cls.MGOI = np.array(
 76: (12)                        [[-1, -1, -1, -1, 0],
 77: (13)                         [1, 1, 0, 0, -1],
 78: (13)                         [0, 0, 1, 0, 1],
 79: (13)                         [0, 0, 0, 1, 0],
 80: (13)                         [0, 0, 0, 0, 0]]
 81: (8)                     )
 82: (8)                     cls.no_edges_G = nx.Graph([(1, 2), (3, 2, {"weight": 8})])
 83: (8)                     cls.no_edges_A = np.array([[0, 0], [0, 0]])
 84: (4)                 def test_incidence_matrix(self):
 85: (8)                     "Conversion to incidence matrix"
 86: (8)                     I = nx.incidence_matrix(
 87: (12)                        self.G,
 88: (12)                        nodelist=sorted(self.G),
 89: (12)                        edgelist=sorted(self.G.edges()),
 90: (12)                        oriented=True,
 91: (12)                        dtype=int,
 92: (8)                     ).todense()
 93: (8)                     np.testing.assert_equal(I, self.OI)
 94: (8)                     I = nx.incidence_matrix(
 95: (12)                        self.G,
 96: (12)                        nodelist=sorted(self.G),
 97: (12)                        edgelist=sorted(self.G.edges()),
 98: (12)                        oriented=False,
 99: (12)                        dtype=int,
 100: (8)                    ).todense()
 101: (8)                    np.testing.assert_equal(I, np.abs(self.OI))
 102: (8)                    I = nx.incidence_matrix(
 103: (12)                       self.MG,
 104: (12)                       nodelist=sorted(self.MG),
 105: (12)                       edgelist=sorted(self.MG.edges()),
 106: (12)                       oriented=True,
 107: (12)                       dtype=int,
 108: (8)                    ).todense()
 109: (8)                    np.testing.assert_equal(I, self.OI)
 110: (8)                    I = nx.incidence_matrix(
 111: (12)                       self.MG,
 112: (12)                       nodelist=sorted(self.MG),
 113: (12)                       edgelist=sorted(self.MG.edges()),
 114: (12)                       oriented=False,
 115: (12)                       dtype=int,
 116: (8)                    ).todense()
 117: (8)                    np.testing.assert_equal(I, np.abs(self.OI))
 118: (8)                    I = nx.incidence_matrix(
 119: (12)                       self.MG2,
 120: (12)                       nodelist=sorted(self.MG2),
 121: (12)                       edgelist=sorted(self.MG2.edges()),
 122: (12)                       oriented=True,
 123: (12)                       dtype=int,
 124: (8)                    ).todense()
 125: (8)                    np.testing.assert_equal(I, self.MGOI)
 126: (8)                    I = nx.incidence_matrix(
 127: (12)                       self.MG2,
 128: (12)                       nodelist=sorted(self.MG),
 129: (12)                       edgelist=sorted(self.MG2.edges()),
```

```
(12)                        oriented=False,
(12)                        dtype=int,
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, np.abs(self.MGOI))
(8)                     I = nx.incidence_matrix(self.G, dtype=np.uint8)
(8)                     assert I.dtype == np.uint8
(4)                 def test_weighted_incidence_matrix(self):
(8)                     I = nx.incidence_matrix(
(12)                        self.WG,
(12)                        nodelist=sorted(self.WG),
(12)                        edgelist=sorted(self.WG.edges()),
(12)                        oriented=True,
(12)                        dtype=int,
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, self.OI)
(8)                     I = nx.incidence_matrix(
(12)                        self.WG,
(12)                        nodelist=sorted(self.WG),
(12)                        edgelist=sorted(self.WG.edges()),
(12)                        oriented=False,
(12)                        dtype=int,
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, np.abs(self.OI))
(8)                     I = nx.incidence_matrix(
(12)                        self.WG,
(12)                        nodelist=sorted(self.WG),
(12)                        edgelist=sorted(self.WG.edges()),
(12)                        oriented=True,
(12)                        weight="weight",
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, 0.5 * self.OI)
(8)                     I = nx.incidence_matrix(
(12)                        self.WG,
(12)                        nodelist=sorted(self.WG),
(12)                        edgelist=sorted(self.WG.edges()),
(12)                        oriented=False,
(12)                        weight="weight",
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, np.abs(0.5 * self.OI))
(8)                     I = nx.incidence_matrix(
(12)                        self.WG,
(12)                        nodelist=sorted(self.WG),
(12)                        edgelist=sorted(self.WG.edges()),
(12)                        oriented=True,
(12)                        weight="other",
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, 0.3 * self.OI)
(8)                     WMG = nx.MultiGraph(self.WG)
(8)                     WMG.add_edge(0, 1, weight=0.5, other=0.3)
(8)                     I = nx.incidence_matrix(
(12)                        WMG,
(12)                        nodelist=sorted(WMG),
(12)                        edgelist=sorted(WMG.edges(keys=True)),
(12)                        oriented=True,
(12)                        weight="weight",
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, 0.5 * self.MGOI)
(8)                     I = nx.incidence_matrix(
(12)                        WMG,
(12)                        nodelist=sorted(WMG),
(12)                        edgelist=sorted(WMG.edges(keys=True)),
(12)                        oriented=False,
(12)                        weight="weight",
(8)                     ).todense()
(8)                     np.testing.assert_equal(I, np.abs(0.5 * self.MGOI))
(8)                     I = nx.incidence_matrix(
(12)                        WMG,
(12)                        nodelist=sorted(WMG),
(12)                        edgelist=sorted(WMG.edges(keys=True)),
```

```
199: (12)                        oriented=True,
200: (12)                        weight="other",
201: (8)                    ).todense()
202: (8)                    np.testing.assert_equal(I, 0.3 * self.MGOI)
203: (4)                def test_adjacency_matrix(self):
204: (8)                    "Conversion to adjacency matrix"
205: (8)                    np.testing.assert_equal(nx.adjacency_matrix(self.G).todense(), self.A)
206: (8)                    np.testing.assert_equal(nx.adjacency_matrix(self.MG).todense(),
self.A)
207: (8)                    np.testing.assert_equal(nx.adjacency_matrix(self.MG2).todense(),
self.MG2A)
208: (8)                    np.testing.assert_equal(
209: (12)                       nx.adjacency_matrix(self.G, nodelist=[0, 1]).todense(), self.A[:2,
:2]
210: (8)                    )
211: (8)                    np.testing.assert_equal(nx.adjacency_matrix(self.WG).todense(),
self.WA)
212: (8)                    np.testing.assert_equal(
213: (12)                       nx.adjacency_matrix(self.WG, weight=None).todense(), self.A
214: (8)                    )
215: (8)                    np.testing.assert_equal(
216: (12)                       nx.adjacency_matrix(self.MG2, weight=None).todense(), self.MG2A
217: (8)                    )
218: (8)                    np.testing.assert_equal(
219: (12)                       nx.adjacency_matrix(self.WG, weight="other").todense(), 0.6 *
self.WA
220: (8)                    )
221: (8)                    np.testing.assert_equal(
222: (12)                       nx.adjacency_matrix(self.no_edges_G, nodelist=[1, 3]).todense(),
223: (12)                       self.no_edges_A,
224: (8)                    )
```

File 499 - test_spectrum.py:

```
1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.generators.degree_seq import havel_hakimi_graph
6: (0)              class TestSpectrum:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      deg = [3, 2, 2, 1, 0]
10: (8)                     cls.G = havel_hakimi_graph(deg)
11: (8)                     cls.P = nx.path_graph(3)
12: (8)                     cls.WG = nx.Graph(
13: (12)                        (u, v, {"weight": 0.5, "other": 0.3}) for (u, v) in cls.G.edges()
14: (8)                     )
15: (8)                     cls.WG.add_node(4)
16: (8)                     cls.DG = nx.DiGraph()
17: (8)                     nx.add_path(cls.DG, [0, 1, 2])
18: (4)                 def test_laplacian_spectrum(self):
19: (8)                     "Laplacian eigenvalues"
20: (8)                     evals = np.array([0, 0, 1, 3, 4])
21: (8)                     e = sorted(nx.laplacian_spectrum(self.G))
22: (8)                     np.testing.assert_almost_equal(e, evals)
23: (8)                     e = sorted(nx.laplacian_spectrum(self.WG, weight=None))
24: (8)                     np.testing.assert_almost_equal(e, evals)
25: (8)                     e = sorted(nx.laplacian_spectrum(self.WG))
26: (8)                     np.testing.assert_almost_equal(e, 0.5 * evals)
27: (8)                     e = sorted(nx.laplacian_spectrum(self.WG, weight="other"))
28: (8)                     np.testing.assert_almost_equal(e, 0.3 * evals)
29: (4)                 def test_normalized_laplacian_spectrum(self):
30: (8)                     "Normalized Laplacian eigenvalues"
31: (8)                     evals = np.array([0, 0, 0.7712864461218, 1.5, 1.7287135538781])
32: (8)                     e = sorted(nx.normalized_laplacian_spectrum(self.G))
33: (8)                     np.testing.assert_almost_equal(e, evals)
```

```
34: (8)                     e = sorted(nx.normalized_laplacian_spectrum(self.WG, weight=None))
35: (8)                     np.testing.assert_almost_equal(e, evals)
36: (8)                     e = sorted(nx.normalized_laplacian_spectrum(self.WG))
37: (8)                     np.testing.assert_almost_equal(e, evals)
38: (8)                     e = sorted(nx.normalized_laplacian_spectrum(self.WG, weight="other"))
39: (8)                     np.testing.assert_almost_equal(e, evals)
40: (4)                 def test_adjacency_spectrum(self):
41: (8)                     "Adjacency eigenvalues"
42: (8)                     evals = np.array([-np.sqrt(2), 0, np.sqrt(2)])
43: (8)                     e = sorted(nx.adjacency_spectrum(self.P))
44: (8)                     np.testing.assert_almost_equal(e, evals)
45: (4)                 def test_modularity_spectrum(self):
46: (8)                     "Modularity eigenvalues"
47: (8)                     evals = np.array([-1.5, 0.0, 0.0])
48: (8)                     e = sorted(nx.modularity_spectrum(self.P))
49: (8)                     np.testing.assert_almost_equal(e, evals)
50: (8)                     evals = np.array([-0.5, 0.0, 0.0])
51: (8)                     e = sorted(nx.modularity_spectrum(self.DG))
52: (8)                     np.testing.assert_almost_equal(e, evals)
53: (4)                 def test_bethe_hessian_spectrum(self):
54: (8)                     "Bethe Hessian eigenvalues"
55: (8)                     evals = np.array([0.5 * (9 - np.sqrt(33)), 4, 0.5 * (9 +
np.sqrt(33))])
56: (8)                     e = sorted(nx.bethe_hessian_spectrum(self.P, r=2))
57: (8)                     np.testing.assert_almost_equal(e, evals)
58: (8)                     e1 = sorted(nx.bethe_hessian_spectrum(self.P, r=1))
59: (8)                     e2 = sorted(nx.laplacian_spectrum(self.P))
60: (8)                     np.testing.assert_almost_equal(e1, e2)

----------------------------------------

File 500 - test_modularity.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.generators.degree_seq import havel_hakimi_graph
6: (0)              class TestModularity:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      deg = [3, 2, 2, 1, 0]
10: (8)                     cls.G = havel_hakimi_graph(deg)
11: (8)                     cls.DG = nx.DiGraph()
12: (8)                     cls.DG.add_edges_from(
13: (12)                        (
14: (16)                            (1, 2),
15: (16)                            (1, 3),
16: (16)                            (3, 1),
17: (16)                            (3, 2),
18: (16)                            (3, 5),
19: (16)                            (4, 5),
20: (16)                            (4, 6),
21: (16)                            (5, 4),
22: (16)                            (5, 6),
23: (16)                            (6, 4),
24: (12)                        )
25: (8)                     )
26: (4)                 def test_modularity(self):
27: (8)                     "Modularity matrix"
28: (8)                     B = np.array([[-1.125,  0.25,  0.25,  0.625,  0.],
29: (22)                                  [0.25, -0.5,  0.5, -0.25,  0.],
30: (22)                                  [0.25,  0.5, -0.5, -0.25,  0.],
31: (22)                                  [0.625, -0.25, -0.25, -0.125,  0.],
32: (22)                                  [0.,  0.,  0.,  0.,  0.]])
33: (8)                     permutation = [4, 0, 1, 2, 3]
34: (8)                     np.testing.assert_equal(nx.modularity_matrix(self.G), B)
35: (8)                     np.testing.assert_equal(
36: (12)                        nx.modularity_matrix(self.G, nodelist=permutation),
```

```
37: (12)                        B[np.ix_(permutation, permutation)],
38: (8)                     )
39: (4)                 def test_modularity_weight(self):
40: (8)                     "Modularity matrix with weights"
41: (8)                     B = np.array([[-1.125,  0.25,  0.25,  0.625,  0.],
42: (22)                                  [0.25, -0.5,  0.5, -0.25,  0.],
43: (22)                                  [0.25,  0.5, -0.5, -0.25,  0.],
44: (22)                                  [0.625, -0.25, -0.25, -0.125,  0.],
45: (22)                                  [0.,  0.,  0.,  0.,  0.]])
46: (8)                     G_weighted = self.G.copy()
47: (8)                     for n1, n2 in G_weighted.edges():
48: (12)                        G_weighted.edges[n1, n2]["weight"] = 0.5
49: (8)                     np.testing.assert_equal(nx.modularity_matrix(G_weighted), B)
50: (8)                     np.testing.assert_equal(
51: (12)                        nx.modularity_matrix(G_weighted, weight="weight"), 0.5 * B
52: (8)                     )
53: (4)                 def test_directed_modularity(self):
54: (8)                     "Directed Modularity matrix"
55: (8)                     B = np.array([[-0.2,  0.6,  0.8, -0.4, -0.4, -0.4],
56: (22)                                  [0.,  0.,  0.,  0.,  0.,  0.],
57: (22)                                  [0.7,  0.4, -0.3, -0.6,  0.4, -0.6],
58: (22)                                  [-0.2, -0.4, -0.2, -0.4,  0.6,  0.6],
59: (22)                                  [-0.2, -0.4, -0.2,  0.6, -0.4,  0.6],
60: (22)                                  [-0.1, -0.2, -0.1,  0.8, -0.2, -0.2]])
61: (8)                     node_permutation = [5, 1, 2, 3, 4, 6]
62: (8)                     idx_permutation = [4, 0, 1, 2, 3, 5]
63: (8)                     mm = nx.directed_modularity_matrix(self.DG, nodelist=sorted(self.DG))
64: (8)                     np.testing.assert_equal(mm, B)
65: (8)                     np.testing.assert_equal(
66: (12)                        nx.directed_modularity_matrix(self.DG, nodelist=node_permutation),
67: (12)                        B[np.ix_(idx_permutation, idx_permutation)],
68: (8)                     )
```

File 501 - adjlist.py:

```
1: (0)              """
2: (0)              **************
3: (0)              Adjacency List
4: (0)              **************
5: (0)              Read and write NetworkX graphs as adjacency lists.
6: (0)              Adjacency list format is useful for graphs without data associated
7: (0)              with nodes or edges and for nodes that can be meaningfully represented
8: (0)              as strings.
9: (0)              Format
10: (0)             ------
11: (0)             The adjacency list format consists of lines with node labels.  The
12: (0)             first label in a line is the source node.  Further labels in the line
13: (0)             are considered target nodes and are added to the graph along with an edge
14: (0)             between the source node and target node.
15: (0)             The graph with edges a-b, a-c, d-e can be represented as the following
16: (0)             adjacency list (anything following the # in a line is a comment)::
17: (5)                  a b c # source target target
18: (5)                  d e
19: (0)             """
20: (0)             __all__ = ["generate_adjlist", "write_adjlist", "parse_adjlist",
"read_adjlist"]
21: (0)             import networkx as nx
22: (0)             from networkx.utils import open_file
23: (0)             def generate_adjlist(G, delimiter=" "):
24: (4)                 """Generate a single line of the graph G in adjacency list format.
25: (4)                 Parameters
26: (4)                 ----------
27: (4)                 G : NetworkX graph
28: (4)                 delimiter : string, optional
29: (7)                    Separator for node labels
30: (4)                 Returns
31: (4)                 -------
```

```
(4)                 lines : string
(8)                     Lines of data in adjlist format.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.lollipop_graph(4, 3)
(4)                 >>> for line in nx.generate_adjlist(G):
(4)                 ...     print(line)
(4)                 0 1 2 3
(4)                 1 2 3
(4)                 2 3
(4)                 3 4
(4)                 4 5
(4)                 5 6
(4)                 6
(4)                 See Also
(4)                 --------
(4)                 write_adjlist, read_adjlist
(4)                 Notes
(4)                 -----
(4)                 The default `delimiter=" "` will result in unexpected results if node names contain
(4)                 whitespace characters. To avoid this problem, specify an alternate delimiter when spaces are
(4)                 valid in node names.
(4)                 NB: This option is not available for data that isn't user-generated.
(4)                 """
(4)                 directed = G.is_directed()
(4)                 seen = set()
(4)                 for s, nbrs in G.adjacency():
(8)                     line = str(s) + delimiter
(8)                     for t, data in nbrs.items():
(12)                        if not directed and t in seen:
(16)                            continue
(12)                        if G.is_multigraph():
(16)                            for d in data.values():
(20)                                line += str(t) + delimiter
(12)                        else:
(16)                            line += str(t) + delimiter
(8)                     if not directed:
(12)                        seen.add(s)
(8)                     yield line[: -len(delimiter)]
(0)             @open_file(1, mode="wb")
(0)             def write_adjlist(G, path, comments="#", delimiter=" ", encoding="utf-8"):
(4)                 """Write graph G in single-line adjacency-list format to path.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(4)                 path : string or file
(7)                    Filename or file handle for data output.
(7)                    Filenames ending in .gz or .bz2 will be compressed.
(4)                 comments : string, optional
(7)                    Marker for comment lines
(4)                 delimiter : string, optional
(7)                    Separator for node labels
(4)                 encoding : string, optional
(7)                    Text encoding.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> nx.write_adjlist(G, "test.adjlist")
(4)                 The path can be a filehandle or a string with the name of the file. If a
(4)                 filehandle is provided, it has to be opened in 'wb' mode.
(4)                 >>> fh = open("test.adjlist", "wb")
(4)                 >>> nx.write_adjlist(G, fh)
(4)                 Notes
(4)                 -----
(4)                 The default `delimiter=" "` will result in unexpected results if node names contain
(4)                 whitespace characters. To avoid this problem, specify an alternate
```

```
delimiter when spaces are
(4)                 valid in node names.
(4)                 NB: This option is not available for data that isn't user-generated.
(4)                 This format does not store graph, node, or edge data.
(4)                 See Also
(4)                 --------
(4)                 read_adjlist, generate_adjlist
(4)                 """
(4)                 import sys
(4)                 import time
(4)                 pargs = comments + " ".join(sys.argv) + "\n"
(4)                 header = (
(8)                     pargs
(8)                     + comments
(8)                     + f" GMT {time.asctime(time.gmtime())}\n"
(8)                     + comments
(8)                     + f" {G.name}\n"
(4)                 )
(4)                 path.write(header.encode(encoding))
(4)                 for line in generate_adjlist(G, delimiter):
(8)                     line += "\n"
(8)                     path.write(line.encode(encoding))
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def parse_adjlist(
(4)                 lines, comments="#", delimiter=None, create_using=None, nodetype=None
(0)             ):
(4)                 """Parse lines of a graph adjacency list representation.
(4)                 Parameters
(4)                 ----------
(4)                 lines : list or iterator of strings
(8)                     Input data in adjlist format
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 nodetype : Python type, optional
(7)                    Convert nodes to this type.
(4)                 comments : string, optional
(7)                    Marker for comment lines
(4)                 delimiter : string, optional
(7)                    Separator for node labels.  The default is whitespace.
(4)                 Returns
(4)                 -------
(4)                 G: NetworkX graph
(8)                     The graph corresponding to the lines in adjacency list format.
(4)                 Examples
(4)                 --------
(4)                 >>> lines = ["1 2 5", "2 3 4", "3 5", "4", "5"]
(4)                 >>> G = nx.parse_adjlist(lines, nodetype=int)
(4)                 >>> nodes = [1, 2, 3, 4, 5]
(4)                 >>> all(node in G for node in nodes)
(4)                 True
(4)                 >>> edges = [(1, 2), (1, 5), (2, 3), (2, 4), (3, 5)]
(4)                 >>> all((u, v) in G.edges() or (v, u) in G.edges() for (u, v) in edges)
(4)                 True
(4)                 See Also
(4)                 --------
(4)                 read_adjlist
(4)                 """
(4)                 G = nx.empty_graph(0, create_using)
(4)                 for line in lines:
(8)                     p = line.find(comments)
(8)                     if p >= 0:
(12)                        line = line[:p]
(8)                     if not len(line):
(12)                        continue
(8)                     vlist = line.strip().split(delimiter)
(8)                     u = vlist.pop(0)
(8)                     if nodetype is not None:
(12)                        try:
(16)                            u = nodetype(u)
```

```
166: (12)                        except BaseException as err:
167: (16)                            raise TypeError(
168: (20)                                f"Failed to convert node ({u}) to type {nodetype}"
169: (16)                            ) from err
170: (8)                    G.add_node(u)
171: (8)                    if nodetype is not None:
172: (12)                        try:
173: (16)                            vlist = list(map(nodetype, vlist))
174: (12)                        except BaseException as err:
175: (16)                            raise TypeError(
176: (20)                                f"Failed to convert nodes ({','.join(vlist)}) to type
{nodetype}"
177: (16)                            ) from err
178: (8)                    G.add_edges_from([(u, v) for v in vlist])
179: (4)                return G
180: (0)            @open_file(0, mode="rb")
181: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
182: (0)            def read_adjlist(
183: (4)                path,
184: (4)                comments="#",
185: (4)                delimiter=None,
186: (4)                create_using=None,
187: (4)                nodetype=None,
188: (4)                encoding="utf-8",
189: (0)            ):
190: (4)                """Read graph in adjacency list format from path.
191: (4)                Parameters
192: (4)                ----------
193: (4)                path : string or file
194: (7)                   Filename or file handle to read.
195: (7)                   Filenames ending in .gz or .bz2 will be uncompressed.
196: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
197: (7)                   Graph type to create. If graph instance, then cleared before populated.
198: (4)                nodetype : Python type, optional
199: (7)                   Convert nodes to this type.
200: (4)                comments : string, optional
201: (7)                   Marker for comment lines
202: (4)                delimiter : string, optional
203: (7)                   Separator for node labels.  The default is whitespace.
204: (4)                Returns
205: (4)                -------
206: (4)                G: NetworkX graph
207: (8)                    The graph corresponding to the lines in adjacency list format.
208: (4)                Examples
209: (4)                --------
210: (4)                >>> G = nx.path_graph(4)
211: (4)                >>> nx.write_adjlist(G, "test.adjlist")
212: (4)                >>> G = nx.read_adjlist("test.adjlist")
213: (4)                The path can be a filehandle or a string with the name of the file. If a
214: (4)                filehandle is provided, it has to be opened in 'rb' mode.
215: (4)                >>> fh = open("test.adjlist", "rb")
216: (4)                >>> G = nx.read_adjlist(fh)
217: (4)                Filenames ending in .gz or .bz2 will be compressed.
218: (4)                >>> nx.write_adjlist(G, "test.adjlist.gz")
219: (4)                >>> G = nx.read_adjlist("test.adjlist.gz")
220: (4)                The optional nodetype is a function to convert node strings to nodetype.
221: (4)                For example
222: (4)                >>> G = nx.read_adjlist("test.adjlist", nodetype=int)
223: (4)                will attempt to convert all nodes to integer type.
224: (4)                Since nodes must be hashable, the function nodetype must return hashable
225: (4)                types (e.g. int, float, str, frozenset - or tuples of those, etc.)
226: (4)                The optional create_using parameter indicates the type of NetworkX graph
227: (4)                created.  The default is `nx.Graph`, an undirected graph.
228: (4)                To read the data as a directed graph use
229: (4)                >>> G = nx.read_adjlist("test.adjlist", create_using=nx.DiGraph)
230: (4)                Notes
231: (4)                -----
232: (4)                This format does not store graph or node data.
233: (4)                See Also
```

```
234: (4)                --------
235: (4)                write_adjlist
236: (4)                """
237: (4)                lines = (line.decode(encoding) for line in path)
238: (4)                return parse_adjlist(
239: (8)                    lines,
240: (8)                    comments=comments,
241: (8)                    delimiter=delimiter,
242: (8)                    create_using=create_using,
243: (8)                    nodetype=nodetype,
244: (4)                )

----------------------------------------

File 502 - __init__.py:

1: (0)              """
2: (0)              A package for reading and writing graphs in various formats.
3: (0)              """
4: (0)              from networkx.readwrite.adjlist import *
5: (0)              from networkx.readwrite.multiline_adjlist import *
6: (0)              from networkx.readwrite.edgelist import *
7: (0)              from networkx.readwrite.pajek import *
8: (0)              from networkx.readwrite.leda import *
9: (0)              from networkx.readwrite.sparse6 import *
10: (0)             from networkx.readwrite.graph6 import *
11: (0)             from networkx.readwrite.gml import *
12: (0)             from networkx.readwrite.graphml import *
13: (0)             from networkx.readwrite.gexf import *
14: (0)             from networkx.readwrite.json_graph import *
15: (0)             from networkx.readwrite.text import *

----------------------------------------

File 503 - edgelist.py:

1: (0)              """
2: (0)              **********
3: (0)              Edge Lists
4: (0)              **********
5: (0)              Read and write NetworkX graphs as edge lists.
6: (0)              The multi-line adjacency list format is useful for graphs with nodes
7: (0)              that can be meaningfully represented as strings.  With the edgelist
8: (0)              format simple edge data can be stored but node or graph data is not.
9: (0)              There is no way of representing isolated nodes unless the node has a
10: (0)             self-loop edge.
11: (0)             Format
12: (0)             ------
13: (0)             You can read or write three formats of edge lists with these functions.
14: (0)             Node pairs with no data::
15: (1)              1 2
16: (0)             Python dictionary as data::
17: (1)              1 2 {'weight':7, 'color':'green'}
18: (0)             Arbitrary data::
19: (1)              1 2 7 green
20: (0)             """
21: (0)             __all__ = [
22: (4)                 "generate_edgelist",
23: (4)                 "write_edgelist",
24: (4)                 "parse_edgelist",
25: (4)                 "read_edgelist",
26: (4)                 "read_weighted_edgelist",
27: (4)                 "write_weighted_edgelist",
28: (0)             ]
29: (0)             import networkx as nx
30: (0)             from networkx.utils import open_file
31: (0)             def generate_edgelist(G, delimiter=" ", data=True):
32: (4)                 """Generate a single line of the graph G in edge list format.
33: (4)                 Parameters
```

```
    ----------
    G : NetworkX graph
    delimiter : string, optional
       Separator for node labels
    data : bool or list of keys
       If False generate no edge data.  If True use a dictionary
       representation of edge data.  If a list of keys use a list of data
       values corresponding to the keys.
    Returns
    -------
    lines : string
        Lines of data in adjlist format.
    Examples
    --------
    >>> G = nx.lollipop_graph(4, 3)
    >>> G[1][2]["weight"] = 3
    >>> G[3][4]["capacity"] = 12
    >>> for line in nx.generate_edgelist(G, data=False):
    ...     print(line)
    0 1
    0 2
    0 3
    1 2
    1 3
    2 3
    3 4
    4 5
    5 6
    >>> for line in nx.generate_edgelist(G):
    ...     print(line)
    0 1 {}
    0 2 {}
    0 3 {}
    1 2 {'weight': 3}
    1 3 {}
    2 3 {}
    3 4 {'capacity': 12}
    4 5 {}
    5 6 {}
    >>> for line in nx.generate_edgelist(G, data=["weight"]):
    ...     print(line)
    0 1
    0 2
    0 3
    1 2 3
    1 3
    2 3
    3 4
    4 5
    5 6
    See Also
    --------
    write_adjlist, read_adjlist
    """
    if data is True:
        for u, v, d in G.edges(data=True):
            e = u, v, dict(d)
            yield delimiter.join(map(str, e))
    elif data is False:
        for u, v in G.edges(data=False):
            e = u, v
            yield delimiter.join(map(str, e))
    else:
        for u, v, d in G.edges(data=True):
            e = [u, v]
            try:
                e.extend(d[k] for k in data)
            except KeyError:
                pass  # missing data for this edge, should warn?
```

```
(12)                        yield delimiter.join(map(str, e))
(0)             @open_file(1, mode="wb")
(0)             def write_edgelist(G, path, comments="#", delimiter=" ", data=True, encoding="utf-8"):
(4)                 """Write graph as a list of edges.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A NetworkX graph
(4)                 path : file or string
(7)                    File or filename to write. If a file is provided, it must be
(7)                    opened in 'wb' mode. Filenames ending in .gz or .bz2 will be compressed.
(4)                 comments : string, optional
(7)                    The character used to indicate the start of a comment
(4)                 delimiter : string, optional
(7)                    The string used to separate values.  The default is whitespace.
(4)                 data : bool or list, optional
(7)                    If False write no edge data.
(7)                    If True write a string representation of the edge data dictionary..
(7)                    If a list (or other iterable) is provided, write the  keys specified
(7)                    in the list.
(4)                 encoding: string, optional
(7)                    Specify which encoding to use when writing file.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> nx.write_edgelist(G, "test.edgelist")
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> fh = open("test.edgelist", "wb")
(4)                 >>> nx.write_edgelist(G, fh)
(4)                 >>> nx.write_edgelist(G, "test.edgelist.gz")
(4)                 >>> nx.write_edgelist(G, "test.edgelist.gz", data=False)
(4)                 >>> G = nx.Graph()
(4)                 >>> G.add_edge(1, 2, weight=7, color="red")
(4)                 >>> nx.write_edgelist(G, "test.edgelist", data=False)
(4)                 >>> nx.write_edgelist(G, "test.edgelist", data=["color"])
(4)                 >>> nx.write_edgelist(G, "test.edgelist", data=["color", "weight"])
(4)                 See Also
(4)                 --------
(4)                 read_edgelist
(4)                 write_weighted_edgelist
(4)                 """
(4)                 for line in generate_edgelist(G, delimiter, data):
(8)                     line += "\n"
(8)                     path.write(line.encode(encoding))
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def parse_edgelist(
(4)                 lines, comments="#", delimiter=None, create_using=None, nodetype=None, data=True
(0)             ):
(4)                 """Parse lines of an edge list representation of a graph.
(4)                 Parameters
(4)                 ----------
(4)                 lines : list or iterator of strings
(8)                     Input data in edgelist format
(4)                 comments : string, optional
(7)                    Marker for comment lines. Default is `'#'`. To specify that no character
(7)                    should be treated as a comment, use ``comments=None``.
(4)                 delimiter : string, optional
(7)                    Separator for node labels. Default is `None`, meaning any whitespace.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 nodetype : Python type, optional
(7)                    Convert nodes to this type. Default is `None`, meaning no conversion is
(7)                    performed.
(4)                 data : bool or list of (label,type) tuples
(7)                    If `False` generate no edge data or if `True` use a dictionary
```

```
(7)                 representation of edge data or a list tuples specifying dictionary
(7)                 key names and types for edge data.
(4)             Returns
(4)             -------
(4)             G: NetworkX Graph
(8)                 The graph corresponding to lines
(4)             Examples
(4)             --------
(4)             Edgelist with no data:
(4)             >>> lines = ["1 2", "2 3", "3 4"]
(4)             >>> G = nx.parse_edgelist(lines, nodetype=int)
(4)             >>> list(G)
(4)             [1, 2, 3, 4]
(4)             >>> list(G.edges())
(4)             [(1, 2), (2, 3), (3, 4)]
(4)             Edgelist with data in Python dictionary representation:
(4)             >>> lines = ["1 2 {'weight': 3}", "2 3 {'weight': 27}", "3 4 {'weight':
3.0}"]
(4)             >>> G = nx.parse_edgelist(lines, nodetype=int)
(4)             >>> list(G)
(4)             [1, 2, 3, 4]
(4)             >>> list(G.edges(data=True))
(4)             [(1, 2, {'weight': 3}), (2, 3, {'weight': 27}), (3, 4, {'weight': 3.0})]
(4)             Edgelist with data in a list:
(4)             >>> lines = ["1 2 3", "2 3 27", "3 4 3.0"]
(4)             >>> G = nx.parse_edgelist(lines, nodetype=int, data=(("weight", float),))
(4)             >>> list(G)
(4)             [1, 2, 3, 4]
(4)             >>> list(G.edges(data=True))
(4)             [(1, 2, {'weight': 3.0}), (2, 3, {'weight': 27.0}), (3, 4, {'weight':
3.0})]
(4)             See Also
(4)             --------
(4)             read_weighted_edgelist
(4)             """
(4)             from ast import literal_eval
(4)             G = nx.empty_graph(0, create_using)
(4)             for line in lines:
(8)                 if comments is not None:
(12)                    p = line.find(comments)
(12)                    if p >= 0:
(16)                        line = line[:p]
(12)                    if not line:
(16)                        continue
(8)                 s = line.strip().split(delimiter)
(8)                 if len(s) < 2:
(12)                    continue
(8)                 u = s.pop(0)
(8)                 v = s.pop(0)
(8)                 d = s
(8)                 if nodetype is not None:
(12)                    try:
(16)                        u = nodetype(u)
(16)                        v = nodetype(v)
(12)                    except Exception as err:
(16)                        raise TypeError(
(20)                            f"Failed to convert nodes {u},{v} to type {nodetype}."
(16)                        ) from err
(8)                 if len(d) == 0 or data is False:
(12)                    edgedata = {}
(8)                 elif data is True:
(12)                    try:  # try to evaluate as dictionary
(16)                        if delimiter == ",":
(20)                            edgedata_str = ",".join(d)
(16)                        else:
(20)                            edgedata_str = " ".join(d)
(16)                        edgedata = dict(literal_eval(edgedata_str.strip()))
(12)                    except Exception as err:
(16)                        raise TypeError(
```

```
235: (20)                                f"Failed to convert edge data ({d}) to dictionary."
236: (16)                            ) from err
237: (8)                     else:
238: (12)                        if len(d) != len(data):
239: (16)                            raise IndexError(
240: (20)                                f"Edge data {d} and data_keys {data} are not the same
length"
241: (16)                            )
242: (12)                        edgedata = {}
243: (12)                        for (edge_key, edge_type), edge_value in zip(data, d):
244: (16)                            try:
245: (20)                                edge_value = edge_type(edge_value)
246: (16)                            except Exception as err:
247: (20)                                raise TypeError(
248: (24)                                    f"Failed to convert {edge_key} data {edge_value} "
249: (24)                                    f"to type {edge_type}."
250: (20)                                ) from err
251: (16)                            edgedata.update({edge_key: edge_value})
252: (8)                     G.add_edge(u, v, **edgedata)
253: (4)                 return G
254: (0)             @open_file(0, mode="rb")
255: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
256: (0)             def read_edgelist(
257: (4)                 path,
258: (4)                 comments="#",
259: (4)                 delimiter=None,
260: (4)                 create_using=None,
261: (4)                 nodetype=None,
262: (4)                 data=True,
263: (4)                 edgetype=None,
264: (4)                 encoding="utf-8",
265: (0)             ):
266: (4)                 """Read a graph from a list of edges.
267: (4)                 Parameters
268: (4)                 ----------
269: (4)                 path : file or string
270: (7)                    File or filename to read. If a file is provided, it must be
271: (7)                    opened in 'rb' mode.
272: (7)                    Filenames ending in .gz or .bz2 will be uncompressed.
273: (4)                 comments : string, optional
274: (7)                    The character used to indicate the start of a comment. To specify that
275: (7)                    no character should be treated as a comment, use ``comments=None``.
276: (4)                 delimiter : string, optional
277: (7)                    The string used to separate values.  The default is whitespace.
278: (4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
279: (7)                    Graph type to create. If graph instance, then cleared before populated.
280: (4)                 nodetype : int, float, str, Python type, optional
281: (7)                    Convert node data from strings to specified type
282: (4)                 data : bool or list of (label,type) tuples
283: (7)                    Tuples specifying dictionary key names and types for edge data
284: (4)                 edgetype : int, float, str, Python type, optional OBSOLETE
285: (7)                    Convert edge data from strings to specified type and use as 'weight'
286: (4)                 encoding: string, optional
287: (7)                    Specify which encoding to use when reading file.
288: (4)                 Returns
289: (4)                 -------
290: (4)                 G : graph
291: (7)                    A networkx Graph or other type specified with create_using
292: (4)                 Examples
293: (4)                 --------
294: (4)                 >>> nx.write_edgelist(nx.path_graph(4), "test.edgelist")
295: (4)                 >>> G = nx.read_edgelist("test.edgelist")
296: (4)                 >>> fh = open("test.edgelist", "rb")
297: (4)                 >>> G = nx.read_edgelist(fh)
298: (4)                 >>> fh.close()
299: (4)                 >>> G = nx.read_edgelist("test.edgelist", nodetype=int)
300: (4)                 >>> G = nx.read_edgelist("test.edgelist", create_using=nx.DiGraph)
301: (4)                 Edgelist with data in a list:
302: (4)                 >>> textline = "1 2 3"
```

```
(4)                >>> fh = open("test.edgelist", "w")
(4)                >>> d = fh.write(textline)
(4)                >>> fh.close()
(4)                >>> G = nx.read_edgelist("test.edgelist", nodetype=int, data=(("weight",
float),))
(4)                >>> list(G)
(4)                [1, 2]
(4)                >>> list(G.edges(data=True))
(4)                [(1, 2, {'weight': 3.0})]
(4)                See parse_edgelist() for more examples of formatting.
(4)                See Also
(4)                --------
(4)                parse_edgelist
(4)                write_edgelist
(4)                Notes
(4)                -----
(4)                Since nodes must be hashable, the function nodetype must return hashable
(4)                types (e.g. int, float, str, frozenset - or tuples of those, etc.)
(4)                """
(4)                lines = (line if isinstance(line, str) else line.decode(encoding) for line
in path)
(4)                return parse_edgelist(
(8)                    lines,
(8)                    comments=comments,
(8)                    delimiter=delimiter,
(8)                    create_using=create_using,
(8)                    nodetype=nodetype,
(8)                    data=data,
(4)                )
(0)            def write_weighted_edgelist(G, path, comments="#", delimiter=" ",
encoding="utf-8"):
(4)                """Write graph G as a list of edges with numeric weights.
(4)                Parameters
(4)                ----------
(4)                G : graph
(7)                   A NetworkX graph
(4)                path : file or string
(7)                   File or filename to write. If a file is provided, it must be
(7)                   opened in 'wb' mode.
(7)                   Filenames ending in .gz or .bz2 will be compressed.
(4)                comments : string, optional
(7)                   The character used to indicate the start of a comment
(4)                delimiter : string, optional
(7)                   The string used to separate values.  The default is whitespace.
(4)                encoding: string, optional
(7)                   Specify which encoding to use when writing file.
(4)                Examples
(4)                --------
(4)                >>> G = nx.Graph()
(4)                >>> G.add_edge(1, 2, weight=7)
(4)                >>> nx.write_weighted_edgelist(G, "test.weighted.edgelist")
(4)                See Also
(4)                --------
(4)                read_edgelist
(4)                write_edgelist
(4)                read_weighted_edgelist
(4)                """
(4)                write_edgelist(
(8)                    G,
(8)                    path,
(8)                    comments=comments,
(8)                    delimiter=delimiter,
(8)                    data=("weight",),
(8)                    encoding=encoding,
(4)                )
(0)            @nx._dispatchable(graphs=None, returns_graph=True)
(0)            def read_weighted_edgelist(
(4)                path,
(4)                comments="#",
```

```
369: (4)                delimiter=None,
370: (4)                create_using=None,
371: (4)                nodetype=None,
372: (4)                encoding="utf-8",
373: (0)            ):
374: (4)                """Read a graph as list of edges with numeric weights.
375: (4)                Parameters
376: (4)                ----------
377: (4)                path : file or string
378: (7)                   File or filename to read. If a file is provided, it must be
379: (7)                   opened in 'rb' mode.
380: (7)                   Filenames ending in .gz or .bz2 will be uncompressed.
381: (4)                comments : string, optional
382: (7)                   The character used to indicate the start of a comment.
383: (4)                delimiter : string, optional
384: (7)                   The string used to separate values.  The default is whitespace.
385: (4)                create_using : NetworkX graph constructor, optional (default=nx.Graph)
386: (7)                   Graph type to create. If graph instance, then cleared before populated.
387: (4)                nodetype : int, float, str, Python type, optional
388: (7)                   Convert node data from strings to specified type
389: (4)                encoding: string, optional
390: (7)                   Specify which encoding to use when reading file.
391: (4)                Returns
392: (4)                -------
393: (4)                G : graph
394: (7)                   A networkx Graph or other type specified with create_using
395: (4)                Notes
396: (4)                -----
397: (4)                Since nodes must be hashable, the function nodetype must return hashable
398: (4)                types (e.g. int, float, str, frozenset - or tuples of those, etc.)
399: (4)                Example edgelist file format.
400: (4)                With numeric edge data::
401: (5)                 a b 1
402: (5)                 a c 3.14159
403: (5)                 d e 42
404: (4)                See Also
405: (4)                --------
406: (4)                write_weighted_edgelist
407: (4)                """
408: (4)                return read_edgelist(
409: (8)                    path,
410: (8)                    comments=comments,
411: (8)                    delimiter=delimiter,
412: (8)                    create_using=create_using,
413: (8)                    nodetype=nodetype,
414: (8)                    data=(("weight", float),),
415: (8)                    encoding=encoding,
416: (4)                )

----------------------------------------

File 504 - gexf.py:

1: (0)              """Read and write graphs in GEXF format.
2: (0)              .. warning::
3: (4)                  This parser uses the standard xml library present in Python, which is
4: (4)                  insecure - see :external+python:mod:`xml` for additional information.
5: (4)                  Only parse GEFX files you trust.
6: (0)              GEXF (Graph Exchange XML Format) is a language for describing complex
7: (0)              network structures, their associated data and dynamics.
8: (0)              This implementation does not support mixed graphs (directed and
9: (0)              undirected edges together).
10: (0)             Format
11: (0)             ------
12: (0)             GEXF is an XML format.  See http://gexf.net/schema.html for the
13: (0)             specification and http://gexf.net/basic.html for examples.
14: (0)             """
15: (0)             import itertools
16: (0)             import time
```

```
 17: (0)             from xml.etree.ElementTree import (
 18: (4)                 Element,
 19: (4)                 ElementTree,
 20: (4)                 SubElement,
 21: (4)                 register_namespace,
 22: (4)                 tostring,
 23: (0)             )
 24: (0)             import networkx as nx
 25: (0)             from networkx.utils import open_file
 26: (0)             __all__ = ["write_gexf", "read_gexf", "relabel_gexf_graph", "generate_gexf"]
 27: (0)             @open_file(1, mode="wb")
 28: (0)             def write_gexf(G, path, encoding="utf-8", prettyprint=True,
version="1.2draft"):
 29: (4)                 """Write G in GEXF format to path.
 30: (4)                 "GEXF (Graph Exchange XML Format) is a language for describing
 31: (4)                 complex networks structures, their associated data and dynamics" [1]_.
 32: (4)                 Node attributes are checked according to the version of the GEXF
 33: (4)                 schemas used for parameters which are not user defined,
 34: (4)                 e.g. visualization 'viz' [2]_. See example for usage.
 35: (4)                 Parameters
 36: (4)                 ----------
 37: (4)                 G : graph
 38: (7)                    A NetworkX graph
 39: (4)                 path : file or string
 40: (7)                    File or file name to write.
 41: (7)                    File names ending in .gz or .bz2 will be compressed.
 42: (4)                 encoding : string (optional, default: 'utf-8')
 43: (7)                    Encoding for text data.
 44: (4)                 prettyprint : bool (optional, default: True)
 45: (7)                    If True use line breaks and indenting in output XML.
 46: (4)                 version: string (optional, default: '1.2draft')
 47: (7)                    The version of GEXF to be used for nodes attributes checking
 48: (4)                 Examples
 49: (4)                 --------
 50: (4)                 >>> G = nx.path_graph(4)
 51: (4)                 >>> nx.write_gexf(G, "test.gexf")
 52: (4)                 >>> G.nodes[0]["viz"] = {"size": 54}
 53: (4)                 >>> G.nodes[0]["viz"]["position"] = {"x": 0, "y": 1}
 54: (4)                 >>> G.nodes[0]["viz"]["color"] = {"r": 0, "g": 0, "b": 256}
 55: (4)                 Notes
 56: (4)                 -----
 57: (4)                 This implementation does not support mixed graphs (directed and undirected
 58: (4)                 edges together).
 59: (4)                 The node id attribute is set to be the string of the node label.
 60: (4)                 If you want to specify an id use set it as node data, e.g.
 61: (4)                 node['a']['id']=1 to set the id of node 'a' to 1.
 62: (4)                 References
 63: (4)                 ----------
 64: (4)                 .. [1] GEXF File Format, http://gexf.net/
 65: (4)                 .. [2] GEXF schema, http://gexf.net/schema.html
 66: (4)                 """
 67: (4)                 writer = GEXFWriter(encoding=encoding, prettyprint=prettyprint,
version=version)
 68: (4)                 writer.add_graph(G)
 69: (4)                 writer.write(path)
 70: (0)             def generate_gexf(G, encoding="utf-8", prettyprint=True, version="1.2draft"):
 71: (4)                 """Generate lines of GEXF format representation of G.
 72: (4)                 "GEXF (Graph Exchange XML Format) is a language for describing
 73: (4)                 complex networks structures, their associated data and dynamics" [1]_.
 74: (4)                 Parameters
 75: (4)                 ----------
 76: (4)                 G : graph
 77: (4)                 A NetworkX graph
 78: (4)                 encoding : string (optional, default: 'utf-8')
 79: (4)                 Encoding for text data.
 80: (4)                 prettyprint : bool (optional, default: True)
 81: (4)                 If True use line breaks and indenting in output XML.
 82: (4)                 version : string (default: 1.2draft)
 83: (4)                 Version of GEFX File Format (see http://gexf.net/schema.html)
```

```
 84: (4)                Supported values: "1.1draft", "1.2draft"
 85: (4)                Examples
 86: (4)                --------
 87: (4)                >>> G = nx.path_graph(4)
 88: (4)                >>> linefeed = chr(10)  # linefeed=\n
 89: (4)                >>> s = linefeed.join(nx.generate_gexf(G))
 90: (4)                >>> for line in nx.generate_gexf(G):  # doctest: +SKIP
 91: (4)                ...     print(line)
 92: (4)                Notes
 93: (4)                -----
 94: (4)                This implementation does not support mixed graphs (directed and undirected
 95: (4)                edges together).
 96: (4)                The node id attribute is set to be the string of the node label.
 97: (4)                If you want to specify an id use set it as node data, e.g.
 98: (4)                node['a']['id']=1 to set the id of node 'a' to 1.
 99: (4)                References
 100: (4)               ----------
 101: (4)               .. [1] GEXF File Format, https://gephi.org/gexf/format/
 102: (4)               """
 103: (4)               writer = GEXFWriter(encoding=encoding, prettyprint=prettyprint,
version=version)
 104: (4)               writer.add_graph(G)
 105: (4)               yield from str(writer).splitlines()
 106: (0)           @open_file(0, mode="rb")
 107: (0)           @nx._dispatchable(graphs=None, returns_graph=True)
 108: (0)           def read_gexf(path, node_type=None, relabel=False, version="1.2draft"):
 109: (4)               """Read graph in GEXF format from path.
 110: (4)               "GEXF (Graph Exchange XML Format) is a language for describing
 111: (4)               complex networks structures, their associated data and dynamics" [1]_.
 112: (4)               Parameters
 113: (4)               ----------
 114: (4)               path : file or string
 115: (7)                  File or file name to read.
 116: (7)                  File names ending in .gz or .bz2 will be decompressed.
 117: (4)               node_type: Python type (default: None)
 118: (7)                  Convert node ids to this type if not None.
 119: (4)               relabel : bool (default: False)
 120: (7)                  If True relabel the nodes to use the GEXF node "label" attribute
 121: (7)                  instead of the node "id" attribute as the NetworkX node label.
 122: (4)               version : string (default: 1.2draft)
 123: (4)               Version of GEFX File Format (see http://gexf.net/schema.html)
 124: (7)                  Supported values: "1.1draft", "1.2draft"
 125: (4)               Returns
 126: (4)               -------
 127: (4)               graph: NetworkX graph
 128: (8)                   If no parallel edges are found a Graph or DiGraph is returned.
 129: (8)                   Otherwise a MultiGraph or MultiDiGraph is returned.
 130: (4)               Notes
 131: (4)               -----
 132: (4)               This implementation does not support mixed graphs (directed and undirected
 133: (4)               edges together).
 134: (4)               References
 135: (4)               ----------
 136: (4)               .. [1] GEXF File Format, http://gexf.net/
 137: (4)               """
 138: (4)               reader = GEXFReader(node_type=node_type, version=version)
 139: (4)               if relabel:
 140: (8)                   G = relabel_gexf_graph(reader(path))
 141: (4)               else:
 142: (8)                   G = reader(path)
 143: (4)               return G
 144: (0)           class GEXF:
 145: (4)               versions = {
 146: (8)                   "1.1draft": {
 147: (12)                      "NS_GEXF": "http://www.gexf.net/1.1draft",
 148: (12)                      "NS_VIZ": "http://www.gexf.net/1.1draft/viz",
 149: (12)                      "NS_XSI": "http://www.w3.org/2001/XMLSchema-instance",
 150: (12)                      "SCHEMALOCATION": " ".join(
 151: (16)                          [
```

```
(20)                                "http://www.gexf.net/1.1draft",
(20)                                "http://www.gexf.net/1.1draft/gexf.xsd",
(16)                            ]
(12)                        ),
(12)                        "VERSION": "1.1",
(8)                     },
(8)                     "1.2draft": {
(12)                        "NS_GEXF": "http://www.gexf.net/1.2draft",
(12)                        "NS_VIZ": "http://www.gexf.net/1.2draft/viz",
(12)                        "NS_XSI": "http://www.w3.org/2001/XMLSchema-instance",
(12)                        "SCHEMALOCATION": " ".join(
(16)                            [
(20)                                "http://www.gexf.net/1.2draft",
(20)                                "http://www.gexf.net/1.2draft/gexf.xsd",
(16)                            ]
(12)                        ),
(12)                        "VERSION": "1.2",
(8)                     },
(4)                 }
(4)                 def construct_types(self):
(8)                     types = [
(12)                        (int, "integer"),
(12)                        (float, "float"),
(12)                        (float, "double"),
(12)                        (bool, "boolean"),
(12)                        (list, "string"),
(12)                        (dict, "string"),
(12)                        (int, "long"),
(12)                        (str, "liststring"),
(12)                        (str, "anyURI"),
(12)                        (str, "string"),
(8)                     ]
(8)                     try:
(12)                        import numpy as np
(8)                     except ImportError:
(12)                        pass
(8)                     else:
(12)                        types = [
(16)                            (np.float64, "float"),
(16)                            (np.float32, "float"),
(16)                            (np.float16, "float"),
(16)                            (np.int_, "int"),
(16)                            (np.int8, "int"),
(16)                            (np.int16, "int"),
(16)                            (np.int32, "int"),
(16)                            (np.int64, "int"),
(16)                            (np.uint8, "int"),
(16)                            (np.uint16, "int"),
(16)                            (np.uint32, "int"),
(16)                            (np.uint64, "int"),
(16)                            (np.int_, "int"),
(16)                            (np.intc, "int"),
(16)                            (np.intp, "int"),
(12)                        ] + types
(8)                     self.xml_type = dict(types)
(8)                     self.python_type = dict(reversed(a) for a in types)
(4)                 convert_bool = {
(8)                     "true": True,
(8)                     "false": False,
(8)                     "True": True,
(8)                     "False": False,
(8)                     "0": False,
(8)                     0: False,
(8)                     "1": True,
(8)                     1: True,
(4)                 }
(4)                 def set_version(self, version):
(8)                     d = self.versions.get(version)
(8)                     if d is None:
```

```
221: (12)                        raise nx.NetworkXError(f"Unknown GEXF version {version}.")
222: (8)                    self.NS_GEXF = d["NS_GEXF"]
223: (8)                    self.NS_VIZ = d["NS_VIZ"]
224: (8)                    self.NS_XSI = d["NS_XSI"]
225: (8)                    self.SCHEMALOCATION = d["SCHEMALOCATION"]
226: (8)                    self.VERSION = d["VERSION"]
227: (8)                    self.version = version
228: (0)            class GEXFWriter(GEXF):
229: (4)                def __init__(
230: (8)                    self, graph=None, encoding="utf-8", prettyprint=True,
version="1.2draft"
231: (4)                ):
232: (8)                    self.construct_types()
233: (8)                    self.prettyprint = prettyprint
234: (8)                    self.encoding = encoding
235: (8)                    self.set_version(version)
236: (8)                    self.xml = Element(
237: (12)                       "gexf",
238: (12)                       {
239: (16)                           "xmlns": self.NS_GEXF,
240: (16)                           "xmlns:xsi": self.NS_XSI,
241: (16)                           "xsi:schemaLocation": self.SCHEMALOCATION,
242: (16)                           "version": self.VERSION,
243: (12)                       },
244: (8)                    )
245: (8)                    meta_element = Element("meta")
246: (8)                    subelement_text = f"NetworkX {nx.__version__}"
247: (8)                    SubElement(meta_element, "creator").text = subelement_text
248: (8)                    meta_element.set("lastmodifieddate", time.strftime("%Y-%m-%d"))
249: (8)                    self.xml.append(meta_element)
250: (8)                    register_namespace("viz", self.NS_VIZ)
251: (8)                    self.edge_id = itertools.count()
252: (8)                    self.attr_id = itertools.count()
253: (8)                    self.all_edge_ids = set()
254: (8)                    self.attr = {}
255: (8)                    self.attr["node"] = {}
256: (8)                    self.attr["edge"] = {}
257: (8)                    self.attr["node"]["dynamic"] = {}
258: (8)                    self.attr["node"]["static"] = {}
259: (8)                    self.attr["edge"]["dynamic"] = {}
260: (8)                    self.attr["edge"]["static"] = {}
261: (8)                    if graph is not None:
262: (12)                       self.add_graph(graph)
263: (4)                def __str__(self):
264: (8)                    if self.prettyprint:
265: (12)                       self.indent(self.xml)
266: (8)                    s = tostring(self.xml).decode(self.encoding)
267: (8)                    return s
268: (4)                def add_graph(self, G):
269: (8)                    for u, v, dd in G.edges(data=True):
270: (12)                       eid = dd.get("id")
271: (12)                       if eid is not None:
272: (16)                           self.all_edge_ids.add(str(eid))
273: (8)                    if G.graph.get("mode") == "dynamic":
274: (12)                       mode = "dynamic"
275: (8)                    else:
276: (12)                       mode = "static"
277: (8)                    if G.is_directed():
278: (12)                       default = "directed"
279: (8)                    else:
280: (12)                       default = "undirected"
281: (8)                    name = G.graph.get("name", "")
282: (8)                    graph_element = Element("graph", defaultedgetype=default, mode=mode,
name=name)
283: (8)                    self.graph_element = graph_element
284: (8)                    self.add_nodes(G, graph_element)
285: (8)                    self.add_edges(G, graph_element)
286: (8)                    self.xml.append(graph_element)
287: (4)                def add_nodes(self, G, graph_element):
```

```
288: (8)                    nodes_element = Element("nodes")
289: (8)                    for node, data in G.nodes(data=True):
290: (12)                       node_data = data.copy()
291: (12)                       node_id = str(node_data.pop("id", node))
292: (12)                       kw = {"id": node_id}
293: (12)                       label = str(node_data.pop("label", node))
294: (12)                       kw["label"] = label
295: (12)                       try:
296: (16)                           pid = node_data.pop("pid")
297: (16)                           kw["pid"] = str(pid)
298: (12)                       except KeyError:
299: (16)                           pass
300: (12)                       try:
301: (16)                           start = node_data.pop("start")
302: (16)                           kw["start"] = str(start)
303: (16)                           self.alter_graph_mode_timeformat(start)
304: (12)                       except KeyError:
305: (16)                           pass
306: (12)                       try:
307: (16)                           end = node_data.pop("end")
308: (16)                           kw["end"] = str(end)
309: (16)                           self.alter_graph_mode_timeformat(end)
310: (12)                       except KeyError:
311: (16)                           pass
312: (12)                       node_element = Element("node", **kw)
313: (12)                       default = G.graph.get("node_default", {})
314: (12)                       node_data = self.add_parents(node_element, node_data)
315: (12)                       if self.VERSION == "1.1":
316: (16)                           node_data = self.add_slices(node_element, node_data)
317: (12)                       else:
318: (16)                           node_data = self.add_spells(node_element, node_data)
319: (12)                       node_data = self.add_viz(node_element, node_data)
320: (12)                       node_data = self.add_attributes("node", node_element, node_data,
default)
321: (12)                       nodes_element.append(node_element)
322: (8)                    graph_element.append(nodes_element)
323: (4)                def add_edges(self, G, graph_element):
324: (8)                    def edge_key_data(G):
325: (12)                       if G.is_multigraph():
326: (16)                           for u, v, key, data in G.edges(data=True, keys=True):
327: (20)                               edge_data = data.copy()
328: (20)                               edge_data.update(key=key)
329: (20)                               edge_id = edge_data.pop("id", None)
330: (20)                               if edge_id is None:
331: (24)                                   edge_id = next(self.edge_id)
332: (24)                                   while str(edge_id) in self.all_edge_ids:
333: (28)                                       edge_id = next(self.edge_id)
334: (24)                                   self.all_edge_ids.add(str(edge_id))
335: (20)                               yield u, v, edge_id, edge_data
336: (12)                       else:
337: (16)                           for u, v, data in G.edges(data=True):
338: (20)                               edge_data = data.copy()
339: (20)                               edge_id = edge_data.pop("id", None)
340: (20)                               if edge_id is None:
341: (24)                                   edge_id = next(self.edge_id)
342: (24)                                   while str(edge_id) in self.all_edge_ids:
343: (28)                                       edge_id = next(self.edge_id)
344: (24)                                   self.all_edge_ids.add(str(edge_id))
345: (20)                               yield u, v, edge_id, edge_data
346: (8)                    edges_element = Element("edges")
347: (8)                    for u, v, key, edge_data in edge_key_data(G):
348: (12)                       kw = {"id": str(key)}
349: (12)                       try:
350: (16)                           edge_label = edge_data.pop("label")
351: (16)                           kw["label"] = str(edge_label)
352: (12)                       except KeyError:
353: (16)                           pass
354: (12)                       try:
355: (16)                           edge_weight = edge_data.pop("weight")
```

```
356: (16)                            kw["weight"] = str(edge_weight)
357: (12)                        except KeyError:
358: (16)                            pass
359: (12)                        try:
360: (16)                            edge_type = edge_data.pop("type")
361: (16)                            kw["type"] = str(edge_type)
362: (12)                        except KeyError:
363: (16)                            pass
364: (12)                        try:
365: (16)                            start = edge_data.pop("start")
366: (16)                            kw["start"] = str(start)
367: (16)                            self.alter_graph_mode_timeformat(start)
368: (12)                        except KeyError:
369: (16)                            pass
370: (12)                        try:
371: (16)                            end = edge_data.pop("end")
372: (16)                            kw["end"] = str(end)
373: (16)                            self.alter_graph_mode_timeformat(end)
374: (12)                        except KeyError:
375: (16)                            pass
376: (12)                        source_id = str(G.nodes[u].get("id", u))
377: (12)                        target_id = str(G.nodes[v].get("id", v))
378: (12)                        edge_element = Element("edge", source=source_id, target=target_id,
**kw)
379: (12)                        default = G.graph.get("edge_default", {})
380: (12)                        if self.VERSION == "1.1":
381: (16)                            edge_data = self.add_slices(edge_element, edge_data)
382: (12)                        else:
383: (16)                            edge_data = self.add_spells(edge_element, edge_data)
384: (12)                        edge_data = self.add_viz(edge_element, edge_data)
385: (12)                        edge_data = self.add_attributes("edge", edge_element, edge_data,
default)
386: (12)                        edges_element.append(edge_element)
387: (8)                    graph_element.append(edges_element)
388: (4)                def add_attributes(self, node_or_edge, xml_obj, data, default):
389: (8)                    attvalues = Element("attvalues")
390: (8)                    if len(data) == 0:
391: (12)                        return data
392: (8)                    mode = "static"
393: (8)                    for k, v in data.items():
394: (12)                        if k == "key":
395: (16)                            k = "networkx_key"
396: (12)                        val_type = type(v)
397: (12)                        if val_type not in self.xml_type:
398: (16)                            raise TypeError(f"attribute value type is not allowed:
{val_type}")
399: (12)                        if isinstance(v, list):
400: (16)                            for val, start, end in v:
401: (20)                                val_type = type(val)
402: (20)                                if start is not None or end is not None:
403: (24)                                    mode = "dynamic"
404: (24)                                    self.alter_graph_mode_timeformat(start)
405: (24)                                    self.alter_graph_mode_timeformat(end)
406: (24)                                    break
407: (16)                            attr_id = self.get_attr_id(
408: (20)                                str(k), self.xml_type[val_type], node_or_edge, default,
mode
409: (16)                            )
410: (16)                            for val, start, end in v:
411: (20)                                e = Element("attvalue")
412: (20)                                e.attrib["for"] = attr_id
413: (20)                                e.attrib["value"] = str(val)
414: (20)                                if val_type == float:
415: (24)                                    if e.attrib["value"] == "inf":
416: (28)                                        e.attrib["value"] = "INF"
417: (24)                                    elif e.attrib["value"] == "nan":
418: (28)                                        e.attrib["value"] = "NaN"
419: (24)                                    elif e.attrib["value"] == "-inf":
420: (28)                                        e.attrib["value"] = "-INF"
```

```
(20)                                if start is not None:
(24)                                    e.attrib["start"] = str(start)
(20)                                if end is not None:
(24)                                    e.attrib["end"] = str(end)
(20)                                attvalues.append(e)
(12)                            else:
(16)                                mode = "static"
(16)                                attr_id = self.get_attr_id(
(20)                                    str(k), self.xml_type[val_type], node_or_edge, default,
mode
(16)                                )
(16)                                e = Element("attvalue")
(16)                                e.attrib["for"] = attr_id
(16)                                if isinstance(v, bool):
(20)                                    e.attrib["value"] = str(v).lower()
(16)                                else:
(20)                                    e.attrib["value"] = str(v)
(20)                                    if val_type == float:
(24)                                        if e.attrib["value"] == "inf":
(28)                                            e.attrib["value"] = "INF"
(24)                                        elif e.attrib["value"] == "nan":
(28)                                            e.attrib["value"] = "NaN"
(24)                                        elif e.attrib["value"] == "-inf":
(28)                                            e.attrib["value"] = "-INF"
(16)                                attvalues.append(e)
(8)                         xml_obj.append(attvalues)
(8)                         return data
(4)                     def get_attr_id(self, title, attr_type, edge_or_node, default, mode):
(8)                         try:
(12)                            return self.attr[edge_or_node][mode][title]
(8)                         except KeyError:
(12)                            new_id = str(next(self.attr_id))
(12)                            self.attr[edge_or_node][mode][title] = new_id
(12)                            attr_kwargs = {"id": new_id, "title": title, "type": attr_type}
(12)                            attribute = Element("attribute", **attr_kwargs)
(12)                            default_title = default.get(title)
(12)                            if default_title is not None:
(16)                                default_element = Element("default")
(16)                                default_element.text = str(default_title)
(16)                                attribute.append(default_element)
(12)                            attributes_element = None
(12)                            for a in self.graph_element.findall("attributes"):
(16)                                a_class = a.get("class")
(16)                                a_mode = a.get("mode", "static")
(16)                                if a_class == edge_or_node and a_mode == mode:
(20)                                    attributes_element = a
(12)                            if attributes_element is None:
(16)                                attr_kwargs = {"mode": mode, "class": edge_or_node}
(16)                                attributes_element = Element("attributes", **attr_kwargs)
(16)                                self.graph_element.insert(0, attributes_element)
(12)                            attributes_element.append(attribute)
(8)                         return new_id
(4)                     def add_viz(self, element, node_data):
(8)                         viz = node_data.pop("viz", False)
(8)                         if viz:
(12)                            color = viz.get("color")
(12)                            if color is not None:
(16)                                if self.VERSION == "1.1":
(20)                                    e = Element(
(24)                                        f"{{{self.NS_VIZ}}}color",
(24)                                        r=str(color.get("r")),
(24)                                        g=str(color.get("g")),
(24)                                        b=str(color.get("b")),
(20)                                    )
(16)                                else:
(20)                                    e = Element(
(24)                                        f"{{{self.NS_VIZ}}}color",
(24)                                        r=str(color.get("r")),
(24)                                        g=str(color.get("g")),
```

```
489: (24)                                   b=str(color.get("b")),
490: (24)                                   a=str(color.get("a", 1.0)),
491: (20)                               )
492: (16)                           element.append(e)
493: (12)                       size = viz.get("size")
494: (12)                       if size is not None:
495: (16)                           e = Element(f"{{{self.NS_VIZ}}}size", value=str(size))
496: (16)                           element.append(e)
497: (12)                       thickness = viz.get("thickness")
498: (12)                       if thickness is not None:
499: (16)                           e = Element(f"{{{self.NS_VIZ}}}thickness",
value=str(thickness))
500: (16)                           element.append(e)
501: (12)                       shape = viz.get("shape")
502: (12)                       if shape is not None:
503: (16)                           if shape.startswith("http"):
504: (20)                               e = Element(
505: (24)                                   f"{{{self.NS_VIZ}}}shape", value="image",
uri=str(shape)
506: (20)                               )
507: (16)                           else:
508: (20)                               e = Element(f"{{{self.NS_VIZ}}}shape", value=str(shape))
509: (16)                           element.append(e)
510: (12)                       position = viz.get("position")
511: (12)                       if position is not None:
512: (16)                           e = Element(
513: (20)                               f"{{{self.NS_VIZ}}}position",
514: (20)                               x=str(position.get("x")),
515: (20)                               y=str(position.get("y")),
516: (20)                               z=str(position.get("z")),
517: (16)                           )
518: (16)                           element.append(e)
519: (8)                    return node_data
520: (4)                def add_parents(self, node_element, node_data):
521: (8)                    parents = node_data.pop("parents", False)
522: (8)                    if parents:
523: (12)                       parents_element = Element("parents")
524: (12)                       for p in parents:
525: (16)                           e = Element("parent")
526: (16)                           e.attrib["for"] = str(p)
527: (16)                           parents_element.append(e)
528: (12)                       node_element.append(parents_element)
529: (8)                    return node_data
530: (4)                def add_slices(self, node_or_edge_element, node_or_edge_data):
531: (8)                    slices = node_or_edge_data.pop("slices", False)
532: (8)                    if slices:
533: (12)                       slices_element = Element("slices")
534: (12)                       for start, end in slices:
535: (16)                           e = Element("slice", start=str(start), end=str(end))
536: (16)                           slices_element.append(e)
537: (12)                       node_or_edge_element.append(slices_element)
538: (8)                    return node_or_edge_data
539: (4)                def add_spells(self, node_or_edge_element, node_or_edge_data):
540: (8)                    spells = node_or_edge_data.pop("spells", False)
541: (8)                    if spells:
542: (12)                       spells_element = Element("spells")
543: (12)                       for start, end in spells:
544: (16)                           e = Element("spell")
545: (16)                           if start is not None:
546: (20)                               e.attrib["start"] = str(start)
547: (20)                               self.alter_graph_mode_timeformat(start)
548: (16)                           if end is not None:
549: (20)                               e.attrib["end"] = str(end)
550: (20)                               self.alter_graph_mode_timeformat(end)
551: (16)                           spells_element.append(e)
552: (12)                       node_or_edge_element.append(spells_element)
553: (8)                    return node_or_edge_data
554: (4)                def alter_graph_mode_timeformat(self, start_or_end):
555: (8)                    if self.graph_element.get("mode") == "static":
```

```
556: (12)                        if start_or_end is not None:
557: (16)                            if isinstance(start_or_end, str):
558: (20)                                timeformat = "date"
559: (16)                            elif isinstance(start_or_end, float):
560: (20)                                timeformat = "double"
561: (16)                            elif isinstance(start_or_end, int):
562: (20)                                timeformat = "long"
563: (16)                            else:
564: (20)                                raise nx.NetworkXError(
565: (24)                                    "timeformat should be of the type int, float or str"
566: (20)                                )
567: (16)                            self.graph_element.set("timeformat", timeformat)
568: (16)                            self.graph_element.set("mode", "dynamic")
569: (4)                def write(self, fh):
570: (8)                    if self.prettyprint:
571: (12)                        self.indent(self.xml)
572: (8)                    document = ElementTree(self.xml)
573: (8)                    document.write(fh, encoding=self.encoding, xml_declaration=True)
574: (4)                def indent(self, elem, level=0):
575: (8)                    i = "\n" + "  " * level
576: (8)                    if len(elem):
577: (12)                        if not elem.text or not elem.text.strip():
578: (16)                            elem.text = i + "  "
579: (12)                        if not elem.tail or not elem.tail.strip():
580: (16)                            elem.tail = i
581: (12)                        for elem in elem:
582: (16)                            self.indent(elem, level + 1)
583: (12)                        if not elem.tail or not elem.tail.strip():
584: (16)                            elem.tail = i
585: (8)                    else:
586: (12)                        if level and (not elem.tail or not elem.tail.strip()):
587: (16)                            elem.tail = i
588: (0)            class GEXFReader(GEXF):
589: (4)                def __init__(self, node_type=None, version="1.2draft"):
590: (8)                    self.construct_types()
591: (8)                    self.node_type = node_type
592: (8)                    self.simple_graph = True
593: (8)                    self.set_version(version)
594: (4)                def __call__(self, stream):
595: (8)                    self.xml = ElementTree(file=stream)
596: (8)                    g = self.xml.find(f"{{{self.NS_GEXF}}}graph")
597: (8)                    if g is not None:
598: (12)                       return self.make_graph(g)
599: (8)                    for version in self.versions:
600: (12)                       self.set_version(version)
601: (12)                       g = self.xml.find(f"{{{self.NS_GEXF}}}graph")
602: (12)                       if g is not None:
603: (16)                           return self.make_graph(g)
604: (8)                    raise nx.NetworkXError("No <graph> element in GEXF file.")
605: (4)                def make_graph(self, graph_xml):
606: (8)                    edgedefault = graph_xml.get("defaultedgetype", None)
607: (8)                    if edgedefault == "directed":
608: (12)                       G = nx.MultiDiGraph()
609: (8)                    else:
610: (12)                       G = nx.MultiGraph()
611: (8)                    graph_name = graph_xml.get("name", "")
612: (8)                    if graph_name != "":
613: (12)                       G.graph["name"] = graph_name
614: (8)                    graph_start = graph_xml.get("start")
615: (8)                    if graph_start is not None:
616: (12)                       G.graph["start"] = graph_start
617: (8)                    graph_end = graph_xml.get("end")
618: (8)                    if graph_end is not None:
619: (12)                       G.graph["end"] = graph_end
620: (8)                    graph_mode = graph_xml.get("mode", "")
621: (8)                    if graph_mode == "dynamic":
622: (12)                       G.graph["mode"] = "dynamic"
623: (8)                    else:
624: (12)                       G.graph["mode"] = "static"
```

```
625: (8)                    self.timeformat = graph_xml.get("timeformat")
626: (8)                    if self.timeformat == "date":
627: (12)                       self.timeformat = "string"
628: (8)                    attributes_elements = graph_xml.findall(f"
{{{self.NS_GEXF}}}attributes")
629: (8)                    node_attr = {}
630: (8)                    node_default = {}
631: (8)                    edge_attr = {}
632: (8)                    edge_default = {}
633: (8)                    for a in attributes_elements:
634: (12)                       attr_class = a.get("class")
635: (12)                       if attr_class == "node":
636: (16)                           na, nd = self.find_gexf_attributes(a)
637: (16)                           node_attr.update(na)
638: (16)                           node_default.update(nd)
639: (16)                           G.graph["node_default"] = node_default
640: (12)                       elif attr_class == "edge":
641: (16)                           ea, ed = self.find_gexf_attributes(a)
642: (16)                           edge_attr.update(ea)
643: (16)                           edge_default.update(ed)
644: (16)                           G.graph["edge_default"] = edge_default
645: (12)                       else:
646: (16)                           raise  # unknown attribute class
647: (8)                    ea = {"weight": {"type": "double", "mode": "static", "title":
"weight"}}
648: (8)                    ed = {}
649: (8)                    edge_attr.update(ea)
650: (8)                    edge_default.update(ed)
651: (8)                    G.graph["edge_default"] = edge_default
652: (8)                    nodes_element = graph_xml.find(f"{{{self.NS_GEXF}}}nodes")
653: (8)                    if nodes_element is not None:
654: (12)                       for node_xml in nodes_element.findall(f"{{{self.NS_GEXF}}}node"):
655: (16)                           self.add_node(G, node_xml, node_attr)
656: (8)                    edges_element = graph_xml.find(f"{{{self.NS_GEXF}}}edges")
657: (8)                    if edges_element is not None:
658: (12)                       for edge_xml in edges_element.findall(f"{{{self.NS_GEXF}}}edge"):
659: (16)                           self.add_edge(G, edge_xml, edge_attr)
660: (8)                    if self.simple_graph:
661: (12)                       if G.is_directed():
662: (16)                           G = nx.DiGraph(G)
663: (12)                       else:
664: (16)                           G = nx.Graph(G)
665: (8)                    return G
666: (4)                def add_node(self, G, node_xml, node_attr, node_pid=None):
667: (8)                    data = self.decode_attr_elements(node_attr, node_xml)
668: (8)                    data = self.add_parents(data, node_xml)  # add any parents
669: (8)                    if self.VERSION == "1.1":
670: (12)                       data = self.add_slices(data, node_xml)  # add slices
671: (8)                    else:
672: (12)                       data = self.add_spells(data, node_xml)  # add spells
673: (8)                    data = self.add_viz(data, node_xml)  # add viz
674: (8)                    data = self.add_start_end(data, node_xml)  # add start/end
675: (8)                    node_id = node_xml.get("id")
676: (8)                    if self.node_type is not None:
677: (12)                       node_id = self.node_type(node_id)
678: (8)                    node_label = node_xml.get("label")
679: (8)                    data["label"] = node_label
680: (8)                    node_pid = node_xml.get("pid", node_pid)
681: (8)                    if node_pid is not None:
682: (12)                       data["pid"] = node_pid
683: (8)                    subnodes = node_xml.find(f"{{{self.NS_GEXF}}}nodes")
684: (8)                    if subnodes is not None:
685: (12)                       for node_xml in subnodes.findall(f"{{{self.NS_GEXF}}}node"):
686: (16)                           self.add_node(G, node_xml, node_attr, node_pid=node_id)
687: (8)                    G.add_node(node_id, **data)
688: (4)                def add_start_end(self, data, xml):
689: (8)                    ttype = self.timeformat
690: (8)                    node_start = xml.get("start")
691: (8)                    if node_start is not None:
```

```
(12)                        data["start"] = self.python_type[ttype](node_start)
(8)                     node_end = xml.get("end")
(8)                     if node_end is not None:
(12)                        data["end"] = self.python_type[ttype](node_end)
(8)                     return data
(4)                 def add_viz(self, data, node_xml):
(8)                     viz = {}
(8)                     color = node_xml.find(f"{{{self.NS_VIZ}}}color")
(8)                     if color is not None:
(12)                        if self.VERSION == "1.1":
(16)                            viz["color"] = {
(20)                                "r": int(color.get("r")),
(20)                                "g": int(color.get("g")),
(20)                                "b": int(color.get("b")),
(16)                            }
(12)                        else:
(16)                            viz["color"] = {
(20)                                "r": int(color.get("r")),
(20)                                "g": int(color.get("g")),
(20)                                "b": int(color.get("b")),
(20)                                "a": float(color.get("a", 1)),
(16)                            }
(8)                     size = node_xml.find(f"{{{self.NS_VIZ}}}size")
(8)                     if size is not None:
(12)                        viz["size"] = float(size.get("value"))
(8)                     thickness = node_xml.find(f"{{{self.NS_VIZ}}}thickness")
(8)                     if thickness is not None:
(12)                        viz["thickness"] = float(thickness.get("value"))
(8)                     shape = node_xml.find(f"{{{self.NS_VIZ}}}shape")
(8)                     if shape is not None:
(12)                        viz["shape"] = shape.get("shape")
(12)                        if viz["shape"] == "image":
(16)                            viz["shape"] = shape.get("uri")
(8)                     position = node_xml.find(f"{{{self.NS_VIZ}}}position")
(8)                     if position is not None:
(12)                        viz["position"] = {
(16)                            "x": float(position.get("x", 0)),
(16)                            "y": float(position.get("y", 0)),
(16)                            "z": float(position.get("z", 0)),
(12)                        }
(8)                     if len(viz) > 0:
(12)                        data["viz"] = viz
(8)                     return data
(4)                 def add_parents(self, data, node_xml):
(8)                     parents_element = node_xml.find(f"{{{self.NS_GEXF}}}parents")
(8)                     if parents_element is not None:
(12)                        data["parents"] = []
(12)                        for p in parents_element.findall(f"{{{self.NS_GEXF}}}parent"):
(16)                            parent = p.get("for")
(16)                            data["parents"].append(parent)
(8)                     return data
(4)                 def add_slices(self, data, node_or_edge_xml):
(8)                     slices_element = node_or_edge_xml.find(f"{{{self.NS_GEXF}}}slices")
(8)                     if slices_element is not None:
(12)                        data["slices"] = []
(12)                        for s in slices_element.findall(f"{{{self.NS_GEXF}}}slice"):
(16)                            start = s.get("start")
(16)                            end = s.get("end")
(16)                            data["slices"].append((start, end))
(8)                     return data
(4)                 def add_spells(self, data, node_or_edge_xml):
(8)                     spells_element = node_or_edge_xml.find(f"{{{self.NS_GEXF}}}spells")
(8)                     if spells_element is not None:
(12)                        data["spells"] = []
(12)                        ttype = self.timeformat
(12)                        for s in spells_element.findall(f"{{{self.NS_GEXF}}}spell"):
(16)                            start = self.python_type[ttype](s.get("start"))
(16)                            end = self.python_type[ttype](s.get("end"))
(16)                            data["spells"].append((start, end))
```

```
761: (8)                    return data
762: (4)                def add_edge(self, G, edge_element, edge_attr):
763: (8)                    edge_direction = edge_element.get("type")
764: (8)                    if G.is_directed() and edge_direction == "undirected":
765: (12)                       raise nx.NetworkXError("Undirected edge found in directed graph.")
766: (8)                    if (not G.is_directed()) and edge_direction == "directed":
767: (12)                       raise nx.NetworkXError("Directed edge found in undirected graph.")
768: (8)                    source = edge_element.get("source")
769: (8)                    target = edge_element.get("target")
770: (8)                    if self.node_type is not None:
771: (12)                       source = self.node_type(source)
772: (12)                       target = self.node_type(target)
773: (8)                    data = self.decode_attr_elements(edge_attr, edge_element)
774: (8)                    data = self.add_start_end(data, edge_element)
775: (8)                    if self.VERSION == "1.1":
776: (12)                       data = self.add_slices(data, edge_element)  # add slices
777: (8)                    else:
778: (12)                       data = self.add_spells(data, edge_element)  # add spells
779: (8)                    edge_id = edge_element.get("id")
780: (8)                    if edge_id is not None:
781: (12)                       data["id"] = edge_id
782: (8)                    multigraph_key = data.pop("networkx_key", None)
783: (8)                    if multigraph_key is not None:
784: (12)                       edge_id = multigraph_key
785: (8)                    weight = edge_element.get("weight")
786: (8)                    if weight is not None:
787: (12)                       data["weight"] = float(weight)
788: (8)                    edge_label = edge_element.get("label")
789: (8)                    if edge_label is not None:
790: (12)                       data["label"] = edge_label
791: (8)                    if G.has_edge(source, target):
792: (12)                       self.simple_graph = False
793: (8)                    G.add_edge(source, target, key=edge_id, **data)
794: (8)                    if edge_direction == "mutual":
795: (12)                       G.add_edge(target, source, key=edge_id, **data)
796: (4)                def decode_attr_elements(self, gexf_keys, obj_xml):
797: (8)                    attr = {}
798: (8)                    attr_element = obj_xml.find(f"{{{self.NS_GEXF}}}attvalues")
799: (8)                    if attr_element is not None:
800: (12)                       for a in attr_element.findall(f"{{{self.NS_GEXF}}}attvalue"):
801: (16)                           key = a.get("for")  # for is required
802: (16)                           try:  # should be in our gexf_keys dictionary
803: (20)                               title = gexf_keys[key]["title"]
804: (16)                           except KeyError as err:
805: (20)                               raise nx.NetworkXError(f"No attribute defined for={key}.")
from err
806: (16)                           atype = gexf_keys[key]["type"]
807: (16)                           value = a.get("value")
808: (16)                           if atype == "boolean":
809: (20)                               value = self.convert_bool[value]
810: (16)                           else:
811: (20)                               value = self.python_type[atype](value)
812: (16)                           if gexf_keys[key]["mode"] == "dynamic":
813: (20)                               ttype = self.timeformat
814: (20)                               start = self.python_type[ttype](a.get("start"))
815: (20)                               end = self.python_type[ttype](a.get("end"))
816: (20)                               if title in attr:
817: (24)                                   attr[title].append((value, start, end))
818: (20)                               else:
819: (24)                                   attr[title] = [(value, start, end)]
820: (16)                           else:
821: (20)                               attr[title] = value
822: (8)                    return attr
823: (4)                def find_gexf_attributes(self, attributes_element):
824: (8)                    attrs = {}
825: (8)                    defaults = {}
826: (8)                    mode = attributes_element.get("mode")
827: (8)                    for k in attributes_element.findall(f"{{{self.NS_GEXF}}}attribute"):
828: (12)                       attr_id = k.get("id")
```

```
829: (12)                        title = k.get("title")
830: (12)                        atype = k.get("type")
831: (12)                        attrs[attr_id] = {"title": title, "type": atype, "mode": mode}
832: (12)                        default = k.find(f"{{{self.NS_GEXF}}}default")
833: (12)                        if default is not None:
834: (16)                            if atype == "boolean":
835: (20)                                value = self.convert_bool[default.text]
836: (16)                            else:
837: (20)                                value = self.python_type[atype](default.text)
838: (16)                            defaults[title] = value
839: (8)                    return attrs, defaults
840: (0)            def relabel_gexf_graph(G):
841: (4)                """Relabel graph using "label" node keyword for node label.
842: (4)                Parameters
843: (4)                ----------
844: (4)                G : graph
845: (7)                   A NetworkX graph read from GEXF data
846: (4)                Returns
847: (4)                -------
848: (4)                H : graph
849: (6)                  A NetworkX graph with relabeled nodes
850: (4)                Raises
851: (4)                ------
852: (4)                NetworkXError
853: (8)                    If node labels are missing or not unique while relabel=True.
854: (4)                Notes
855: (4)                -----
856: (4)                This function relabels the nodes in a NetworkX graph with the
857: (4)                "label" attribute.  It also handles relabeling the specific GEXF
858: (4)                node attributes "parents", and "pid".
859: (4)                """
860: (4)                try:
861: (8)                    mapping = [(u, G.nodes[u]["label"]) for u in G]
862: (4)                except KeyError as err:
863: (8)                    raise nx.NetworkXError(
864: (12)                       "Failed to relabel nodes: missing node labels found. Use
relabel=False."
865: (8)                    ) from err
866: (4)                x, y = zip(*mapping)
867: (4)                if len(set(y)) != len(G):
868: (8)                    raise nx.NetworkXError(
869: (12)                       "Failed to relabel nodes: "
870: (12)                       "duplicate node labels found. "
871: (12)                       "Use relabel=False."
872: (8)                    )
873: (4)                mapping = dict(mapping)
874: (4)                H = nx.relabel_nodes(G, mapping)
875: (4)                for n in G:
876: (8)                    m = mapping[n]
877: (8)                    H.nodes[m]["id"] = n
878: (8)                    H.nodes[m].pop("label")
879: (8)                    if "pid" in H.nodes[m]:
880: (12)                       H.nodes[m]["pid"] = mapping[G.nodes[n]["pid"]]
881: (8)                    if "parents" in H.nodes[m]:
882: (12)                       H.nodes[m]["parents"] = [mapping[p] for p in G.nodes[n]
["parents"]]
883: (4)                return H
```

File 505 - gml.py:

```
1: (0)              """
2: (0)              Read graphs in GML format.
3: (0)              "GML, the Graph Modelling Language, is our proposal for a portable
4: (0)              file format for graphs. GML's key features are portability, simple
5: (0)              syntax, extensibility and flexibility. A GML file consists of a
6: (0)              hierarchical key-value lists. Graphs can be annotated with arbitrary
7: (0)              data structures. The idea for a common file format was born at the
```

```
 8: (0)              GD'95; this proposal is the outcome of many discussions. GML is the
 9: (0)              standard file format in the Graphlet graph editor system. It has been
 10: (0)             overtaken and adapted by several other systems for drawing graphs."
 11: (0)             GML files are stored using a 7-bit ASCII encoding with any extended
 12: (0)             ASCII characters (iso8859-1) appearing as HTML character entities.
 13: (0)             You will need to give some thought into how the exported data should
 14: (0)             interact with different languages and even different Python versions.
 15: (0)             Re-importing from gml is also a concern.
 16: (0)             Without specifying a `stringizer`/`destringizer`, the code is capable of
 17: (0)             writing `int`/`float`/`str`/`dict`/`list` data as required by the GML
 18: (0)             specification.  For writing other data types, and for reading data other
 19: (0)             than `str` you need to explicitly supply a `stringizer`/`destringizer`.
 20: (0)             For additional documentation on the GML file format, please see the
 21: (0)             `GML website <https://web.archive.org/web/20190207140002/http://www.fim.uni-
passau.de/index.php?id=17297&L=1>`_.
 22: (0)             Several example graphs in GML format may be found on Mark Newman's
 23: (0)             `Network data page <http://www-personal.umich.edu/~mejn/netdata/>`_.
 24: (0)             """
 25: (0)             import html.entities as htmlentitydefs
 26: (0)             import re
 27: (0)             import warnings
 28: (0)             from ast import literal_eval
 29: (0)             from collections import defaultdict
 30: (0)             from enum import Enum
 31: (0)             from io import StringIO
 32: (0)             from typing import Any, NamedTuple
 33: (0)             import networkx as nx
 34: (0)             from networkx.exception import NetworkXError
 35: (0)             from networkx.utils import open_file
 36: (0)             __all__ = ["read_gml", "parse_gml", "generate_gml", "write_gml"]
 37: (0)             def escape(text):
 38: (4)                 """Use XML character references to escape characters.
 39: (4)                 Use XML character references for unprintable or non-ASCII
 40: (4)                 characters, double quotes and ampersands in a string
 41: (4)                 """
 42: (4)                 def fixup(m):
 43: (8)                     ch = m.group(0)
 44: (8)                     return "&#" + str(ord(ch)) + ";"
 45: (4)                 text = re.sub('[^ -~]|[&"]', fixup, text)
 46: (4)                 return text if isinstance(text, str) else str(text)
 47: (0)             def unescape(text):
 48: (4)                 """Replace XML character references with the referenced characters"""
 49: (4)                 def fixup(m):
 50: (8)                     text = m.group(0)
 51: (8)                     if text[1] == "#":
 52: (12)                        if text[2] == "x":
 53: (16)                            code = int(text[3:-1], 16)
 54: (12)                        else:
 55: (16)                            code = int(text[2:-1])
 56: (8)                     else:
 57: (12)                        try:
 58: (16)                            code = htmlentitydefs.name2codepoint[text[1:-1]]
 59: (12)                        except KeyError:
 60: (16)                            return text  # leave unchanged
 61: (8)                     try:
 62: (12)                        return chr(code)
 63: (8)                     except (ValueError, OverflowError):
 64: (12)                        return text  # leave unchanged
 65: (4)                 return re.sub("&(?:[0-9A-Za-z]+|#(?:[0-9]+|x[0-9A-Fa-f]+));", fixup, text)
 66: (0)             def literal_destringizer(rep):
 67: (4)                 """Convert a Python literal to the value it represents.
 68: (4)                 Parameters
 69: (4)                 ----------
 70: (4)                 rep : string
 71: (8)                     A Python literal.
 72: (4)                 Returns
 73: (4)                 -------
 74: (4)                 value : object
 75: (8)                     The value of the Python literal.
```

```
 76: (4)                 Raises
 77: (4)                 ------
 78: (4)                 ValueError
 79: (8)                     If `rep` is not a Python literal.
 80: (4)                 """
 81: (4)                 if isinstance(rep, str):
 82: (8)                     orig_rep = rep
 83: (8)                     try:
 84: (12)                        return literal_eval(rep)
 85: (8)                     except SyntaxError as err:
 86: (12)                        raise ValueError(f"{orig_rep!r} is not a valid Python literal")
from err
 87: (4)                 else:
 88: (8)                     raise ValueError(f"{rep!r} is not a string")
 89: (0)             @open_file(0, mode="rb")
 90: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 91: (0)             def read_gml(path, label="label", destringizer=None):
 92: (4)                 """Read graph in GML format from `path`.
 93: (4)                 Parameters
 94: (4)                 ----------
 95: (4)                 path : filename or filehandle
 96: (8)                     The filename or filehandle to read from.
 97: (4)                 label : string, optional
 98: (8)                     If not None, the parsed nodes will be renamed according to node
 99: (8)                     attributes indicated by `label`. Default value: 'label'.
 100: (4)                destringizer : callable, optional
 101: (8)                    A `destringizer` that recovers values stored as strings in GML. If it
 102: (8)                    cannot convert a string to a value, a `ValueError` is raised. Default
 103: (8)                    value : None.
 104: (4)                Returns
 105: (4)                -------
 106: (4)                G : NetworkX graph
 107: (8)                    The parsed graph.
 108: (4)                Raises
 109: (4)                ------
 110: (4)                NetworkXError
 111: (8)                    If the input cannot be parsed.
 112: (4)                See Also
 113: (4)                --------
 114: (4)                write_gml, parse_gml
 115: (4)                literal_destringizer
 116: (4)                Notes
 117: (4)                -----
 118: (4)                GML files are stored using a 7-bit ASCII encoding with any extended
 119: (4)                ASCII characters (iso8859-1) appearing as HTML character entities.
 120: (4)                Without specifying a `stringizer`/`destringizer`, the code is capable of
 121: (4)                writing `int`/`float`/`str`/`dict`/`list` data as required by the GML
 122: (4)                specification.  For writing other data types, and for reading data other
 123: (4)                than `str` you need to explicitly supply a `stringizer`/`destringizer`.
 124: (4)                For additional documentation on the GML file format, please see the
 125: (4)                `GML url <https://web.archive.org/web/20190207140002/http://www.fim.uni-
passau.de/index.php?id=17297&L=1>`_.
 126: (4)                See the module docstring :mod:`networkx.readwrite.gml` for more details.
 127: (4)                Examples
 128: (4)                --------
 129: (4)                >>> G = nx.path_graph(4)
 130: (4)                >>> nx.write_gml(G, "test.gml")
 131: (4)                GML values are interpreted as strings by default:
 132: (4)                >>> H = nx.read_gml("test.gml")
 133: (4)                >>> H.nodes
 134: (4)                NodeView(('0', '1', '2', '3'))
 135: (4)                When a `destringizer` is provided, GML values are converted to the
provided type.
 136: (4)                For example, integer nodes can be recovered as shown below:
 137: (4)                >>> J = nx.read_gml("test.gml", destringizer=int)
 138: (4)                >>> J.nodes
 139: (4)                NodeView((0, 1, 2, 3))
 140: (4)                """
 141: (4)                def filter_lines(lines):
```

```
(8)                     for line in lines:
(12)                        try:
(16)                            line = line.decode("ascii")
(12)                        except UnicodeDecodeError as err:
(16)                            raise NetworkXError("input is not ASCII-encoded") from err
(12)                        if not isinstance(line, str):
(16)                            lines = str(lines)
(12)                        if line and line[-1] == "\n":
(16)                            line = line[:-1]
(12)                        yield line
(4)                 G = parse_gml_lines(filter_lines(path), label, destringizer)
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def parse_gml(lines, label="label", destringizer=None):
(4)                 """Parse GML graph from a string or iterable.
(4)                 Parameters
(4)                 ----------
(4)                 lines : string or iterable of strings
(7)                    Data in GML format.
(4)                 label : string, optional
(8)                     If not None, the parsed nodes will be renamed according to node
(8)                     attributes indicated by `label`. Default value: 'label'.
(4)                 destringizer : callable, optional
(8)                     A `destringizer` that recovers values stored as strings in GML. If it
(8)                     cannot convert a string to a value, a `ValueError` is raised. Default
(8)                     value : None.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX graph
(8)                     The parsed graph.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If the input cannot be parsed.
(4)                 See Also
(4)                 --------
(4)                 write_gml, read_gml
(4)                 Notes
(4)                 -----
(4)                 This stores nested GML attributes as dictionaries in the NetworkX graph,
(4)                 node, and edge attribute structures.
(4)                 GML files are stored using a 7-bit ASCII encoding with any extended
(4)                 ASCII characters (iso8859-1) appearing as HTML character entities.
(4)                 Without specifying a `stringizer`/`destringizer`, the code is capable of
(4)                 writing `int`/`float`/`str`/`dict`/`list` data as required by the GML
(4)                 specification.  For writing other data types, and for reading data other
(4)                 than `str` you need to explicitly supply a `stringizer`/`destringizer`.
(4)                 For additional documentation on the GML file format, please see the
(4)                 `GML url <https://web.archive.org/web/20190207140002/http://www.fim.uni-
passau.de/index.php?id=17297&L=1>`_.
(4)                 See the module docstring :mod:`networkx.readwrite.gml` for more details.
(4)                 """
(4)                 def decode_line(line):
(8)                     if isinstance(line, bytes):
(12)                        try:
(16)                            line.decode("ascii")
(12)                        except UnicodeDecodeError as err:
(16)                            raise NetworkXError("input is not ASCII-encoded") from err
(8)                     if not isinstance(line, str):
(12)                        line = str(line)
(8)                     return line
(4)                 def filter_lines(lines):
(8)                     if isinstance(lines, str):
(12)                        lines = decode_line(lines)
(12)                        lines = lines.splitlines()
(12)                        yield from lines
(8)                     else:
(12)                        for line in lines:
(16)                            line = decode_line(line)
```

```
(16)                        if line and line[-1] == "\n":
(20)                            line = line[:-1]
(16)                        if line.find("\n") != -1:
(20)                            raise NetworkXError("input line contains newline")
(16)                        yield line
(4)             G = parse_gml_lines(filter_lines(lines), label, destringizer)
(4)             return G
(0)         class Pattern(Enum):
(4)             """encodes the index of each token-matching pattern in `tokenize`."""
(4)             KEYS = 0
(4)             REALS = 1
(4)             INTS = 2
(4)             STRINGS = 3
(4)             DICT_START = 4
(4)             DICT_END = 5
(4)             COMMENT_WHITESPACE = 6
(0)         class Token(NamedTuple):
(4)             category: Pattern
(4)             value: Any
(4)             line: int
(4)             position: int
(0)         LIST_START_VALUE = "_networkx_list_start"
(0)         def parse_gml_lines(lines, label, destringizer):
(4)             """Parse GML `lines` into a graph."""
(4)             def tokenize():
(8)                 patterns = [
(12)                    r"[A-Za-z][0-9A-Za-z_]*\b",  # keys
(12)                    r"[+-]?(?:[0-9]*\.[0-9]+|[0-9]+\.[0-9]*|INF)(?:[Ee][+-]?[0-9]+)?",
(12)                    r"[+-]?[0-9]+",  # ints
(12)                    r'".*?"',  # strings
(12)                    r"\[",  # dict start
(12)                    r"\]",  # dict end
(12)                    r"#.*$|\s+",  # comments and whitespaces
(8)                 ]
(8)                 tokens = re.compile("|".join(f"({pattern})" for pattern in patterns))
(8)                 lineno = 0
(8)                 multilines = []  # entries spread across multiple lines
(8)                 for line in lines:
(12)                    pos = 0
(12)                    if multilines:
(16)                        multilines.append(line.strip())
(16)                        if line[-1] == '"':  # closing multiline entry
(20)                            line = " ".join(multilines)
(20)                            multilines = []
(16)                        else:  # continued multiline entry
(20)                            lineno += 1
(20)                            continue
(12)                    else:
(16)                        if line.count('"') == 1:  # opening multiline entry
(20)                            if line.strip()[0] != '"' and line.strip()[-1] != '"':
(24)                                multilines = [line.rstrip()]
(24)                                lineno += 1
(24)                                continue
(12)                    length = len(line)
(12)                    while pos < length:
(16)                        match = tokens.match(line, pos)
(16)                        if match is None:
(20)                            m = f"cannot tokenize {line[pos:]} at ({lineno + 1}, {pos
+ 1})"
(20)                            raise NetworkXError(m)
(16)                        for i in range(len(patterns)):
(20)                            group = match.group(i + 1)
(20)                            if group is not None:
(24)                                if i == 0:  # keys
(28)                                    value = group.rstrip()
(24)                                elif i == 1:  # reals
(28)                                    value = float(group)
(24)                                elif i == 2:  # ints
(28)                                    value = int(group)
```

```
(24)                                    else:
(28)                                        value = group
(24)                                    if i != 6:  # comments and whitespaces
(28)                                        yield Token(Pattern(i), value, lineno + 1, pos +
1)
(24)                                    pos += len(group)
(24)                                    break
(12)                        lineno += 1
(8)                     yield Token(None, None, lineno + 1, 1)  # EOF
(4)                 def unexpected(curr_token, expected):
(8)                     category, value, lineno, pos = curr_token
(8)                     value = repr(value) if value is not None else "EOF"
(8)                     raise NetworkXError(f"expected {expected}, found {value} at ({lineno},
{pos})")
(4)                 def consume(curr_token, category, expected):
(8)                     if curr_token.category == category:
(12)                        return next(tokens)
(8)                     unexpected(curr_token, expected)
(4)                 def parse_kv(curr_token):
(8)                     dct = defaultdict(list)
(8)                     while curr_token.category == Pattern.KEYS:
(12)                        key = curr_token.value
(12)                        curr_token = next(tokens)
(12)                        category = curr_token.category
(12)                        if category == Pattern.REALS or category == Pattern.INTS:
(16)                            value = curr_token.value
(16)                            curr_token = next(tokens)
(12)                        elif category == Pattern.STRINGS:
(16)                            value = unescape(curr_token.value[1:-1])
(16)                            if destringizer:
(20)                                try:
(24)                                    value = destringizer(value)
(20)                                except ValueError:
(24)                                    pass
(16)                            if value == "()":
(20)                                value = ()
(16)                            if value == "[]":
(20)                                value = []
(16)                            curr_token = next(tokens)
(12)                        elif category == Pattern.DICT_START:
(16)                            curr_token, value = parse_dict(curr_token)
(12)                        else:
(16)                            if key in ("id", "label", "source", "target"):
(20)                                try:
(24)                                    value = unescape(str(curr_token.value))
(24)                                    if destringizer:
(28)                                        try:
(32)                                            value = destringizer(value)
(28)                                        except ValueError:
(32)                                            pass
(24)                                    curr_token = next(tokens)
(20)                                except Exception:
(24)                                    msg = (
(28)                                        "an int, float, string, '[' or string"
(28)                                        + " convertible ASCII value for node id or label"
(24)                                    )
(24)                                    unexpected(curr_token, msg)
(16)                            elif curr_token.value in {"NAN", "INF"}:
(20)                                value = float(curr_token.value)
(20)                                curr_token = next(tokens)
(16)                            else:  # Otherwise error out
(20)                                unexpected(curr_token, "an int, float, string or '['")
(12)                        dct[key].append(value)
(8)                     def clean_dict_value(value):
(12)                        if not isinstance(value, list):
(16)                            return value
(12)                        if len(value) == 1:
(16)                            return value[0]
(12)                        if value[0] == LIST_START_VALUE:
```

```
345: (16)                            return value[1:]
346: (12)                        return value
347: (8)                    dct = {key: clean_dict_value(value) for key, value in dct.items()}
348: (8)                    return curr_token, dct
349: (4)                def parse_dict(curr_token):
350: (8)                    curr_token = consume(curr_token, Pattern.DICT_START, "'['")
351: (8)                    curr_token, dct = parse_kv(curr_token)
352: (8)                    curr_token = consume(curr_token, Pattern.DICT_END, "']'")
353: (8)                    return curr_token, dct
354: (4)                def parse_graph():
355: (8)                    curr_token, dct = parse_kv(next(tokens))
356: (8)                    if curr_token.category is not None:  # EOF
357: (12)                       unexpected(curr_token, "EOF")
358: (8)                    if "graph" not in dct:
359: (12)                       raise NetworkXError("input contains no graph")
360: (8)                    graph = dct["graph"]
361: (8)                    if isinstance(graph, list):
362: (12)                       raise NetworkXError("input contains more than one graph")
363: (8)                    return graph
364: (4)                tokens = tokenize()
365: (4)                graph = parse_graph()
366: (4)                directed = graph.pop("directed", False)
367: (4)                multigraph = graph.pop("multigraph", False)
368: (4)                if not multigraph:
369: (8)                    G = nx.DiGraph() if directed else nx.Graph()
370: (4)                else:
371: (8)                    G = nx.MultiDiGraph() if directed else nx.MultiGraph()
372: (4)                graph_attr = {k: v for k, v in graph.items() if k not in ("node", "edge")}
373: (4)                G.graph.update(graph_attr)
374: (4)                def pop_attr(dct, category, attr, i):
375: (8)                    try:
376: (12)                       return dct.pop(attr)
377: (8)                    except KeyError as err:
378: (12)                       raise NetworkXError(f"{category} #{i} has no {attr!r} attribute")
from err
379: (4)                nodes = graph.get("node", [])
380: (4)                mapping = {}
381: (4)                node_labels = set()
382: (4)                for i, node in enumerate(nodes if isinstance(nodes, list) else [nodes]):
383: (8)                    id = pop_attr(node, "node", "id", i)
384: (8)                    if id in G:
385: (12)                       raise NetworkXError(f"node id {id!r} is duplicated")
386: (8)                    if label is not None and label != "id":
387: (12)                       node_label = pop_attr(node, "node", label, i)
388: (12)                       if node_label in node_labels:
389: (16)                           raise NetworkXError(f"node label {node_label!r} is
duplicated")
390: (12)                       node_labels.add(node_label)
391: (12)                       mapping[id] = node_label
392: (8)                    G.add_node(id, **node)
393: (4)                edges = graph.get("edge", [])
394: (4)                for i, edge in enumerate(edges if isinstance(edges, list) else [edges]):
395: (8)                    source = pop_attr(edge, "edge", "source", i)
396: (8)                    target = pop_attr(edge, "edge", "target", i)
397: (8)                    if source not in G:
398: (12)                       raise NetworkXError(f"edge #{i} has undefined source {source!r}")
399: (8)                    if target not in G:
400: (12)                       raise NetworkXError(f"edge #{i} has undefined target {target!r}")
401: (8)                    if not multigraph:
402: (12)                       if not G.has_edge(source, target):
403: (16)                           G.add_edge(source, target, **edge)
404: (12)                       else:
405: (16)                           arrow = "->" if directed else "--"
406: (16)                           msg = f"edge #{i} ({source!r}{arrow}{target!r}) is duplicated"
407: (16)                           raise nx.NetworkXError(msg)
408: (8)                    else:
409: (12)                       key = edge.pop("key", None)
410: (12)                       if key is not None and G.has_edge(source, target, key):
411: (16)                           arrow = "->" if directed else "--"
```

```
(16)                            msg = f"edge #{i} ({source!r}{arrow}{target!r}, {key!r})"
(16)                            msg2 = 'Hint: If multigraph add "multigraph 1" to file
header.'
(16)                            raise nx.NetworkXError(msg + " is duplicated\n" + msg2)
(12)                        G.add_edge(source, target, key, **edge)
(4)                 if label is not None and label != "id":
(8)                     G = nx.relabel_nodes(G, mapping)
(4)                 return G
(0)             def literal_stringizer(value):
(4)                 """Convert a `value` to a Python literal in GML representation.
(4)                 Parameters
(4)                 ----------
(4)                 value : object
(8)                     The `value` to be converted to GML representation.
(4)                 Returns
(4)                 -------
(4)                 rep : string
(8)                     A double-quoted Python literal representing value. Unprintable
(8)                     characters are replaced by XML character references.
(4)                 Raises
(4)                 ------
(4)                 ValueError
(8)                     If `value` cannot be converted to GML.
(4)                 Notes
(4)                 -----
(4)                 The original value can be recovered using the
(4)                 :func:`networkx.readwrite.gml.literal_destringizer` function.
(4)                 """
(4)                 def stringize(value):
(8)                     if isinstance(value, int | bool) or value is None:
(12)                        if value is True:  # GML uses 1/0 for boolean values.
(16)                            buf.write(str(1))
(12)                        elif value is False:
(16)                            buf.write(str(0))
(12)                        else:
(16)                            buf.write(str(value))
(8)                     elif isinstance(value, str):
(12)                        text = repr(value)
(12)                        if text[0] != "u":
(16)                            try:
(20)                                value.encode("latin1")
(16)                            except UnicodeEncodeError:
(20)                                text = "u" + text
(12)                        buf.write(text)
(8)                     elif isinstance(value, float | complex | str | bytes):
(12)                        buf.write(repr(value))
(8)                     elif isinstance(value, list):
(12)                        buf.write("[")
(12)                        first = True
(12)                        for item in value:
(16)                            if not first:
(20)                                buf.write(",")
(16)                            else:
(20)                                first = False
(16)                            stringize(item)
(12)                        buf.write("]")
(8)                     elif isinstance(value, tuple):
(12)                        if len(value) > 1:
(16)                            buf.write("(")
(16)                            first = True
(16)                            for item in value:
(20)                                if not first:
(24)                                    buf.write(",")
(20)                                else:
(24)                                    first = False
(20)                                stringize(item)
(16)                            buf.write(")")
(12)                        elif value:
(16)                            buf.write("(")
```

```
(16)                            stringize(value[0])
(16)                            buf.write(",)")
(12)                        else:
(16)                            buf.write("()")
(8)                     elif isinstance(value, dict):
(12)                        buf.write("{")
(12)                        first = True
(12)                        for key, value in value.items():
(16)                            if not first:
(20)                                buf.write(",")
(16)                            else:
(20)                                first = False
(16)                            stringize(key)
(16)                            buf.write(":")
(16)                            stringize(value)
(12)                        buf.write("}")
(8)                     elif isinstance(value, set):
(12)                        buf.write("{")
(12)                        first = True
(12)                        for item in value:
(16)                            if not first:
(20)                                buf.write(",")
(16)                            else:
(20)                                first = False
(16)                            stringize(item)
(12)                        buf.write("}")
(8)                     else:
(12)                        msg = f"{value!r} cannot be converted into a Python literal"
(12)                        raise ValueError(msg)
(4)                 buf = StringIO()
(4)                 stringize(value)
(4)                 return buf.getvalue()
(0)             def generate_gml(G, stringizer=None):
(4)                 r"""Generate a single entry of the graph `G` in GML format.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     The graph to be converted to GML.
(4)                 stringizer : callable, optional
(8)                     A `stringizer` which converts non-int/non-float/non-dict values into
(8)                     strings. If it cannot convert a value into a string, it should raise a
(8)                     `ValueError` to indicate that. Default value: None.
(4)                 Returns
(4)                 -------
(4)                 lines: generator of strings
(8)                     Lines of GML data. Newlines are not appended.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If `stringizer` cannot convert a value into a string, or the value to
(8)                     convert is not a string while `stringizer` is None.
(4)                 See Also
(4)                 --------
(4)                 literal_stringizer
(4)                 Notes
(4)                 -----
(4)                 Graph attributes named 'directed', 'multigraph', 'node' or
(4)                 'edge', node attributes named 'id' or 'label', edge attributes
(4)                 named 'source' or 'target' (or 'key' if `G` is a multigraph)
(4)                 are ignored because these attribute names are used to encode the graph
(4)                 structure.
(4)                 GML files are stored using a 7-bit ASCII encoding with any extended
(4)                 ASCII characters (iso8859-1) appearing as HTML character entities.
(4)                 Without specifying a `stringizer`/`destringizer`, the code is capable of
(4)                 writing `int`/`float`/`str`/`dict`/`list` data as required by the GML
(4)                 specification.  For writing other data types, and for reading data other
(4)                 than `str` you need to explicitly supply a `stringizer`/`destringizer`.
(4)                 For additional documentation on the GML file format, please see the
(4)                 `GML url <https://web.archive.org/web/20190207140002/http://www.fim.uni-
```

```
passau.de/index.php?id=17297&L=1>`_.
    See the module docstring :mod:`networkx.readwrite.gml` for more details.
    Examples
    --------
    >>> G = nx.Graph()
    >>> G.add_node("1")
    >>> print("\n".join(nx.generate_gml(G)))
    graph [
      node [
        id 0
        label "1"
      ]
    ]
    >>> G = nx.MultiGraph([("a", "b"), ("a", "b")])
    >>> print("\n".join(nx.generate_gml(G)))
    graph [
      multigraph 1
      node [
        id 0
        label "a"
      ]
      node [
        id 1
        label "b"
      ]
      edge [
        source 0
        target 1
        key 0
      ]
      edge [
        source 0
        target 1
        key 1
      ]
    ]
    """
    valid_keys = re.compile("^[A-Za-z][0-9A-Za-z_]*$")
    def stringize(key, value, ignored_keys, indent, in_list=False):
        if not isinstance(key, str):
            raise NetworkXError(f"{key!r} is not a string")
        if not valid_keys.match(key):
            raise NetworkXError(f"{key!r} is not a valid key")
        if not isinstance(key, str):
            key = str(key)
        if key not in ignored_keys:
            if isinstance(value, int | bool):
                if key == "label":
                    yield indent + key + ' "' + str(value) + '"'
                elif value is True:
                    yield indent + key + " 1"
                elif value is False:
                    yield indent + key + " 0"
                elif value < -(2**31) or value >= 2**31:
                    yield indent + key + ' "' + str(value) + '"'
                else:
                    yield indent + key + " " + str(value)
            elif isinstance(value, float):
                text = repr(value).upper()
                if text == repr(float("inf")).upper():
                    text = "+" + text
                else:
                    epos = text.rfind("E")
                    if epos != -1 and text.find(".", 0, epos) == -1:
                        text = text[:epos] + "." + text[epos:]
                if key == "label":
                    yield indent + key + ' "' + text + '"'
                else:
                    yield indent + key + " " + text
```

```
(12)                        elif isinstance(value, dict):
(16)                            yield indent + key + " ["
(16)                            next_indent = indent + "  "
(16)                            for key, value in value.items():
(20)                                yield from stringize(key, value, (), next_indent)
(16)                            yield indent + "]"
(12)                        elif isinstance(value, tuple) and key == "label":
(16)                            yield indent + key + f" \"({','.join(repr(v) for v in
value)})\""
(12)                        elif isinstance(value, list | tuple) and key != "label" and not
in_list:
(16)                            if len(value) == 0:
(20)                                yield indent + key + " " + f'"{value!r}"'
(16)                            if len(value) == 1:
(20)                                yield indent + key + " " + f'"{LIST_START_VALUE}"'
(16)                            for val in value:
(20)                                yield from stringize(key, val, (), indent, True)
(12)                        else:
(16)                            if stringizer:
(20)                                try:
(24)                                    value = stringizer(value)
(20)                                except ValueError as err:
(24)                                    raise NetworkXError(
(28)                                        f"{value!r} cannot be converted into a string"
(24)                                    ) from err
(16)                            if not isinstance(value, str):
(20)                                raise NetworkXError(f"{value!r} is not a string")
(16)                            yield indent + key + ' "' + escape(value) + '"'
(4)                 multigraph = G.is_multigraph()
(4)                 yield "graph ["
(4)                 if G.is_directed():
(8)                     yield "  directed 1"
(4)                 if multigraph:
(8)                     yield "  multigraph 1"
(4)                 ignored_keys = {"directed", "multigraph", "node", "edge"}
(4)                 for attr, value in G.graph.items():
(8)                     yield from stringize(attr, value, ignored_keys, "  ")
(4)                 node_id = dict(zip(G, range(len(G))))
(4)                 ignored_keys = {"id", "label"}
(4)                 for node, attrs in G.nodes.items():
(8)                     yield "  node ["
(8)                     yield "    id " + str(node_id[node])
(8)                     yield from stringize("label", node, (), "    ")
(8)                     for attr, value in attrs.items():
(12)                        yield from stringize(attr, value, ignored_keys, "    ")
(8)                     yield "  ]"
(4)                 ignored_keys = {"source", "target"}
(4)                 kwargs = {"data": True}
(4)                 if multigraph:
(8)                     ignored_keys.add("key")
(8)                     kwargs["keys"] = True
(4)                 for e in G.edges(**kwargs):
(8)                     yield "  edge ["
(8)                     yield "    source " + str(node_id[e[0]])
(8)                     yield "    target " + str(node_id[e[1]])
(8)                     if multigraph:
(12)                        yield from stringize("key", e[2], (), "    ")
(8)                     for attr, value in e[-1].items():
(12)                        yield from stringize(attr, value, ignored_keys, "    ")
(8)                     yield "  ]"
(4)                 yield "]"
(0)             @open_file(1, mode="wb")
(0)             def write_gml(G, path, stringizer=None):
(4)                 """Write a graph `G` in GML format to the file or file handle `path`.
(4)                 Parameters
(4)                 ----------
(4)                 G : NetworkX graph
(8)                     The graph to be converted to GML.
(4)                 path : filename or filehandle
```

```
684: (8)                    The filename or filehandle to write. Files whose names end with .gz or
685: (8)                    .bz2 will be compressed.
686: (4)                stringizer : callable, optional
687: (8)                    A `stringizer` which converts non-int/non-float/non-dict values into
688: (8)                    strings. If it cannot convert a value into a string, it should raise a
689: (8)                    `ValueError` to indicate that. Default value: None.
690: (4)                Raises
691: (4)                ------
692: (4)                NetworkXError
693: (8)                    If `stringizer` cannot convert a value into a string, or the value to
694: (8)                    convert is not a string while `stringizer` is None.
695: (4)                See Also
696: (4)                --------
697: (4)                read_gml, generate_gml
698: (4)                literal_stringizer
699: (4)                Notes
700: (4)                -----
701: (4)                Graph attributes named 'directed', 'multigraph', 'node' or
702: (4)                'edge', node attributes named 'id' or 'label', edge attributes
703: (4)                named 'source' or 'target' (or 'key' if `G` is a multigraph)
704: (4)                are ignored because these attribute names are used to encode the graph
705: (4)                structure.
706: (4)                GML files are stored using a 7-bit ASCII encoding with any extended
707: (4)                ASCII characters (iso8859-1) appearing as HTML character entities.
708: (4)                Without specifying a `stringizer`/`destringizer`, the code is capable of
709: (4)                writing `int`/`float`/`str`/`dict`/`list` data as required by the GML
710: (4)                specification.  For writing other data types, and for reading data other
711: (4)                than `str` you need to explicitly supply a `stringizer`/`destringizer`.
712: (4)                Note that while we allow non-standard GML to be read from a file, we make
713: (4)                sure to write GML format. In particular, underscores are not allowed in
714: (4)                attribute names.
715: (4)                For additional documentation on the GML file format, please see the
716: (4)                `GML url <https://web.archive.org/web/20190207140002/http://www.fim.uni-
passau.de/index.php?id=17297&L=1>`_.
717: (4)                See the module docstring :mod:`networkx.readwrite.gml` for more details.
718: (4)                Examples
719: (4)                --------
720: (4)                >>> G = nx.path_graph(4)
721: (4)                >>> nx.write_gml(G, "test.gml")
722: (4)                Filenames ending in .gz or .bz2 will be compressed.
723: (4)                >>> nx.write_gml(G, "test.gml.gz")
724: (4)                """
725: (4)                for line in generate_gml(G, stringizer):
726: (8)                    path.write((line + "\n").encode("ascii"))

----------------------------------------

File 506 - graph6.py:

1: (0)              """Functions for reading and writing graphs in the *graph6* format.
2: (0)              The *graph6* file format is suitable for small graphs or large dense
3: (0)              graphs. For large sparse graphs, use the *sparse6* format.
4: (0)              For more information, see the `graph6`_ homepage.
5: (0)              .. _graph6: http://users.cecs.anu.edu.au/~bdm/data/formats.html
6: (0)              """
7: (0)              from itertools import islice
8: (0)              import networkx as nx
9: (0)              from networkx.exception import NetworkXError
10: (0)             from networkx.utils import not_implemented_for, open_file
11: (0)             __all__ = ["from_graph6_bytes", "read_graph6", "to_graph6_bytes",
"write_graph6"]
12: (0)             def _generate_graph6_bytes(G, nodes, header):
13: (4)                 """Yield bytes in the graph6 encoding of a graph.
14: (4)                 `G` is an undirected simple graph. `nodes` is the list of nodes for
15: (4)                 which the node-induced subgraph will be encoded; if `nodes` is the
16: (4)                 list of all nodes in the graph, the entire graph will be
17: (4)                 encoded. `header` is a Boolean that specifies whether to generate
18: (4)                 the header ``b'>>graph6<<'`` before the remaining data.
19: (4)                 This function generates `bytes` objects in the following order:
```

```
(4)                 1. the header (if requested),
(4)                 2. the encoding of the number of nodes,
(4)                 3. each character, one-at-a-time, in the encoding of the requested
(7)                    node-induced subgraph,
(4)                 4. a newline character.
(4)                 This function raises :exc:`ValueError` if the graph is too large for
(4)                 the graph6 format (that is, greater than ``2 ** 36`` nodes).
(4)                 """
(4)                 n = len(G)
(4)                 if n >= 2**36:
(8)                     raise ValueError(
(12)                        "graph6 is only defined if number of nodes is less than 2 ** 36"
(8)                     )
(4)                 if header:
(8)                     yield b">>graph6<<"
(4)                 for d in n_to_data(n):
(8)                     yield str.encode(chr(d + 63))
(4)                 bits = (nodes[j] in G[nodes[i]] for j in range(1, n) for i in range(j))
(4)                 chunk = list(islice(bits, 6))
(4)                 while chunk:
(8)                     d = sum(b << 5 - i for i, b in enumerate(chunk))
(8)                     yield str.encode(chr(d + 63))
(8)                     chunk = list(islice(bits, 6))
(4)                 yield b"\n"
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def from_graph6_bytes(bytes_in):
(4)                 """Read a simple undirected graph in graph6 format from bytes.
(4)                 Parameters
(4)                 ----------
(4)                 bytes_in : bytes
(7)                    Data in graph6 format, without a trailing newline.
(4)                 Returns
(4)                 -------
(4)                 G : Graph
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If bytes_in is unable to be parsed in graph6 format
(4)                 ValueError
(8)                     If any character ``c`` in bytes_in does not satisfy
(8)                     ``63 <= ord(c) < 127``.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.from_graph6_bytes(b"A_")
(4)                 >>> sorted(G.edges())
(4)                 [(0, 1)]
(4)                 See Also
(4)                 --------
(4)                 read_graph6, write_graph6
(4)                 References
(4)                 ----------
(4)                 .. [1] Graph6 specification
(11)                       <http://users.cecs.anu.edu.au/~bdm/data/formats.html>
(4)                 """
(4)                 def bits():
(8)                     """Returns sequence of individual bits from 6-bit-per-value
(8)                     list of data values."""
(8)                     for d in data:
(12)                        for i in [5, 4, 3, 2, 1, 0]:
(16)                            yield (d >> i) & 1
(4)                 if bytes_in.startswith(b">>graph6<<"):
(8)                     bytes_in = bytes_in[10:]
(4)                 data = [c - 63 for c in bytes_in]
(4)                 if any(c > 63 for c in data):
(8)                     raise ValueError("each input character must be in range(63, 127)")
(4)                 n, data = data_to_n(data)
(4)                 nd = (n * (n - 1) // 2 + 5) // 6
(4)                 if len(data) != nd:
(8)                     raise NetworkXError(
```

```
89: (12)                        f"Expected {n * (n - 1) // 2} bits but got {len(data) * 6} in
graph6"
90: (8)                     )
91: (4)                 G = nx.Graph()
92: (4)                 G.add_nodes_from(range(n))
93: (4)                 for (i, j), b in zip(((i, j) for j in range(1, n) for i in range(j)),
bits()):
94: (8)                     if b:
95: (12)                        G.add_edge(i, j)
96: (4)                 return G
97: (0)             @not_implemented_for("directed")
98: (0)             @not_implemented_for("multigraph")
99: (0)             def to_graph6_bytes(G, nodes=None, header=True):
100: (4)                """Convert a simple undirected graph to bytes in graph6 format.
101: (4)                Parameters
102: (4)                ----------
103: (4)                G : Graph (undirected)
104: (4)                nodes: list or iterable
105: (7)                   Nodes are labeled 0...n-1 in the order provided.  If None the ordering
106: (7)                   given by ``G.nodes()`` is used.
107: (4)                header: bool
108: (7)                   If True add '>>graph6<<' bytes to head of data.
109: (4)                Raises
110: (4)                ------
111: (4)                NetworkXNotImplemented
112: (8)                    If the graph is directed or is a multigraph.
113: (4)                ValueError
114: (8)                    If the graph has at least ``2 ** 36`` nodes; the graph6 format
115: (8)                    is only defined for graphs of order less than ``2 ** 36``.
116: (4)                Examples
117: (4)                --------
118: (4)                >>> nx.to_graph6_bytes(nx.path_graph(2))
119: (4)                b'>>graph6<<A_\\n'
120: (4)                See Also
121: (4)                --------
122: (4)                from_graph6_bytes, read_graph6, write_graph6_bytes
123: (4)                Notes
124: (4)                -----
125: (4)                The returned bytes end with a newline character.
126: (4)                The format does not support edge or node labels, parallel edges or
127: (4)                self loops. If self loops are present they are silently ignored.
128: (4)                References
129: (4)                ----------
130: (4)                .. [1] Graph6 specification
131: (11)                      <http://users.cecs.anu.edu.au/~bdm/data/formats.html>
132: (4)                """
133: (4)                if nodes is not None:
134: (8)                    G = G.subgraph(nodes)
135: (4)                H = nx.convert_node_labels_to_integers(G)
136: (4)                nodes = sorted(H.nodes())
137: (4)                return b"".join(_generate_graph6_bytes(H, nodes, header))
138: (0)            @open_file(0, mode="rb")
139: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
140: (0)            def read_graph6(path):
141: (4)                """Read simple undirected graphs in graph6 format from path.
142: (4)                Parameters
143: (4)                ----------
144: (4)                path : file or string
145: (7)                   File or filename to write.
146: (4)                Returns
147: (4)                -------
148: (4)                G : Graph or list of Graphs
149: (7)                   If the file contains multiple lines then a list of graphs is returned
150: (4)                Raises
151: (4)                ------
152: (4)                NetworkXError
153: (8)                    If the string is unable to be parsed in graph6 format
154: (4)                Examples
155: (4)                --------
```

```
    You can read a graph6 file by giving the path to the file::
        >>> import tempfile
        >>> with tempfile.NamedTemporaryFile(delete=False) as f:
        ...     _ = f.write(b">>graph6<<A_\\n")
        ...     _ = f.seek(0)
        ...     G = nx.read_graph6(f.name)
        >>> list(G.edges())
        [(0, 1)]
    You can also read a graph6 file by giving an open file-like object::
        >>> import tempfile
        >>> with tempfile.NamedTemporaryFile() as f:
        ...     _ = f.write(b">>graph6<<A_\\n")
        ...     _ = f.seek(0)
        ...     G = nx.read_graph6(f)
        >>> list(G.edges())
        [(0, 1)]
    See Also
    --------
    from_graph6_bytes, write_graph6
    References
    ----------
    .. [1] Graph6 specification
           <http://users.cecs.anu.edu.au/~bdm/data/formats.html>
    """
    glist = []
    for line in path:
        line = line.strip()
        if not len(line):
            continue
        glist.append(from_graph6_bytes(line))
    if len(glist) == 1:
        return glist[0]
    else:
        return glist
@not_implemented_for("directed")
@not_implemented_for("multigraph")
@open_file(1, mode="wb")
def write_graph6(G, path, nodes=None, header=True):
    """Write a simple undirected graph to a path in graph6 format.
    Parameters
    ----------
    G : Graph (undirected)
    path : str
       The path naming the file to which to write the graph.
    nodes: list or iterable
       Nodes are labeled 0...n-1 in the order provided.  If None the ordering
       given by ``G.nodes()`` is used.
    header: bool
       If True add '>>graph6<<' string to head of data
    Raises
    ------
    NetworkXNotImplemented
        If the graph is directed or is a multigraph.
    ValueError
        If the graph has at least ``2 ** 36`` nodes; the graph6 format
        is only defined for graphs of order less than ``2 ** 36``.
    Examples
    --------
    You can write a graph6 file by giving the path to a file::
        >>> import tempfile
        >>> with tempfile.NamedTemporaryFile(delete=False) as f:
        ...     nx.write_graph6(nx.path_graph(2), f.name)
        ...     _ = f.seek(0)
        ...     print(f.read())
        b'>>graph6<<A_\\n'
    See Also
    --------
    from_graph6_bytes, read_graph6
    Notes
```

```
(4)                 -----
(4)                 The function writes a newline character after writing the encoding
(4)                 of the graph.
(4)                 The format does not support edge or node labels, parallel edges or
(4)                 self loops.  If self loops are present they are silently ignored.
(4)                 References
(4)                 ----------
(4)                 .. [1] Graph6 specification
(11)                       <http://users.cecs.anu.edu.au/~bdm/data/formats.html>
(4)                 """
(4)                 return write_graph6_file(G, path, nodes=nodes, header=header)
(0)             @not_implemented_for("directed")
(0)             @not_implemented_for("multigraph")
(0)             def write_graph6_file(G, f, nodes=None, header=True):
(4)                 """Write a simple undirected graph to a file-like object in graph6 format.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph (undirected)
(4)                 f : file-like object
(7)                    The file to write.
(4)                 nodes: list or iterable
(7)                    Nodes are labeled 0...n-1 in the order provided.  If None the ordering
(7)                    given by ``G.nodes()`` is used.
(4)                 header: bool
(7)                    If True add '>>graph6<<' string to head of data
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If the graph is directed or is a multigraph.
(4)                 ValueError
(8)                     If the graph has at least ``2 ** 36`` nodes; the graph6 format
(8)                     is only defined for graphs of order less than ``2 ** 36``.
(4)                 Examples
(4)                 --------
(4)                 You can write a graph6 file by giving an open file-like object::
(8)                     >>> import tempfile
(8)                     >>> with tempfile.NamedTemporaryFile() as f:
(8)                     ...     nx.write_graph6(nx.path_graph(2), f)
(8)                     ...     _ = f.seek(0)
(8)                     ...     print(f.read())
(8)                     b'>>graph6<<A_\\n'
(4)                 See Also
(4)                 --------
(4)                 from_graph6_bytes, read_graph6
(4)                 Notes
(4)                 -----
(4)                 The function writes a newline character after writing the encoding
(4)                 of the graph.
(4)                 The format does not support edge or node labels, parallel edges or
(4)                 self loops.  If self loops are present they are silently ignored.
(4)                 References
(4)                 ----------
(4)                 .. [1] Graph6 specification
(11)                       <http://users.cecs.anu.edu.au/~bdm/data/formats.html>
(4)                 """
(4)                 if nodes is not None:
(8)                     G = G.subgraph(nodes)
(4)                 H = nx.convert_node_labels_to_integers(G)
(4)                 nodes = sorted(H.nodes())
(4)                 for b in _generate_graph6_bytes(H, nodes, header):
(8)                     f.write(b)
(0)             def data_to_n(data):
(4)                 """Read initial one-, four- or eight-unit value from graph6
(4)                 integer sequence.
(4)                 Return (value, rest of seq.)"""
(4)                 if data[0] <= 62:
(8)                     return data[0], data[1:]
(4)                 if data[1] <= 62:
(8)                     return (data[1] << 12) + (data[2] << 6) + data[3], data[4:]
```

```
294: (4)                return (
295: (8)                    (data[2] << 30)
296: (8)                    + (data[3] << 24)
297: (8)                    + (data[4] << 18)
298: (8)                    + (data[5] << 12)
299: (8)                    + (data[6] << 6)
300: (8)                    + data[7],
301: (8)                    data[8:],
302: (4)                )
303: (0)            def n_to_data(n):
304: (4)                """Convert an integer to one-, four- or eight-unit graph6 sequence.
305: (4)                This function is undefined if `n` is not in ``range(2 ** 36)``.
306: (4)                """
307: (4)                if n <= 62:
308: (8)                    return [n]
309: (4)                elif n <= 258047:
310: (8)                    return [63, (n >> 12) & 0x3F, (n >> 6) & 0x3F, n & 0x3F]
311: (4)                else:  # if n <= 68719476735:
312: (8)                    return [
313: (12)                       63,
314: (12)                       63,
315: (12)                       (n >> 30) & 0x3F,
316: (12)                       (n >> 24) & 0x3F,
317: (12)                       (n >> 18) & 0x3F,
318: (12)                       (n >> 12) & 0x3F,
319: (12)                       (n >> 6) & 0x3F,
320: (12)                       n & 0x3F,
321: (8)                    ]
```

File 507 - graphml.py:

```
1: (0)              """
2: (0)              *******
3: (0)              GraphML
4: (0)              *******
5: (0)              Read and write graphs in GraphML format.
6: (0)              .. warning::
7: (4)                  This parser uses the standard xml library present in Python, which is
8: (4)                  insecure - see :external+python:mod:`xml` for additional information.
9: (4)                  Only parse GraphML files you trust.
10: (0)             This implementation does not support mixed graphs (directed and unidirected
11: (0)             edges together), hyperedges, nested graphs, or ports.
12: (0)             "GraphML is a comprehensive and easy-to-use file format for graphs. It
13: (0)             consists of a language core to describe the structural properties of a
14: (0)             graph and a flexible extension mechanism to add application-specific
15: (0)             data. Its main features include support of
16: (4)                 * directed, undirected, and mixed graphs,
17: (4)                 * hypergraphs,
18: (4)                 * hierarchical graphs,
19: (4)                 * graphical representations,
20: (4)                 * references to external data,
21: (4)                 * application-specific attribute data, and
22: (4)                 * light-weight parsers.
23: (0)             Unlike many other file formats for graphs, GraphML does not use a
24: (0)             custom syntax. Instead, it is based on XML and hence ideally suited as
25: (0)             a common denominator for all kinds of services generating, archiving,
26: (0)             or processing graphs."
27: (0)             http://graphml.graphdrawing.org/
28: (0)             Format
29: (0)             ------
30: (0)             GraphML is an XML format.  See
31: (0)             http://graphml.graphdrawing.org/specification.html for the specification and
32: (0)             http://graphml.graphdrawing.org/primer/graphml-primer.html
33: (0)             for examples.
34: (0)             """
35: (0)             import warnings
36: (0)             from collections import defaultdict
```

```
 37: (0)             import networkx as nx
 38: (0)             from networkx.utils import open_file
 39: (0)             __all__ = [
 40: (4)                 "write_graphml",
 41: (4)                 "read_graphml",
 42: (4)                 "generate_graphml",
 43: (4)                 "write_graphml_xml",
 44: (4)                 "write_graphml_lxml",
 45: (4)                 "parse_graphml",
 46: (4)                 "GraphMLWriter",
 47: (4)                 "GraphMLReader",
 48: (0)             ]
 49: (0)             @open_file(1, mode="wb")
 50: (0)             def write_graphml_xml(
 51: (4)                 G,
 52: (4)                 path,
 53: (4)                 encoding="utf-8",
 54: (4)                 prettyprint=True,
 55: (4)                 infer_numeric_types=False,
 56: (4)                 named_key_ids=False,
 57: (4)                 edge_id_from_attribute=None,
 58: (0)             ):
 59: (4)                 """Write G in GraphML XML format to path
 60: (4)                 Parameters
 61: (4)                 ----------
 62: (4)                 G : graph
 63: (7)                    A networkx graph
 64: (4)                 path : file or string
 65: (7)                    File or filename to write.
 66: (7)                    Filenames ending in .gz or .bz2 will be compressed.
 67: (4)                 encoding : string (optional)
 68: (7)                    Encoding for text data.
 69: (4)                 prettyprint : bool (optional)
 70: (7)                    If True use line breaks and indenting in output XML.
 71: (4)                 infer_numeric_types : boolean
 72: (7)                    Determine if numeric types should be generalized.
 73: (7)                    For example, if edges have both int and float 'weight' attributes,
 74: (7)                    we infer in GraphML that both are floats.
 75: (4)                 named_key_ids : bool (optional)
 76: (7)                    If True use attr.name as value for key elements' id attribute.
 77: (4)                 edge_id_from_attribute : dict key (optional)
 78: (8)                     If provided, the graphml edge id is set by looking up the
corresponding
 79: (8)                     edge data attribute keyed by this parameter. If `None` or the key does
not exist in edge data,
 80: (8)                     the edge id is set by the edge key if `G` is a MultiGraph, else the
edge id is left unset.
 81: (4)                 Examples
 82: (4)                 --------
 83: (4)                 >>> G = nx.path_graph(4)
 84: (4)                 >>> nx.write_graphml(G, "test.graphml")
 85: (4)                 Notes
 86: (4)                 -----
 87: (4)                 This implementation does not support mixed graphs (directed
 88: (4)                 and unidirected edges together) hyperedges, nested graphs, or ports.
 89: (4)                 """
 90: (4)                 writer = GraphMLWriter(
 91: (8)                     encoding=encoding,
 92: (8)                     prettyprint=prettyprint,
 93: (8)                     infer_numeric_types=infer_numeric_types,
 94: (8)                     named_key_ids=named_key_ids,
 95: (8)                     edge_id_from_attribute=edge_id_from_attribute,
 96: (4)                 )
 97: (4)                 writer.add_graph_element(G)
 98: (4)                 writer.dump(path)
 99: (0)             @open_file(1, mode="wb")
100: (0)             def write_graphml_lxml(
101: (4)                 G,
102: (4)                 path,
```

```
(4)                 encoding="utf-8",
(4)                 prettyprint=True,
(4)                 infer_numeric_types=False,
(4)                 named_key_ids=False,
(4)                 edge_id_from_attribute=None,
(0)             ):
(4)                 """Write G in GraphML XML format to path
(4)                 This function uses the LXML framework and should be faster than
(4)                 the version using the xml library.
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A networkx graph
(4)                 path : file or string
(7)                    File or filename to write.
(7)                    Filenames ending in .gz or .bz2 will be compressed.
(4)                 encoding : string (optional)
(7)                    Encoding for text data.
(4)                 prettyprint : bool (optional)
(7)                    If True use line breaks and indenting in output XML.
(4)                 infer_numeric_types : boolean
(7)                    Determine if numeric types should be generalized.
(7)                    For example, if edges have both int and float 'weight' attributes,
(7)                    we infer in GraphML that both are floats.
(4)                 named_key_ids : bool (optional)
(7)                    If True use attr.name as value for key elements' id attribute.
(4)                 edge_id_from_attribute : dict key (optional)
(8)                     If provided, the graphml edge id is set by looking up the corresponding
(8)                     edge data attribute keyed by this parameter. If `None` or the key does not exist in edge data,
(8)                     the edge id is set by the edge key if `G` is a MultiGraph, else the edge id is left unset.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> nx.write_graphml_lxml(G, "fourpath.graphml")
(4)                 Notes
(4)                 -----
(4)                 This implementation does not support mixed graphs (directed
(4)                 and unidirected edges together) hyperedges, nested graphs, or ports.
(4)                 """
(4)                 try:
(8)                     import lxml.etree as lxmletree
(4)                 except ImportError:
(8)                     return write_graphml_xml(
(12)                        G,
(12)                        path,
(12)                        encoding,
(12)                        prettyprint,
(12)                        infer_numeric_types,
(12)                        named_key_ids,
(12)                        edge_id_from_attribute,
(8)                     )
(4)                 writer = GraphMLWriterLxml(
(8)                     path,
(8)                     graph=G,
(8)                     encoding=encoding,
(8)                     prettyprint=prettyprint,
(8)                     infer_numeric_types=infer_numeric_types,
(8)                     named_key_ids=named_key_ids,
(8)                     edge_id_from_attribute=edge_id_from_attribute,
(4)                 )
(4)                 writer.dump()
(0)             def generate_graphml(
(4)                 G,
(4)                 encoding="utf-8",
(4)                 prettyprint=True,
(4)                 named_key_ids=False,
```

```
(4)                 edge_id_from_attribute=None,
(0)             ):
(4)                 """Generate GraphML lines for G
(4)                 Parameters
(4)                 ----------
(4)                 G : graph
(7)                    A networkx graph
(4)                 encoding : string (optional)
(7)                    Encoding for text data.
(4)                 prettyprint : bool (optional)
(7)                    If True use line breaks and indenting in output XML.
(4)                 named_key_ids : bool (optional)
(7)                    If True use attr.name as value for key elements' id attribute.
(4)                 edge_id_from_attribute : dict key (optional)
(8)                     If provided, the graphml edge id is set by looking up the corresponding
(8)                     edge data attribute keyed by this parameter. If `None` or the key does not exist in edge data,
(8)                     the edge id is set by the edge key if `G` is a MultiGraph, else the edge id is left unset.
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(4)
(4)                 >>> linefeed = chr(10)  # linefeed = \n
(4)                 >>> s = linefeed.join(nx.generate_graphml(G))
(4)                 >>> for line in nx.generate_graphml(G):  # doctest: +SKIP
(4)                 ...     print(line)
(4)                 Notes
(4)                 -----
(4)                 This implementation does not support mixed graphs (directed and unidirected
(4)                 edges together) hyperedges, nested graphs, or ports.
(4)                 """
(4)                 writer = GraphMLWriter(
(8)                     encoding=encoding,
(8)                     prettyprint=prettyprint,
(8)                     named_key_ids=named_key_ids,
(8)                     edge_id_from_attribute=edge_id_from_attribute,
(4)                 )
(4)                 writer.add_graph_element(G)
(4)                 yield from str(writer).splitlines()
(0)             @open_file(0, mode="rb")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def read_graphml(path, node_type=str, edge_key_type=int, force_multigraph=False):
(4)                 """Read graph in GraphML format from path.
(4)                 Parameters
(4)                 ----------
(4)                 path : file or string
(7)                    File or filename to write.
(7)                    Filenames ending in .gz or .bz2 will be compressed.
(4)                 node_type: Python type (default: str)
(7)                    Convert node ids to this type
(4)                 edge_key_type: Python type (default: int)
(7)                    Convert graphml edge ids to this type. Multigraphs use id as edge key.
(7)                    Non-multigraphs add to edge attribute dict with name "id".
(4)                 force_multigraph : bool (default: False)
(7)                    If True, return a multigraph with edge keys. If False (the default)
(7)                    return a multigraph when multiedges are in the graph.
(4)                 Returns
(4)                 -------
(4)                 graph: NetworkX graph
(8)                     If parallel edges are present or `force_multigraph=True` then
(8)                     a MultiGraph or MultiDiGraph is returned. Otherwise a Graph/DiGraph.
(8)                     The returned graph is directed if the file indicates it should be.
(4)                 Notes
(4)                 -----
(4)                 Default node and edge attributes are not propagated to each node and edge.
(4)                 They can be obtained from `G.graph` and applied to node and edge
```

```
attributes
    if desired using something like this:
    >>> default_color = G.graph["node_default"]["color"]  # doctest: +SKIP
    >>> for node, data in G.nodes(data=True):  # doctest: +SKIP
    ...     if "color" not in data:
    ...         data["color"] = default_color
    >>> default_color = G.graph["edge_default"]["color"]  # doctest: +SKIP
    >>> for u, v, data in G.edges(data=True):  # doctest: +SKIP
    ...     if "color" not in data:
    ...         data["color"] = default_color
    This implementation does not support mixed graphs (directed and
unidirected
    edges together), hypergraphs, nested graphs, or ports.
    For multigraphs the GraphML edge "id" will be used as the edge
    key.  If not specified then they "key" attribute will be used.  If
    there is no "key" attribute a default NetworkX multigraph edge key
    will be provided.
    Files with the yEd "yfiles" extension can be read. The type of the node's
    shape is preserved in the `shape_type` node attribute.
    yEd compressed files ("file.graphmlz" extension) can be read by renaming
    the file to "file.graphml.gz".
    """
    reader = GraphMLReader(node_type, edge_key_type, force_multigraph)
    glist = list(reader(path=path))
    if len(glist) == 0:
        header = b'<graphml xmlns="http://graphml.graphdrawing.org/xmlns">'
        path.seek(0)
        old_bytes = path.read()
        new_bytes = old_bytes.replace(b"<graphml>", header)
        glist = list(reader(string=new_bytes))
        if len(glist) == 0:
            raise nx.NetworkXError("file not successfully read as graphml")
    return glist[0]
@nx._dispatchable(graphs=None, returns_graph=True)
def parse_graphml(
    graphml_string, node_type=str, edge_key_type=int, force_multigraph=False
):
    """Read graph in GraphML format from string.
    Parameters
    ----------
    graphml_string : string
       String containing graphml information
       (e.g., contents of a graphml file).
    node_type: Python type (default: str)
       Convert node ids to this type
    edge_key_type: Python type (default: int)
       Convert graphml edge ids to this type. Multigraphs use id as edge key.
       Non-multigraphs add to edge attribute dict with name "id".
    force_multigraph : bool (default: False)
       If True, return a multigraph with edge keys. If False (the default)
       return a multigraph when multiedges are in the graph.
    Returns
    -------
    graph: NetworkX graph
        If no parallel edges are found a Graph or DiGraph is returned.
        Otherwise a MultiGraph or MultiDiGraph is returned.
    Examples
    --------
    >>> G = nx.path_graph(4)
    >>> linefeed = chr(10)  # linefeed = \n
    >>> s = linefeed.join(nx.generate_graphml(G))
    >>> H = nx.parse_graphml(s)
    Notes
    -----
    Default node and edge attributes are not propagated to each node and edge.
    They can be obtained from `G.graph` and applied to node and edge
attributes
    if desired using something like this:
    >>> default_color = G.graph["node_default"]["color"]  # doctest: +SKIP
```

```
299: (4)                >>> for node, data in G.nodes(data=True):  # doctest: +SKIP
300: (4)                ...     if "color" not in data:
301: (4)                ...         data["color"] = default_color
302: (4)                >>> default_color = G.graph["edge_default"]["color"]  # doctest: +SKIP
303: (4)                >>> for u, v, data in G.edges(data=True):  # doctest: +SKIP
304: (4)                ...     if "color" not in data:
305: (4)                ...         data["color"] = default_color
306: (4)                This implementation does not support mixed graphs (directed and
unidirected
307: (4)                edges together), hypergraphs, nested graphs, or ports.
308: (4)                For multigraphs the GraphML edge "id" will be used as the edge
309: (4)                key.  If not specified then they "key" attribute will be used.  If
310: (4)                there is no "key" attribute a default NetworkX multigraph edge key
311: (4)                will be provided.
312: (4)                """
313: (4)                reader = GraphMLReader(node_type, edge_key_type, force_multigraph)
314: (4)                glist = list(reader(string=graphml_string))
315: (4)                if len(glist) == 0:
316: (8)                    header = '<graphml xmlns="http://graphml.graphdrawing.org/xmlns">'
317: (8)                    new_string = graphml_string.replace("<graphml>", header)
318: (8)                    glist = list(reader(string=new_string))
319: (8)                    if len(glist) == 0:
320: (12)                       raise nx.NetworkXError("file not successfully read as graphml")
321: (4)                return glist[0]
322: (0)            class GraphML:
323: (4)                NS_GRAPHML = "http://graphml.graphdrawing.org/xmlns"
324: (4)                NS_XSI = "http://www.w3.org/2001/XMLSchema-instance"
325: (4)                NS_Y = "http://www.yworks.com/xml/graphml"
326: (4)                SCHEMALOCATION = " ".join(
327: (8)                    [
328: (12)                       "http://graphml.graphdrawing.org/xmlns",
329: (12)                       "http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd",
330: (8)                    ]
331: (4)                )
332: (4)                def construct_types(self):
333: (8)                    types = [
334: (12)                       (int, "integer"),  # for Gephi GraphML bug
335: (12)                       (str, "yfiles"),
336: (12)                       (str, "string"),
337: (12)                       (int, "int"),
338: (12)                       (int, "long"),
339: (12)                       (float, "float"),
340: (12)                       (float, "double"),
341: (12)                       (bool, "boolean"),
342: (8)                    ]
343: (8)                    try:
344: (12)                       import numpy as np
345: (8)                    except:
346: (12)                       pass
347: (8)                    else:
348: (12)                       types = [
349: (16)                           (np.float64, "float"),
350: (16)                           (np.float32, "float"),
351: (16)                           (np.float16, "float"),
352: (16)                           (np.int_, "int"),
353: (16)                           (np.int8, "int"),
354: (16)                           (np.int16, "int"),
355: (16)                           (np.int32, "int"),
356: (16)                           (np.int64, "int"),
357: (16)                           (np.uint8, "int"),
358: (16)                           (np.uint16, "int"),
359: (16)                           (np.uint32, "int"),
360: (16)                           (np.uint64, "int"),
361: (16)                           (np.int_, "int"),
362: (16)                           (np.intc, "int"),
363: (16)                           (np.intp, "int"),
364: (12)                       ] + types
365: (8)                    self.xml_type = dict(types)
366: (8)                    self.python_type = dict(reversed(a) for a in types)
```

```
(4)                 convert_bool = {
(8)                     "true": True,
(8)                     "false": False,
(8)                     "0": False,
(8)                     0: False,
(8)                     "1": True,
(8)                     1: True,
(4)                 }
(4)                 def get_xml_type(self, key):
(8)                     """Wrapper around the xml_type dict that raises a more informative
(8)                     exception message when a user attempts to use data of a type not
(8)                     supported by GraphML."""
(8)                     try:
(12)                        return self.xml_type[key]
(8)                     except KeyError as err:
(12)                        raise TypeError(
(16)                            f"GraphML does not support type {key} as data values."
(12)                        ) from err
(0)             class GraphMLWriter(GraphML):
(4)                 def __init__(
(8)                     self,
(8)                     graph=None,
(8)                     encoding="utf-8",
(8)                     prettyprint=True,
(8)                     infer_numeric_types=False,
(8)                     named_key_ids=False,
(8)                     edge_id_from_attribute=None,
(4)                 ):
(8)                     self.construct_types()
(8)                     from xml.etree.ElementTree import Element
(8)                     self.myElement = Element
(8)                     self.infer_numeric_types = infer_numeric_types
(8)                     self.prettyprint = prettyprint
(8)                     self.named_key_ids = named_key_ids
(8)                     self.edge_id_from_attribute = edge_id_from_attribute
(8)                     self.encoding = encoding
(8)                     self.xml = self.myElement(
(12)                        "graphml",
(12)                        {
(16)                            "xmlns": self.NS_GRAPHML,
(16)                            "xmlns:xsi": self.NS_XSI,
(16)                            "xsi:schemaLocation": self.SCHEMALOCATION,
(12)                        },
(8)                     )
(8)                     self.keys = {}
(8)                     self.attributes = defaultdict(list)
(8)                     self.attribute_types = defaultdict(set)
(8)                     if graph is not None:
(12)                        self.add_graph_element(graph)
(4)                 def __str__(self):
(8)                     from xml.etree.ElementTree import tostring
(8)                     if self.prettyprint:
(12)                        self.indent(self.xml)
(8)                     s = tostring(self.xml).decode(self.encoding)
(8)                     return s
(4)                 def attr_type(self, name, scope, value):
(8)                     """Infer the attribute type of data named name. Currently this only
(8)                     supports inference of numeric types.
(8)                     If self.infer_numeric_types is false, type is used. Otherwise, pick
the
(8)                     most general of types found across all values with name and scope.
This
(8)                     means edges with data named 'weight' are treated separately from nodes
(8)                     with data named 'weight'.
(8)                     """
(8)                     if self.infer_numeric_types:
(12)                        types = self.attribute_types[(name, scope)]
(12)                        if len(types) > 1:
(16)                            types = {self.get_xml_type(t) for t in types}
```

```
434: (16)                            if "string" in types:
435: (20)                                return str
436: (16)                            elif "float" in types or "double" in types:
437: (20)                                return float
438: (16)                            else:
439: (20)                                return int
440: (12)                        else:
441: (16)                            return list(types)[0]
442: (8)                    else:
443: (12)                        return type(value)
444: (4)                def get_key(self, name, attr_type, scope, default):
445: (8)                    keys_key = (name, attr_type, scope)
446: (8)                    try:
447: (12)                        return self.keys[keys_key]
448: (8)                    except KeyError:
449: (12)                        if self.named_key_ids:
450: (16)                            new_id = name
451: (12)                        else:
452: (16)                            new_id = f"d{len(list(self.keys))}"
453: (12)                        self.keys[keys_key] = new_id
454: (12)                        key_kwargs = {
455: (16)                            "id": new_id,
456: (16)                            "for": scope,
457: (16)                            "attr.name": name,
458: (16)                            "attr.type": attr_type,
459: (12)                        }
460: (12)                        key_element = self.myElement("key", **key_kwargs)
461: (12)                        if default is not None:
462: (16)                            default_element = self.myElement("default")
463: (16)                            default_element.text = str(default)
464: (16)                            key_element.append(default_element)
465: (12)                        self.xml.insert(0, key_element)
466: (8)                    return new_id
467: (4)                def add_data(self, name, element_type, value, scope="all", default=None):
468: (8)                    """
469: (8)                    Make a data element for an edge or a node. Keep a log of the
470: (8)                    type in the keys table.
471: (8)                    """
472: (8)                    if element_type not in self.xml_type:
473: (12)                        raise nx.NetworkXError(
474: (16)                            f"GraphML writer does not support {element_type} as data
values."
475: (12)                        )
476: (8)                    keyid = self.get_key(name, self.get_xml_type(element_type), scope,
default)
477: (8)                    data_element = self.myElement("data", key=keyid)
478: (8)                    data_element.text = str(value)
479: (8)                    return data_element
480: (4)                def add_attributes(self, scope, xml_obj, data, default):
481: (8)                    """Appends attribute data to edges or nodes, and stores type
information
482: (8)                    to be added later. See add_graph_element.
483: (8)                    """
484: (8)                    for k, v in data.items():
485: (12)                        self.attribute_types[(str(k), scope)].add(type(v))
486: (12)                        self.attributes[xml_obj].append([k, v, scope, default.get(k)])
487: (4)                def add_nodes(self, G, graph_element):
488: (8)                    default = G.graph.get("node_default", {})
489: (8)                    for node, data in G.nodes(data=True):
490: (12)                        node_element = self.myElement("node", id=str(node))
491: (12)                        self.add_attributes("node", node_element, data, default)
492: (12)                        graph_element.append(node_element)
493: (4)                def add_edges(self, G, graph_element):
494: (8)                    if G.is_multigraph():
495: (12)                        for u, v, key, data in G.edges(data=True, keys=True):
496: (16)                            edge_element = self.myElement(
497: (20)                                "edge",
498: (20)                                source=str(u),
499: (20)                                target=str(v),
```

```
500: (20)                                id=str(data.get(self.edge_id_from_attribute))
501: (20)                                if self.edge_id_from_attribute
502: (20)                                and self.edge_id_from_attribute in data
503: (20)                                else str(key),
504: (16)                            )
505: (16)                            default = G.graph.get("edge_default", {})
506: (16)                            self.add_attributes("edge", edge_element, data, default)
507: (16)                            graph_element.append(edge_element)
508: (8)                    else:
509: (12)                        for u, v, data in G.edges(data=True):
510: (16)                            if self.edge_id_from_attribute and self.edge_id_from_attribute
in data:
511: (20)                                edge_element = self.myElement(
512: (24)                                    "edge",
513: (24)                                    source=str(u),
514: (24)                                    target=str(v),
515: (24)                                    id=str(data.get(self.edge_id_from_attribute)),
516: (20)                                )
517: (16)                            else:
518: (20)                                edge_element = self.myElement("edge", source=str(u),
target=str(v))
519: (16)                            default = G.graph.get("edge_default", {})
520: (16)                            self.add_attributes("edge", edge_element, data, default)
521: (16)                            graph_element.append(edge_element)
522: (4)                def add_graph_element(self, G):
523: (8)                    """
524: (8)                    Serialize graph G in GraphML to the stream.
525: (8)                    """
526: (8)                    if G.is_directed():
527: (12)                       default_edge_type = "directed"
528: (8)                    else:
529: (12)                       default_edge_type = "undirected"
530: (8)                    graphid = G.graph.pop("id", None)
531: (8)                    if graphid is None:
532: (12)                       graph_element = self.myElement("graph",
edgedefault=default_edge_type)
533: (8)                    else:
534: (12)                       graph_element = self.myElement(
535: (16)                           "graph", edgedefault=default_edge_type, id=graphid
536: (12)                       )
537: (8)                    default = {}
538: (8)                    data = {
539: (12)                       k: v
540: (12)                       for (k, v) in G.graph.items()
541: (12)                       if k not in ["node_default", "edge_default"]
542: (8)                    }
543: (8)                    self.add_attributes("graph", graph_element, data, default)
544: (8)                    self.add_nodes(G, graph_element)
545: (8)                    self.add_edges(G, graph_element)
546: (8)                    for xml_obj, data in self.attributes.items():
547: (12)                       for k, v, scope, default in data:
548: (16)                           xml_obj.append(
549: (20)                               self.add_data(
550: (24)                                   str(k), self.attr_type(k, scope, v), str(v), scope,
default
551: (20)                               )
552: (16)                           )
553: (8)                    self.xml.append(graph_element)
554: (4)                def add_graphs(self, graph_list):
555: (8)                    """Add many graphs to this GraphML document."""
556: (8)                    for G in graph_list:
557: (12)                       self.add_graph_element(G)
558: (4)                def dump(self, stream):
559: (8)                    from xml.etree.ElementTree import ElementTree
560: (8)                    if self.prettyprint:
561: (12)                       self.indent(self.xml)
562: (8)                    document = ElementTree(self.xml)
563: (8)                    document.write(stream, encoding=self.encoding, xml_declaration=True)
564: (4)                def indent(self, elem, level=0):
```

```
565: (8)                    i = "\n" + level * "  "
566: (8)                    if len(elem):
567: (12)                       if not elem.text or not elem.text.strip():
568: (16)                           elem.text = i + "  "
569: (12)                       if not elem.tail or not elem.tail.strip():
570: (16)                           elem.tail = i
571: (12)                       for elem in elem:
572: (16)                           self.indent(elem, level + 1)
573: (12)                       if not elem.tail or not elem.tail.strip():
574: (16)                           elem.tail = i
575: (8)                    else:
576: (12)                       if level and (not elem.tail or not elem.tail.strip()):
577: (16)                           elem.tail = i
578: (0)            class IncrementalElement:
579: (4)                """Wrapper for _IncrementalWriter providing an Element like interface.
580: (4)                This wrapper does not intend to be a complete implementation but rather to
581: (4)                deal with those calls used in GraphMLWriter.
582: (4)                """
583: (4)                def __init__(self, xml, prettyprint):
584: (8)                    self.xml = xml
585: (8)                    self.prettyprint = prettyprint
586: (4)                def append(self, element):
587: (8)                    self.xml.write(element, pretty_print=self.prettyprint)
588: (0)            class GraphMLWriterLxml(GraphMLWriter):
589: (4)                def __init__(
590: (8)                    self,
591: (8)                    path,
592: (8)                    graph=None,
593: (8)                    encoding="utf-8",
594: (8)                    prettyprint=True,
595: (8)                    infer_numeric_types=False,
596: (8)                    named_key_ids=False,
597: (8)                    edge_id_from_attribute=None,
598: (4)                ):
599: (8)                    self.construct_types()
600: (8)                    import lxml.etree as lxmletree
601: (8)                    self.myElement = lxmletree.Element
602: (8)                    self._encoding = encoding
603: (8)                    self._prettyprint = prettyprint
604: (8)                    self.named_key_ids = named_key_ids
605: (8)                    self.edge_id_from_attribute = edge_id_from_attribute
606: (8)                    self.infer_numeric_types = infer_numeric_types
607: (8)                    self._xml_base = lxmletree.xmlfile(path, encoding=encoding)
608: (8)                    self._xml = self._xml_base.__enter__()
609: (8)                    self._xml.write_declaration()
610: (8)                    self.xml = []
611: (8)                    self._keys = self.xml
612: (8)                    self._graphml = self._xml.element(
613: (12)                       "graphml",
614: (12)                       {
615: (16)                           "xmlns": self.NS_GRAPHML,
616: (16)                           "xmlns:xsi": self.NS_XSI,
617: (16)                           "xsi:schemaLocation": self.SCHEMALOCATION,
618: (12)                       },
619: (8)                    )
620: (8)                    self._graphml.__enter__()
621: (8)                    self.keys = {}
622: (8)                    self.attribute_types = defaultdict(set)
623: (8)                    if graph is not None:
624: (12)                       self.add_graph_element(graph)
625: (4)                def add_graph_element(self, G):
626: (8)                    """
627: (8)                    Serialize graph G in GraphML to the stream.
628: (8)                    """
629: (8)                    if G.is_directed():
630: (12)                       default_edge_type = "directed"
631: (8)                    else:
632: (12)                       default_edge_type = "undirected"
633: (8)                    graphid = G.graph.pop("id", None)
```

```
634: (8)                    if graphid is None:
635: (12)                       graph_element = self._xml.element("graph",
edgedefault=default_edge_type)
636: (8)                    else:
637: (12)                       graph_element = self._xml.element(
638: (16)                           "graph", edgedefault=default_edge_type, id=graphid
639: (12)                       )
640: (8)                    graphdata = {
641: (12)                       k: v
642: (12)                       for k, v in G.graph.items()
643: (12)                       if k not in ("node_default", "edge_default")
644: (8)                    }
645: (8)                    node_default = G.graph.get("node_default", {})
646: (8)                    edge_default = G.graph.get("edge_default", {})
647: (8)                    for k, v in graphdata.items():
648: (12)                       self.attribute_types[(str(k), "graph")].add(type(v))
649: (8)                    for k, v in graphdata.items():
650: (12)                       element_type = self.get_xml_type(self.attr_type(k, "graph", v))
651: (12)                       self.get_key(str(k), element_type, "graph", None)
652: (8)                    for node, d in G.nodes(data=True):
653: (12)                       for k, v in d.items():
654: (16)                           self.attribute_types[(str(k), "node")].add(type(v))
655: (8)                    for node, d in G.nodes(data=True):
656: (12)                       for k, v in d.items():
657: (16)                           T = self.get_xml_type(self.attr_type(k, "node", v))
658: (16)                           self.get_key(str(k), T, "node", node_default.get(k))
659: (8)                    if G.is_multigraph():
660: (12)                       for u, v, ekey, d in G.edges(keys=True, data=True):
661: (16)                           for k, v in d.items():
662: (20)                               self.attribute_types[(str(k), "edge")].add(type(v))
663: (12)                       for u, v, ekey, d in G.edges(keys=True, data=True):
664: (16)                           for k, v in d.items():
665: (20)                               T = self.get_xml_type(self.attr_type(k, "edge", v))
666: (20)                               self.get_key(str(k), T, "edge", edge_default.get(k))
667: (8)                    else:
668: (12)                       for u, v, d in G.edges(data=True):
669: (16)                           for k, v in d.items():
670: (20)                               self.attribute_types[(str(k), "edge")].add(type(v))
671: (12)                       for u, v, d in G.edges(data=True):
672: (16)                           for k, v in d.items():
673: (20)                               T = self.get_xml_type(self.attr_type(k, "edge", v))
674: (20)                               self.get_key(str(k), T, "edge", edge_default.get(k))
675: (8)                    for key in self.xml:
676: (12)                       self._xml.write(key, pretty_print=self._prettyprint)
677: (8)                    incremental_writer = IncrementalElement(self._xml, self._prettyprint)
678: (8)                    with graph_element:
679: (12)                       self.add_attributes("graph", incremental_writer, graphdata, {})
680: (12)                       self.add_nodes(G, incremental_writer)  # adds attributes too
681: (12)                       self.add_edges(G, incremental_writer)  # adds attributes too
682: (4)                def add_attributes(self, scope, xml_obj, data, default):
683: (8)                    """Appends attribute data."""
684: (8)                    for k, v in data.items():
685: (12)                       data_element = self.add_data(
686: (16)                           str(k), self.attr_type(str(k), scope, v), str(v), scope,
default.get(k)
687: (12)                       )
688: (12)                       xml_obj.append(data_element)
689: (4)                def __str__(self):
690: (8)                    return object.__str__(self)
691: (4)                def dump(self, stream=None):
692: (8)                    self._graphml.__exit__(None, None, None)
693: (8)                    self._xml_base.__exit__(None, None, None)
694: (0)            write_graphml = write_graphml_lxml
695: (0)            class GraphMLReader(GraphML):
696: (4)                """Read a GraphML document.  Produces NetworkX graph objects."""
697: (4)                def __init__(self, node_type=str, edge_key_type=int,
force_multigraph=False):
698: (8)                    self.construct_types()
699: (8)                    self.node_type = node_type
```

```
700: (8)                    self.edge_key_type = edge_key_type
701: (8)                    self.multigraph = force_multigraph  # If False, test for multiedges
702: (8)                    self.edge_ids = {}  # dict mapping (u,v) tuples to edge id attributes
703: (4)                def __call__(self, path=None, string=None):
704: (8)                    from xml.etree.ElementTree import ElementTree, fromstring
705: (8)                    if path is not None:
706: (12)                       self.xml = ElementTree(file=path)
707: (8)                    elif string is not None:
708: (12)                       self.xml = fromstring(string)
709: (8)                    else:
710: (12)                       raise ValueError("Must specify either 'path' or 'string' as
kwarg")
711: (8)                    (keys, defaults) = self.find_graphml_keys(self.xml)
712: (8)                    for g in self.xml.findall(f"{{{self.NS_GRAPHML}}}graph"):
713: (12)                       yield self.make_graph(g, keys, defaults)
714: (4)                def make_graph(self, graph_xml, graphml_keys, defaults, G=None):
715: (8)                    edgedefault = graph_xml.get("edgedefault", None)
716: (8)                    if G is None:
717: (12)                       if edgedefault == "directed":
718: (16)                           G = nx.MultiDiGraph()
719: (12)                       else:
720: (16)                           G = nx.MultiGraph()
721: (8)                    G.graph["node_default"] = {}
722: (8)                    G.graph["edge_default"] = {}
723: (8)                    for key_id, value in defaults.items():
724: (12)                       key_for = graphml_keys[key_id]["for"]
725: (12)                       name = graphml_keys[key_id]["name"]
726: (12)                       python_type = graphml_keys[key_id]["type"]
727: (12)                       if key_for == "node":
728: (16)                           G.graph["node_default"].update({name: python_type(value)})
729: (12)                       if key_for == "edge":
730: (16)                           G.graph["edge_default"].update({name: python_type(value)})
731: (8)                    hyperedge = graph_xml.find(f"{{{self.NS_GRAPHML}}}hyperedge")
732: (8)                    if hyperedge is not None:
733: (12)                       raise nx.NetworkXError("GraphML reader doesn't support
hyperedges")
734: (8)                    for node_xml in graph_xml.findall(f"{{{self.NS_GRAPHML}}}node"):
735: (12)                       self.add_node(G, node_xml, graphml_keys, defaults)
736: (8)                    for edge_xml in graph_xml.findall(f"{{{self.NS_GRAPHML}}}edge"):
737: (12)                       self.add_edge(G, edge_xml, graphml_keys)
738: (8)                    data = self.decode_data_elements(graphml_keys, graph_xml)
739: (8)                    G.graph.update(data)
740: (8)                    if self.multigraph:
741: (12)                       return G
742: (8)                    G = nx.DiGraph(G) if G.is_directed() else nx.Graph(G)
743: (8)                    nx.set_edge_attributes(G, values=self.edge_ids, name="id")
744: (8)                    return G
745: (4)                def add_node(self, G, node_xml, graphml_keys, defaults):
746: (8)                    """Add a node to the graph."""
747: (8)                    ports = node_xml.find(f"{{{self.NS_GRAPHML}}}port")
748: (8)                    if ports is not None:
749: (12)                       warnings.warn("GraphML port tag not supported.")
750: (8)                    node_id = self.node_type(node_xml.get("id"))
751: (8)                    data = self.decode_data_elements(graphml_keys, node_xml)
752: (8)                    G.add_node(node_id, **data)
753: (8)                    if node_xml.attrib.get("yfiles.foldertype") == "group":
754: (12)                       graph_xml = node_xml.find(f"{{{self.NS_GRAPHML}}}graph")
755: (12)                       self.make_graph(graph_xml, graphml_keys, defaults, G)
756: (4)                def add_edge(self, G, edge_element, graphml_keys):
757: (8)                    """Add an edge to the graph."""
758: (8)                    ports = edge_element.find(f"{{{self.NS_GRAPHML}}}port")
759: (8)                    if ports is not None:
760: (12)                       warnings.warn("GraphML port tag not supported.")
761: (8)                    directed = edge_element.get("directed")
762: (8)                    if G.is_directed() and directed == "false":
763: (12)                       msg = "directed=false edge found in directed graph."
764: (12)                       raise nx.NetworkXError(msg)
765: (8)                    if (not G.is_directed()) and directed == "true":
766: (12)                       msg = "directed=true edge found in undirected graph."
```

```
767: (12)                        raise nx.NetworkXError(msg)
768: (8)                    source = self.node_type(edge_element.get("source"))
769: (8)                    target = self.node_type(edge_element.get("target"))
770: (8)                    data = self.decode_data_elements(graphml_keys, edge_element)
771: (8)                    edge_id = edge_element.get("id")
772: (8)                    if edge_id:
773: (12)                       self.edge_ids[source, target] = edge_id
774: (12)                       try:
775: (16)                           edge_id = self.edge_key_type(edge_id)
776: (12)                       except ValueError:  # Could not convert.
777: (16)                           pass
778: (8)                    else:
779: (12)                       edge_id = data.get("key")
780: (8)                    if G.has_edge(source, target):
781: (12)                       self.multigraph = True
782: (8)                    G.add_edges_from([(source, target, edge_id, data)])
783: (4)                def decode_data_elements(self, graphml_keys, obj_xml):
784: (8)                    """Use the key information to decode the data XML if present."""
785: (8)                    data = {}
786: (8)                    for data_element in obj_xml.findall(f"{{{self.NS_GRAPHML}}}data"):
787: (12)                       key = data_element.get("key")
788: (12)                       try:
789: (16)                           data_name = graphml_keys[key]["name"]
790: (16)                           data_type = graphml_keys[key]["type"]
791: (12)                       except KeyError as err:
792: (16)                           raise nx.NetworkXError(f"Bad GraphML data: no key {key}") from
err
793: (12)                       text = data_element.text
794: (12)                       if text is not None and len(list(data_element)) == 0:
795: (16)                           if data_type == bool:
796: (20)                               data[data_name] = self.convert_bool[text.lower()]
797: (16)                           else:
798: (20)                               data[data_name] = data_type(text)
799: (12)                       elif len(list(data_element)) > 0:
800: (16)                           node_label = None
801: (16)                           gn = data_element.find(f"{{{self.NS_Y}}}GenericNode")
802: (16)                           if gn is not None:
803: (20)                               data["shape_type"] = gn.get("configuration")
804: (16)                           for node_type in ["GenericNode", "ShapeNode", "SVGNode",
"ImageNode"]:
805: (20)                               pref = f"{{{self.NS_Y}}}{node_type}/{{{self.NS_Y}}}"
806: (20)                               geometry = data_element.find(f"{pref}Geometry")
807: (20)                               if geometry is not None:
808: (24)                                   data["x"] = geometry.get("x")
809: (24)                                   data["y"] = geometry.get("y")
810: (20)                               if node_label is None:
811: (24)                                   node_label = data_element.find(f"{pref}NodeLabel")
812: (20)                               shape = data_element.find(f"{pref}Shape")
813: (20)                               if shape is not None:
814: (24)                                   data["shape_type"] = shape.get("type")
815: (16)                           if node_label is not None:
816: (20)                               data["label"] = node_label.text
817: (16)                           for edge_type in [
818: (20)                               "PolyLineEdge",
819: (20)                               "SplineEdge",
820: (20)                               "QuadCurveEdge",
821: (20)                               "BezierEdge",
822: (20)                               "ArcEdge",
823: (16)                           ]:
824: (20)                               pref = f"{{{self.NS_Y}}}{edge_type}/{{{self.NS_Y}}}"
825: (20)                               edge_label = data_element.find(f"{pref}EdgeLabel")
826: (20)                               if edge_label is not None:
827: (24)                                   break
828: (16)                           if edge_label is not None:
829: (20)                               data["label"] = edge_label.text
830: (12)                       elif text is None:
831: (16)                           data[data_name] = ""
832: (8)                    return data
833: (4)                def find_graphml_keys(self, graph_element):
```

```
834: (8)                    """Extracts all the keys and key defaults from the xml."""
835: (8)                    graphml_keys = {}
836: (8)                    graphml_key_defaults = {}
837: (8)                    for k in graph_element.findall(f"{{{self.NS_GRAPHML}}}key"):
838: (12)                       attr_id = k.get("id")
839: (12)                       attr_type = k.get("attr.type")
840: (12)                       attr_name = k.get("attr.name")
841: (12)                       yfiles_type = k.get("yfiles.type")
842: (12)                       if yfiles_type is not None:
843: (16)                           attr_name = yfiles_type
844: (16)                           attr_type = "yfiles"
845: (12)                       if attr_type is None:
846: (16)                           attr_type = "string"
847: (16)                           warnings.warn(f"No key type for id {attr_id}. Using string")
848: (12)                       if attr_name is None:
849: (16)                           raise nx.NetworkXError(f"Unknown key for id {attr_id}.")
850: (12)                       graphml_keys[attr_id] = {
851: (16)                           "name": attr_name,
852: (16)                           "type": self.python_type[attr_type],
853: (16)                           "for": k.get("for"),
854: (12)                       }
855: (12)                       default = k.find(f"{{{self.NS_GRAPHML}}}default")
856: (12)                       if default is not None:
857: (16)                           python_type = graphml_keys[attr_id]["type"]
858: (16)                           if python_type == bool:
859: (20)                               graphml_key_defaults[attr_id] = self.convert_bool[
860: (24)                                   default.text.lower()
861: (20)                               ]
862: (16)                           else:
863: (20)                               graphml_key_defaults[attr_id] = python_type(default.text)
864: (8)                    return graphml_keys, graphml_key_defaults
```

File 508 - leda.py:

```
1: (0)              """
2: (0)              Read graphs in LEDA format.
3: (0)              LEDA is a C++ class library for efficient data types and algorithms.
4: (0)              Format
5: (0)              ------
6: (0)              See http://www.algorithmic-
solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html
7: (0)              """
8: (0)              __all__ = ["read_leda", "parse_leda"]
9: (0)              import networkx as nx
10: (0)             from networkx.exception import NetworkXError
11: (0)             from networkx.utils import open_file
12: (0)             @open_file(0, mode="rb")
13: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
14: (0)             def read_leda(path, encoding="UTF-8"):
15: (4)                 """Read graph in LEDA format from path.
16: (4)                 Parameters
17: (4)                 ----------
18: (4)                 path : file or string
19: (7)                    File or filename to read.  Filenames ending in .gz or .bz2  will be
20: (7)                    uncompressed.
21: (4)                 Returns
22: (4)                 -------
23: (4)                 G : NetworkX graph
24: (4)                 Examples
25: (4)                 --------
26: (4)                 G=nx.read_leda('file.leda')
27: (4)                 References
28: (4)                 ----------
29: (4)                 .. [1] http://www.algorithmic-
solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html
30: (4)                 """
31: (4)                 lines = (line.decode(encoding) for line in path)
```

```
32: (4)                 G = parse_leda(lines)
33: (4)                 return G
34: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
35: (0)             def parse_leda(lines):
36: (4)                 """Read graph in LEDA format from string or iterable.
37: (4)                 Parameters
38: (4)                 ----------
39: (4)                 lines : string or iterable
40: (7)                    Data in LEDA format.
41: (4)                 Returns
42: (4)                 -------
43: (4)                 G : NetworkX graph
44: (4)                 Examples
45: (4)                 --------
46: (4)                 G=nx.parse_leda(string)
47: (4)                 References
48: (4)                 ----------
49: (4)                 .. [1] http://www.algorithmic-
solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html
50: (4)                 """
51: (4)                 if isinstance(lines, str):
52: (8)                     lines = iter(lines.split("\n"))
53: (4)                 lines = iter(
54: (8)                     [
55: (12)                        line.rstrip("\n")
56: (12)                        for line in lines
57: (12)                        if not (line.startswith(("#", "\n")) or line == "")
58: (8)                     ]
59: (4)                 )
60: (4)                 for i in range(3):
61: (8)                     next(lines)
62: (4)                 du = int(next(lines))  # -1=directed, -2=undirected
63: (4)                 if du == -1:
64: (8)                     G = nx.DiGraph()
65: (4)                 else:
66: (8)                     G = nx.Graph()
67: (4)                 n = int(next(lines))  # number of nodes
68: (4)                 node = {}
69: (4)                 for i in range(1, n + 1):  # LEDA counts from 1 to n
70: (8)                     symbol = next(lines).rstrip().strip("|{}|  ")
71: (8)                     if symbol == "":
72: (12)                        symbol = str(i)  # use int if no label - could be trouble
73: (8)                     node[i] = symbol
74: (4)                 G.add_nodes_from([s for i, s in node.items()])
75: (4)                 m = int(next(lines))  # number of edges
76: (4)                 for i in range(m):
77: (8)                     try:
78: (12)                        s, t, reversal, label = next(lines).split()
79: (8)                     except BaseException as err:
80: (12)                        raise NetworkXError(f"Too few fields in LEDA.GRAPH edge {i+1}")
from err
81: (8)                     G.add_edge(node[int(s)], node[int(t)], label=label[2:-2])
82: (4)                 return G
```

File 509 - multiline_adjlist.py:

```
1: (0)              """
2: (0)              *************************
3: (0)              Multi-line Adjacency List
4: (0)              *************************
5: (0)              Read and write NetworkX graphs as multi-line adjacency lists.
6: (0)              The multi-line adjacency list format is useful for graphs with
7: (0)              nodes that can be meaningfully represented as strings.  With this format
8: (0)              simple edge data can be stored but node or graph data is not.
9: (0)              Format
10: (0)             ------
11: (0)             The first label in a line is the source node label followed by the node degree
```

```
 12: (0)             d.  The next d lines are target node labels and optional edge data.
 13: (0)             That pattern repeats for all nodes in the graph.
 14: (0)             The graph with edges a-b, a-c, d-e can be represented as the following
 15: (0)             adjacency list (anything following the # in a line is a comment)::
 16: (5)                  a 2
 17: (5)                  b
 18: (5)                  c
 19: (5)                  d 1
 20: (5)                  e
 21: (0)             """
 22: (0)             __all__ = [
 23: (4)                 "generate_multiline_adjlist",
 24: (4)                 "write_multiline_adjlist",
 25: (4)                 "parse_multiline_adjlist",
 26: (4)                 "read_multiline_adjlist",
 27: (0)             ]
 28: (0)             import networkx as nx
 29: (0)             from networkx.utils import open_file
 30: (0)             def generate_multiline_adjlist(G, delimiter=" "):
 31: (4)                 """Generate a single line of the graph G in multiline adjacency list
format.
 32: (4)                 Parameters
 33: (4)                 ----------
 34: (4)                 G : NetworkX graph
 35: (4)                 delimiter : string, optional
 36: (7)                    Separator for node labels
 37: (4)                 Returns
 38: (4)                 -------
 39: (4)                 lines : string
 40: (8)                     Lines of data in multiline adjlist format.
 41: (4)                 Examples
 42: (4)                 --------
 43: (4)                 >>> G = nx.lollipop_graph(4, 3)
 44: (4)                 >>> for line in nx.generate_multiline_adjlist(G):
 45: (4)                 ...     print(line)
 46: (4)                 0 3
 47: (4)                 1 {}
 48: (4)                 2 {}
 49: (4)                 3 {}
 50: (4)                 1 2
 51: (4)                 2 {}
 52: (4)                 3 {}
 53: (4)                 2 1
 54: (4)                 3 {}
 55: (4)                 3 1
 56: (4)                 4 {}
 57: (4)                 4 1
 58: (4)                 5 {}
 59: (4)                 5 1
 60: (4)                 6 {}
 61: (4)                 6 0
 62: (4)                 See Also
 63: (4)                 --------
 64: (4)                 write_multiline_adjlist, read_multiline_adjlist
 65: (4)                 """
 66: (4)                 if G.is_directed():
 67: (8)                     if G.is_multigraph():
 68: (12)                        for s, nbrs in G.adjacency():
 69: (16)                            nbr_edges = [
 70: (20)                                (u, data)
 71: (20)                                for u, datadict in nbrs.items()
 72: (20)                                for key, data in datadict.items()
 73: (16)                            ]
 74: (16)                            deg = len(nbr_edges)
 75: (16)                            yield str(s) + delimiter + str(deg)
 76: (16)                            for u, d in nbr_edges:
 77: (20)                                if d is None:
 78: (24)                                    yield str(u)
 79: (20)                                else:
```

```
80: (24)                                    yield str(u) + delimiter + str(d)
81: (8)                     else:  # directed single edges
82: (12)                        for s, nbrs in G.adjacency():
83: (16)                            deg = len(nbrs)
84: (16)                            yield str(s) + delimiter + str(deg)
85: (16)                            for u, d in nbrs.items():
86: (20)                                if d is None:
87: (24)                                    yield str(u)
88: (20)                                else:
89: (24)                                    yield str(u) + delimiter + str(d)
90: (4)                 else:  # undirected
91: (8)                     if G.is_multigraph():
92: (12)                        seen = set()  # helper dict used to avoid duplicate edges
93: (12)                        for s, nbrs in G.adjacency():
94: (16)                            nbr_edges = [
95: (20)                                (u, data)
96: (20)                                for u, datadict in nbrs.items()
97: (20)                                if u not in seen
98: (20)                                for key, data in datadict.items()
99: (16)                            ]
100: (16)                           deg = len(nbr_edges)
101: (16)                           yield str(s) + delimiter + str(deg)
102: (16)                           for u, d in nbr_edges:
103: (20)                               if d is None:
104: (24)                                   yield str(u)
105: (20)                               else:
106: (24)                                   yield str(u) + delimiter + str(d)
107: (16)                           seen.add(s)
108: (8)                    else:  # undirected single edges
109: (12)                       seen = set()  # helper dict used to avoid duplicate edges
110: (12)                       for s, nbrs in G.adjacency():
111: (16)                           nbr_edges = [(u, d) for u, d in nbrs.items() if u not in seen]
112: (16)                           deg = len(nbr_edges)
113: (16)                           yield str(s) + delimiter + str(deg)
114: (16)                           for u, d in nbr_edges:
115: (20)                               if d is None:
116: (24)                                   yield str(u)
117: (20)                               else:
118: (24)                                   yield str(u) + delimiter + str(d)
119: (16)                           seen.add(s)
120: (0)            @open_file(1, mode="wb")
121: (0)            def write_multiline_adjlist(G, path, delimiter=" ", comments="#",
encoding="utf-8"):
122: (4)                """Write the graph G in multiline adjacency list format to path
123: (4)                Parameters
124: (4)                ----------
125: (4)                G : NetworkX graph
126: (4)                path : string or file
127: (7)                   Filename or file handle to write to.
128: (7)                   Filenames ending in .gz or .bz2 will be compressed.
129: (4)                comments : string, optional
130: (7)                   Marker for comment lines
131: (4)                delimiter : string, optional
132: (7)                   Separator for node labels
133: (4)                encoding : string, optional
134: (7)                   Text encoding.
135: (4)                Examples
136: (4)                --------
137: (4)                >>> G = nx.path_graph(4)
138: (4)                >>> nx.write_multiline_adjlist(G, "test.adjlist")
139: (4)                The path can be a file handle or a string with the name of the file. If a
140: (4)                file handle is provided, it has to be opened in 'wb' mode.
141: (4)                >>> fh = open("test.adjlist", "wb")
142: (4)                >>> nx.write_multiline_adjlist(G, fh)
143: (4)                Filenames ending in .gz or .bz2 will be compressed.
144: (4)                >>> nx.write_multiline_adjlist(G, "test.adjlist.gz")
145: (4)                See Also
146: (4)                --------
147: (4)                read_multiline_adjlist
```

```
(4)                 """
(4)                 import sys
(4)                 import time
(4)                 pargs = comments + " ".join(sys.argv)
(4)                 header = (
(8)                     f"{pargs}\n"
(8)                     + comments
(8)                     + f" GMT {time.asctime(time.gmtime())}\n"
(8)                     + comments
(8)                     + f" {G.name}\n"
(4)                 )
(4)                 path.write(header.encode(encoding))
(4)                 for multiline in generate_multiline_adjlist(G, delimiter):
(8)                     multiline += "\n"
(8)                     path.write(multiline.encode(encoding))
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def parse_multiline_adjlist(
(4)                 lines, comments="#", delimiter=None, create_using=None, nodetype=None,
edgetype=None
(0)             ):
(4)                 """Parse lines of a multiline adjacency list representation of a graph.
(4)                 Parameters
(4)                 ----------
(4)                 lines : list or iterator of strings
(8)                     Input data in multiline adjlist format
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 nodetype : Python type, optional
(7)                    Convert nodes to this type.
(4)                 edgetype : Python type, optional
(7)                    Convert edges to this type.
(4)                 comments : string, optional
(7)                    Marker for comment lines
(4)                 delimiter : string, optional
(7)                    Separator for node labels.  The default is whitespace.
(4)                 Returns
(4)                 -------
(4)                 G: NetworkX graph
(8)                     The graph corresponding to the lines in multiline adjacency list
format.
(4)                 Examples
(4)                 --------
(4)                 >>> lines = [
(4)                 ...     "1 2",
(4)                 ...     "2 {'weight':3, 'name': 'Frodo'}",
(4)                 ...     "3 {}",
(4)                 ...     "2 1",
(4)                 ...     "5 {'weight':6, 'name': 'Saruman'}",
(4)                 ... ]
(4)                 >>> G = nx.parse_multiline_adjlist(iter(lines), nodetype=int)
(4)                 >>> list(G)
(4)                 [1, 2, 3, 5]
(4)                 """
(4)                 from ast import literal_eval
(4)                 G = nx.empty_graph(0, create_using)
(4)                 for line in lines:
(8)                     p = line.find(comments)
(8)                     if p >= 0:
(12)                        line = line[:p]
(8)                     if not line:
(12)                        continue
(8)                     try:
(12)                        (u, deg) = line.strip().split(delimiter)
(12)                        deg = int(deg)
(8)                     except BaseException as err:
(12)                        raise TypeError(f"Failed to read node and degree on line
({line})") from err
(8)                     if nodetype is not None:
(12)                        try:
```

```
(16)                            u = nodetype(u)
(12)                        except BaseException as err:
(16)                            raise TypeError(
(20)                                f"Failed to convert node ({u}) to type {nodetype}"
(16)                            ) from err
(8)                     G.add_node(u)
(8)                     for i in range(deg):
(12)                        while True:
(16)                            try:
(20)                                line = next(lines)
(16)                            except StopIteration as err:
(20)                                msg = f"Failed to find neighbor for node ({u})"
(20)                                raise TypeError(msg) from err
(16)                            p = line.find(comments)
(16)                            if p >= 0:
(20)                                line = line[:p]
(16)                            if line:
(20)                                break
(12)                        vlist = line.strip().split(delimiter)
(12)                        numb = len(vlist)
(12)                        if numb < 1:
(16)                            continue  # isolated node
(12)                        v = vlist.pop(0)
(12)                        data = "".join(vlist)
(12)                        if nodetype is not None:
(16)                            try:
(20)                                v = nodetype(v)
(16)                            except BaseException as err:
(20)                                raise TypeError(
(24)                                    f"Failed to convert node ({v}) to type {nodetype}"
(20)                                ) from err
(12)                        if edgetype is not None:
(16)                            try:
(20)                                edgedata = {"weight": edgetype(data)}
(16)                            except BaseException as err:
(20)                                raise TypeError(
(24)                                    f"Failed to convert edge data ({data}) to type
{edgetype}"
(20)                                ) from err
(12)                        else:
(16)                            try:  # try to evaluate
(20)                                edgedata = literal_eval(data)
(16)                            except:
(20)                                edgedata = {}
(12)                        G.add_edge(u, v, **edgedata)
(4)                 return G
(0)             @open_file(0, mode="rb")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def read_multiline_adjlist(
(4)                 path,
(4)                 comments="#",
(4)                 delimiter=None,
(4)                 create_using=None,
(4)                 nodetype=None,
(4)                 edgetype=None,
(4)                 encoding="utf-8",
(0)             ):
(4)                 """Read graph in multi-line adjacency list format from path.
(4)                 Parameters
(4)                 ----------
(4)                 path : string or file
(7)                    Filename or file handle to read.
(7)                    Filenames ending in .gz or .bz2 will be uncompressed.
(4)                 create_using : NetworkX graph constructor, optional (default=nx.Graph)
(7)                    Graph type to create. If graph instance, then cleared before populated.
(4)                 nodetype : Python type, optional
(7)                    Convert nodes to this type.
(4)                 edgetype : Python type, optional
(7)                    Convert edge data to this type.
```

```
282: (4)                comments : string, optional
283: (7)                   Marker for comment lines
284: (4)                delimiter : string, optional
285: (7)                   Separator for node labels.  The default is whitespace.
286: (4)                Returns
287: (4)                -------
288: (4)                G: NetworkX graph
289: (4)                Examples
290: (4)                --------
291: (4)                >>> G = nx.path_graph(4)
292: (4)                >>> nx.write_multiline_adjlist(G, "test.adjlist")
293: (4)                >>> G = nx.read_multiline_adjlist("test.adjlist")
294: (4)                The path can be a file or a string with the name of the file. If a
295: (4)                file s provided, it has to be opened in 'rb' mode.
296: (4)                >>> fh = open("test.adjlist", "rb")
297: (4)                >>> G = nx.read_multiline_adjlist(fh)
298: (4)                Filenames ending in .gz or .bz2 will be compressed.
299: (4)                >>> nx.write_multiline_adjlist(G, "test.adjlist.gz")
300: (4)                >>> G = nx.read_multiline_adjlist("test.adjlist.gz")
301: (4)                The optional nodetype is a function to convert node strings to nodetype.
302: (4)                For example
303: (4)                >>> G = nx.read_multiline_adjlist("test.adjlist", nodetype=int)
304: (4)                will attempt to convert all nodes to integer type.
305: (4)                The optional edgetype is a function to convert edge data strings to
306: (4)                edgetype.
307: (4)                >>> G = nx.read_multiline_adjlist("test.adjlist")
308: (4)                The optional create_using parameter is a NetworkX graph container.
309: (4)                The default is Graph(), an undirected graph.  To read the data as
310: (4)                a directed graph use
311: (4)                >>> G = nx.read_multiline_adjlist("test.adjlist", create_using=nx.DiGraph)
312: (4)                Notes
313: (4)                -----
314: (4)                This format does not store graph, node, or edge data.
315: (4)                See Also
316: (4)                --------
317: (4)                write_multiline_adjlist
318: (4)                """
319: (4)                lines = (line.decode(encoding) for line in path)
320: (4)                return parse_multiline_adjlist(
321: (8)                    lines,
322: (8)                    comments=comments,
323: (8)                    delimiter=delimiter,
324: (8)                    create_using=create_using,
325: (8)                    nodetype=nodetype,
326: (8)                    edgetype=edgetype,
327: (4)                )

----------------------------------------

File 510 - p2g.py:

1: (0)              """
2: (0)              This module provides the following: read and write of p2g format
3: (0)              used in metabolic pathway studies.
4: (0)              See
https://web.archive.org/web/20080626113807/http://www.cs.purdue.edu/homes/koyuturk/pathway/ for a
description.
5: (0)              The summary is included here:
6: (0)              A file that describes a uniquely labeled graph (with extension ".gr")
7: (0)              format looks like the following:
8: (0)              name
9: (0)              3 4
10: (0)             a
11: (0)             1 2
12: (0)             b
13: (0)             c
14: (0)             0 2
15: (0)             "name" is simply a description of what the graph corresponds to. The
16: (0)             second line displays the number of nodes and number of edges,
```

```
(0)             respectively. This sample graph contains three nodes labeled "a", "b",
(0)             and "c". The rest of the graph contains two lines for each node. The
(0)             first line for a node contains the node label. After the declaration
(0)             of the node label, the out-edges of that node in the graph are
(0)             provided. For instance, "a" is linked to nodes 1 and 2, which are
(0)             labeled "b" and "c", while the node labeled "b" has no outgoing
(0)             edges. Observe that node labeled "c" has an outgoing edge to
(0)             itself. Indeed, self-loops are allowed. Node index starts from 0.
(0)             """
(0)             import networkx as nx
(0)             from networkx.utils import open_file
(0)             @open_file(1, mode="w")
(0)             def write_p2g(G, path, encoding="utf-8"):
(4)                 """Write NetworkX graph in p2g format.
(4)                 Notes
(4)                 -----
(4)                 This format is meant to be used with directed graphs with
(4)                 possible self loops.
(4)                 """
(4)                 path.write((f"{G.name}\n").encode(encoding))
(4)                 path.write((f"{G.order()} {G.size()}\n").encode(encoding))
(4)                 nodes = list(G)
(4)                 nodenumber = dict(zip(nodes, range(len(nodes))))
(4)                 for n in nodes:
(8)                     path.write((f"{n}\n").encode(encoding))
(8)                     for nbr in G.neighbors(n):
(12)                        path.write((f"{nodenumber[nbr]} ").encode(encoding))
(8)                     path.write("\n".encode(encoding))
(0)             @open_file(0, mode="r")
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def read_p2g(path, encoding="utf-8"):
(4)                 """Read graph in p2g format from path.
(4)                 Returns
(4)                 -------
(4)                 MultiDiGraph
(4)                 Notes
(4)                 -----
(4)                 If you want a DiGraph (with no self loops allowed and no edge data)
(4)                 use D=nx.DiGraph(read_p2g(path))
(4)                 """
(4)                 lines = (line.decode(encoding) for line in path)
(4)                 G = parse_p2g(lines)
(4)                 return G
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def parse_p2g(lines):
(4)                 """Parse p2g format graph from string or iterable.
(4)                 Returns
(4)                 -------
(4)                 MultiDiGraph
(4)                 """
(4)                 description = next(lines).strip()
(4)                 G = nx.MultiDiGraph(name=description, selfloops=True)
(4)                 nnodes, nedges = map(int, next(lines).split())
(4)                 nodelabel = {}
(4)                 nbrs = {}
(4)                 for i in range(nnodes):
(8)                     n = next(lines).strip()
(8)                     nodelabel[i] = n
(8)                     G.add_node(n)
(8)                     nbrs[n] = map(int, next(lines).split())
(4)                 for n in G:
(8)                     for nbr in nbrs[n]:
(12)                        G.add_edge(n, nodelabel[nbr])
(4)                 return G

----------------------------------------

File 511 - pajek.py:
```

```
1: (0)              """
2: (0)              *****
3: (0)              Pajek
4: (0)              *****
5: (0)              Read graphs in Pajek format.
6: (0)              This implementation handles directed and undirected graphs including
7: (0)              those with self loops and parallel edges.
8: (0)              Format
9: (0)              ------
10: (0)             See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm
11: (0)             for format information.
12: (0)             """
13: (0)             import warnings
14: (0)             import networkx as nx
15: (0)             from networkx.utils import open_file
16: (0)             __all__ = ["read_pajek", "parse_pajek", "generate_pajek", "write_pajek"]
17: (0)             def generate_pajek(G):
18: (4)                 """Generate lines in Pajek graph format.
19: (4)                 Parameters
20: (4)                 ----------
21: (4)                 G : graph
22: (7)                    A Networkx graph
23: (4)                 References
24: (4)                 ----------
25: (4)                 See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm
26: (4)                 for format information.
27: (4)                 """
28: (4)                 if G.name == "":
29: (8)                     name = "NetworkX"
30: (4)                 else:
31: (8)                     name = G.name
32: (4)                 yield f"*vertices {G.order()}"
33: (4)                 nodes = list(G)
34: (4)                 nodenumber = dict(zip(nodes, range(1, len(nodes) + 1)))
35: (4)                 for n in nodes:
36: (8)                     na = G.nodes.get(n, {}).copy()
37: (8)                     x = na.pop("x", 0.0)
38: (8)                     y = na.pop("y", 0.0)
39: (8)                     try:
40: (12)                        id = int(na.pop("id", nodenumber[n]))
41: (8)                     except ValueError as err:
42: (12)                        err.args += (
43: (16)                            (
44: (20)                                "Pajek format requires 'id' to be an int()."
45: (20)                                " Refer to the 'Relabeling nodes' section."
46: (16)                            ),
47: (12)                        )
48: (12)                        raise
49: (8)                     nodenumber[n] = id
50: (8)                     shape = na.pop("shape", "ellipse")
51: (8)                     s = " ".join(map(make_qstr, (id, n, x, y, shape)))
52: (8)                     for k, v in na.items():
53: (12)                        if isinstance(v, str) and v.strip() != "":
54: (16)                            s += f" {make_qstr(k)} {make_qstr(v)}"
55: (12)                        else:
56: (16)                            warnings.warn(
57: (20)                                f"Node attribute {k} is not processed. {('Empty attribute'
if isinstance(v, str) else 'Non-string attribute')}."
58: (16)                            )
59: (8)                     yield s
60: (4)                 if G.is_directed():
61: (8)                     yield "*arcs"
62: (4)                 else:
63: (8)                     yield "*edges"
64: (4)                 for u, v, edgedata in G.edges(data=True):
65: (8)                     d = edgedata.copy()
66: (8)                     value = d.pop("weight", 1.0)  # use 1 as default edge value
67: (8)                     s = " ".join(map(make_qstr, (nodenumber[u], nodenumber[v], value)))
68: (8)                     for k, v in d.items():
```

```
 69: (12)                        if isinstance(v, str) and v.strip() != "":
 70: (16)                            s += f" {make_qstr(k)} {make_qstr(v)}"
 71: (12)                        else:
 72: (16)                            warnings.warn(
 73: (20)                                f"Edge attribute {k} is not processed. {('Empty attribute'
 if isinstance(v, str) else 'Non-string attribute')}."
 74: (16)                            )
 75: (8)                     yield s
 76: (0)             @open_file(1, mode="wb")
 77: (0)             def write_pajek(G, path, encoding="UTF-8"):
 78: (4)                 """Write graph in Pajek format to path.
 79: (4)                 Parameters
 80: (4)                 ----------
 81: (4)                 G : graph
 82: (7)                    A Networkx graph
 83: (4)                 path : file or string
 84: (7)                    File or filename to write.
 85: (7)                    Filenames ending in .gz or .bz2 will be compressed.
 86: (4)                 Examples
 87: (4)                 --------
 88: (4)                 >>> G = nx.path_graph(4)
 89: (4)                 >>> nx.write_pajek(G, "test.net")
 90: (4)                 Warnings
 91: (4)                 --------
 92: (4)                 Optional node attributes and edge attributes must be non-empty strings.
 93: (4)                 Otherwise it will not be written into the file. You will need to
 94: (4)                 convert those attributes to strings if you want to keep them.
 95: (4)                 References
 96: (4)                 ----------
 97: (4)                 See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm
 98: (4)                 for format information.
 99: (4)                 """
 100: (4)                for line in generate_pajek(G):
 101: (8)                    line += "\n"
 102: (8)                    path.write(line.encode(encoding))
 103: (0)            @open_file(0, mode="rb")
 104: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
 105: (0)            def read_pajek(path, encoding="UTF-8"):
 106: (4)                """Read graph in Pajek format from path.
 107: (4)                Parameters
 108: (4)                ----------
 109: (4)                path : file or string
 110: (7)                   File or filename to write.
 111: (7)                   Filenames ending in .gz or .bz2 will be uncompressed.
 112: (4)                Returns
 113: (4)                -------
 114: (4)                G : NetworkX MultiGraph or MultiDiGraph.
 115: (4)                Examples
 116: (4)                --------
 117: (4)                >>> G = nx.path_graph(4)
 118: (4)                >>> nx.write_pajek(G, "test.net")
 119: (4)                >>> G = nx.read_pajek("test.net")
 120: (4)                To create a Graph instead of a MultiGraph use
 121: (4)                >>> G1 = nx.Graph(G)
 122: (4)                References
 123: (4)                ----------
 124: (4)                See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm
 125: (4)                for format information.
 126: (4)                """
 127: (4)                lines = (line.decode(encoding) for line in path)
 128: (4)                return parse_pajek(lines)
 129: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
 130: (0)            def parse_pajek(lines):
 131: (4)                """Parse Pajek format graph from string or iterable.
 132: (4)                Parameters
 133: (4)                ----------
 134: (4)                lines : string or iterable
 135: (7)                   Data in Pajek format.
 136: (4)                Returns
```

```
(4)                 -------
(4)                 G : NetworkX graph
(4)                 See Also
(4)                 --------
(4)                 read_pajek
(4)                 """
(4)                 import shlex
(4)                 if isinstance(lines, str):
(8)                     lines = iter(lines.split("\n"))
(4)                 lines = iter([line.rstrip("\n") for line in lines])
(4)                 G = nx.MultiDiGraph()  # are multiedges allowed in Pajek? assume yes
(4)                 labels = []  # in the order of the file, needed for matrix
(4)                 while lines:
(8)                     try:
(12)                        l = next(lines)
(8)                     except:  # EOF
(12)                        break
(8)                     if l.lower().startswith("*network"):
(12)                        try:
(16)                            label, name = l.split(None, 1)
(12)                        except ValueError:
(16)                            pass
(12)                        else:
(16)                            G.graph["name"] = name
(8)                     elif l.lower().startswith("*vertices"):
(12)                        nodelabels = {}
(12)                        l, nnodes = l.split()
(12)                        for i in range(int(nnodes)):
(16)                            l = next(lines)
(16)                            try:
(20)                                splitline = [
(24)                                    x.decode("utf-8") for x in
shlex.split(str(l).encode("utf-8"))
(20)                                ]
(16)                            except AttributeError:
(20)                                splitline = shlex.split(str(l))
(16)                            id, label = splitline[0:2]
(16)                            labels.append(label)
(16)                            G.add_node(label)
(16)                            nodelabels[id] = label
(16)                            G.nodes[label]["id"] = id
(16)                            try:
(20)                                x, y, shape = splitline[2:5]
(20)                                G.nodes[label].update(
(24)                                    {"x": float(x), "y": float(y), "shape": shape}
(20)                                )
(16)                            except:
(20)                                pass
(16)                            extra_attr = zip(splitline[5::2], splitline[6::2])
(16)                            G.nodes[label].update(extra_attr)
(8)                     elif l.lower().startswith("*edges") or l.lower().startswith("*arcs"):
(12)                        if l.lower().startswith("*edge"):
(16)                            G = nx.MultiGraph(G)
(12)                        if l.lower().startswith("*arcs"):
(16)                            G = G.to_directed()
(12)                        for l in lines:
(16)                            try:
(20)                                splitline = [
(24)                                    x.decode("utf-8") for x in
shlex.split(str(l).encode("utf-8"))
(20)                                ]
(16)                            except AttributeError:
(20)                                splitline = shlex.split(str(l))
(16)                            if len(splitline) < 2:
(20)                                continue
(16)                            ui, vi = splitline[0:2]
(16)                            u = nodelabels.get(ui, ui)
(16)                            v = nodelabels.get(vi, vi)
(16)                            edge_data = {}
```

```
204: (16)                            try:
205: (20)                                w = splitline[2:3]
206: (20)                                edge_data.update({"weight": float(w[0])})
207: (16)                            except:
208: (20)                                pass
209: (16)                            extra_attr = zip(splitline[3::2], splitline[4::2])
210: (16)                            edge_data.update(extra_attr)
211: (16)                            G.add_edge(u, v, **edge_data)
212: (8)                    elif l.lower().startswith("*matrix"):
213: (12)                       G = nx.DiGraph(G)
214: (12)                       adj_list = (
215: (16)                           (labels[row], labels[col], {"weight": int(data)})
216: (16)                           for (row, line) in enumerate(lines)
217: (16)                           for (col, data) in enumerate(line.split())
218: (16)                           if int(data) != 0
219: (12)                       )
220: (12)                       G.add_edges_from(adj_list)
221: (4)                return G
222: (0)            def make_qstr(t):
223: (4)                """Returns the string representation of t.
224: (4)                Add outer double-quotes if the string has a space.
225: (4)                """
226: (4)                if not isinstance(t, str):
227: (8)                    t = str(t)
228: (4)                if " " in t:
229: (8)                    t = f'"{t}"'
230: (4)                return t
```

File 512 - sparse6.py:

```
1: (0)              """Functions for reading and writing graphs in the *sparse6* format.
2: (0)              The *sparse6* file format is a space-efficient format for large sparse
3: (0)              graphs. For small graphs or large dense graphs, use the *graph6* file
4: (0)              format.
5: (0)              For more information, see the `sparse6`_ homepage.
6: (0)              .. _sparse6: https://users.cecs.anu.edu.au/~bdm/data/formats.html
7: (0)              """
8: (0)              import networkx as nx
9: (0)              from networkx.exception import NetworkXError
10: (0)             from networkx.readwrite.graph6 import data_to_n, n_to_data
11: (0)             from networkx.utils import not_implemented_for, open_file
12: (0)             __all__ = ["from_sparse6_bytes", "read_sparse6", "to_sparse6_bytes",
"write_sparse6"]
13: (0)             def _generate_sparse6_bytes(G, nodes, header):
14: (4)                 """Yield bytes in the sparse6 encoding of a graph.
15: (4)                 `G` is an undirected simple graph. `nodes` is the list of nodes for
16: (4)                 which the node-induced subgraph will be encoded; if `nodes` is the
17: (4)                 list of all nodes in the graph, the entire graph will be
18: (4)                 encoded. `header` is a Boolean that specifies whether to generate
19: (4)                 the header ``b'>>sparse6<<'`` before the remaining data.
20: (4)                 This function generates `bytes` objects in the following order:
21: (4)                 1. the header (if requested),
22: (4)                 2. the encoding of the number of nodes,
23: (4)                 3. each character, one-at-a-time, in the encoding of the requested
24: (7)                    node-induced subgraph,
25: (4)                 4. a newline character.
26: (4)                 This function raises :exc:`ValueError` if the graph is too large for
27: (4)                 the graph6 format (that is, greater than ``2 ** 36`` nodes).
28: (4)                 """
29: (4)                 n = len(G)
30: (4)                 if n >= 2**36:
31: (8)                     raise ValueError(
32: (12)                        "sparse6 is only defined if number of nodes is less than 2 ** 36"
33: (8)                     )
34: (4)                 if header:
35: (8)                     yield b">>sparse6<<"
36: (4)                 yield b":"
```

```
 37: (4)                 for d in n_to_data(n):
 38: (8)                     yield str.encode(chr(d + 63))
 39: (4)                 k = 1
 40: (4)                 while 1 << k < n:
 41: (8)                     k += 1
 42: (4)                 def enc(x):
 43: (8)                     """Big endian k-bit encoding of x"""
 44: (8)                     return [1 if (x & 1 << (k - 1 - i)) else 0 for i in range(k)]
 45: (4)                 edges = sorted((max(u, v), min(u, v)) for u, v in G.edges())
 46: (4)                 bits = []
 47: (4)                 curv = 0
 48: (4)                 for v, u in edges:
 49: (8)                     if v == curv:  # current vertex edge
 50: (12)                        bits.append(0)
 51: (12)                        bits.extend(enc(u))
 52: (8)                     elif v == curv + 1:  # next vertex edge
 53: (12)                        curv += 1
 54: (12)                        bits.append(1)
 55: (12)                        bits.extend(enc(u))
 56: (8)                     else:  # skip to vertex v and then add edge to u
 57: (12)                        curv = v
 58: (12)                        bits.append(1)
 59: (12)                        bits.extend(enc(v))
 60: (12)                        bits.append(0)
 61: (12)                        bits.extend(enc(u))
 62: (4)                 if k < 6 and n == (1 << k) and ((-len(bits)) % 6) >= k and curv < (n - 1):
 63: (8)                     bits.append(0)
 64: (8)                     bits.extend([1] * ((-len(bits)) % 6))
 65: (4)                 else:
 66: (8)                     bits.extend([1] * ((-len(bits)) % 6))
 67: (4)                 data = [
 68: (8)                     (bits[i + 0] << 5)
 69: (8)                     + (bits[i + 1] << 4)
 70: (8)                     + (bits[i + 2] << 3)
 71: (8)                     + (bits[i + 3] << 2)
 72: (8)                     + (bits[i + 4] << 1)
 73: (8)                     + (bits[i + 5] << 0)
 74: (8)                     for i in range(0, len(bits), 6)
 75: (4)                 ]
 76: (4)                 for d in data:
 77: (8)                     yield str.encode(chr(d + 63))
 78: (4)                 yield b"\n"
 79: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
 80: (0)             def from_sparse6_bytes(string):
 81: (4)                 """Read an undirected graph in sparse6 format from string.
 82: (4)                 Parameters
 83: (4)                 ----------
 84: (4)                 string : string
 85: (7)                    Data in sparse6 format
 86: (4)                 Returns
 87: (4)                 -------
 88: (4)                 G : Graph
 89: (4)                 Raises
 90: (4)                 ------
 91: (4)                 NetworkXError
 92: (8)                     If the string is unable to be parsed in sparse6 format
 93: (4)                 Examples
 94: (4)                 --------
 95: (4)                 >>> G = nx.from_sparse6_bytes(b":A_")
 96: (4)                 >>> sorted(G.edges())
 97: (4)                 [(0, 1), (0, 1), (0, 1)]
 98: (4)                 See Also
 99: (4)                 --------
100: (4)                 read_sparse6, write_sparse6
101: (4)                 References
102: (4)                 ----------
103: (4)                 .. [1] Sparse6 specification
104: (11)                       <https://users.cecs.anu.edu.au/~bdm/data/formats.html>
105: (4)                 """
```

```
(4)                 if string.startswith(b">>sparse6<<"):
(8)                     string = string[11:]
(4)                 if not string.startswith(b":"):
(8)                     raise NetworkXError("Expected leading colon in sparse6")
(4)                 chars = [c - 63 for c in string[1:]]
(4)                 n, data = data_to_n(chars)
(4)                 k = 1
(4)                 while 1 << k < n:
(8)                     k += 1
(4)                 def parseData():
(8)                     """Returns stream of pairs b[i], x[i] for sparse6 format."""
(8)                     chunks = iter(data)
(8)                     d = None  # partial data word
(8)                     dLen = 0  # how many unparsed bits are left in d
(8)                     while 1:
(12)                        if dLen < 1:
(16)                            try:
(20)                                d = next(chunks)
(16)                            except StopIteration:
(20)                                return
(16)                            dLen = 6
(12)                        dLen -= 1
(12)                        b = (d >> dLen) & 1  # grab top remaining bit
(12)                        x = d & ((1 << dLen) - 1)  # partially built up value of x
(12)                        xLen = dLen  # how many bits included so far in x
(12)                        while xLen < k:  # now grab full chunks until we have enough
(16)                            try:
(20)                                d = next(chunks)
(16)                            except StopIteration:
(20)                                return
(16)                            dLen = 6
(16)                            x = (x << 6) + d
(16)                            xLen += 6
(12)                        x = x >> (xLen - k)  # shift back the extra bits
(12)                        dLen = xLen - k
(12)                        yield b, x
(4)                 v = 0
(4)                 G = nx.MultiGraph()
(4)                 G.add_nodes_from(range(n))
(4)                 multigraph = False
(4)                 for b, x in parseData():
(8)                     if b == 1:
(12)                        v += 1
(8)                     if x >= n or v >= n:
(12)                        break
(8)                     elif x > v:
(12)                        v = x
(8)                     else:
(12)                        if G.has_edge(x, v):
(16)                            multigraph = True
(12)                        G.add_edge(x, v)
(4)                 if not multigraph:
(8)                     G = nx.Graph(G)
(4)                 return G
(0)             def to_sparse6_bytes(G, nodes=None, header=True):
(4)                 """Convert an undirected graph to bytes in sparse6 format.
(4)                 Parameters
(4)                 ----------
(4)                 G : Graph (undirected)
(4)                 nodes: list or iterable
(7)                    Nodes are labeled 0...n-1 in the order provided.  If None the ordering
(7)                    given by ``G.nodes()`` is used.
(4)                 header: bool
(7)                    If True add '>>sparse6<<' bytes to head of data.
(4)                 Raises
(4)                 ------
(4)                 NetworkXNotImplemented
(8)                     If the graph is directed.
(4)                 ValueError
```

```
(8)                 If the graph has at least ``2 ** 36`` nodes; the sparse6 format
(8)                 is only defined for graphs of order less than ``2 ** 36``.
(4)             Examples
(4)             --------
(4)             >>> nx.to_sparse6_bytes(nx.path_graph(2))
(4)             b'>>sparse6<<:An\\n'
(4)             See Also
(4)             --------
(4)             to_sparse6_bytes, read_sparse6, write_sparse6_bytes
(4)             Notes
(4)             -----
(4)             The returned bytes end with a newline character.
(4)             The format does not support edge or node labels.
(4)             References
(4)             ----------
(4)             .. [1] Graph6 specification
(11)                   <https://users.cecs.anu.edu.au/~bdm/data/formats.html>
(4)             """
(4)             if nodes is not None:
(8)                 G = G.subgraph(nodes)
(4)             G = nx.convert_node_labels_to_integers(G, ordering="sorted")
(4)             return b"".join(_generate_sparse6_bytes(G, nodes, header))
(0)         @open_file(0, mode="rb")
(0)         @nx._dispatchable(graphs=None, returns_graph=True)
(0)         def read_sparse6(path):
(4)             """Read an undirected graph in sparse6 format from path.
(4)             Parameters
(4)             ----------
(4)             path : file or string
(7)                File or filename to write.
(4)             Returns
(4)             -------
(4)             G : Graph/Multigraph or list of Graphs/MultiGraphs
(7)                If the file contains multiple lines then a list of graphs is returned
(4)             Raises
(4)             ------
(4)             NetworkXError
(8)                 If the string is unable to be parsed in sparse6 format
(4)             Examples
(4)             --------
(4)             You can read a sparse6 file by giving the path to the file::
(8)                 >>> import tempfile
(8)                 >>> with tempfile.NamedTemporaryFile(delete=False) as f:
(8)                 ...     _ = f.write(b">>sparse6<<:An\\n")
(8)                 ...     _ = f.seek(0)
(8)                 ...     G = nx.read_sparse6(f.name)
(8)                 >>> list(G.edges())
(8)                 [(0, 1)]
(4)             You can also read a sparse6 file by giving an open file-like object::
(8)                 >>> import tempfile
(8)                 >>> with tempfile.NamedTemporaryFile() as f:
(8)                 ...     _ = f.write(b">>sparse6<<:An\\n")
(8)                 ...     _ = f.seek(0)
(8)                 ...     G = nx.read_sparse6(f)
(8)                 >>> list(G.edges())
(8)                 [(0, 1)]
(4)             See Also
(4)             --------
(4)             read_sparse6, from_sparse6_bytes
(4)             References
(4)             ----------
(4)             .. [1] Sparse6 specification
(11)                   <https://users.cecs.anu.edu.au/~bdm/data/formats.html>
(4)             """
(4)             glist = []
(4)             for line in path:
(8)                 line = line.strip()
(8)                 if not len(line):
(12)                    continue
```

```
244: (8)                    glist.append(from_sparse6_bytes(line))
245: (4)                if len(glist) == 1:
246: (8)                    return glist[0]
247: (4)                else:
248: (8)                    return glist
249: (0)            @not_implemented_for("directed")
250: (0)            @open_file(1, mode="wb")
251: (0)            def write_sparse6(G, path, nodes=None, header=True):
252: (4)                """Write graph G to given path in sparse6 format.
253: (4)                Parameters
254: (4)                ----------
255: (4)                G : Graph (undirected)
256: (4)                path : file or string
257: (7)                   File or filename to write
258: (4)                nodes: list or iterable
259: (7)                   Nodes are labeled 0...n-1 in the order provided.  If None the ordering
260: (7)                   given by G.nodes() is used.
261: (4)                header: bool
262: (7)                   If True add '>>sparse6<<' string to head of data
263: (4)                Raises
264: (4)                ------
265: (4)                NetworkXError
266: (8)                    If the graph is directed
267: (4)                Examples
268: (4)                --------
269: (4)                You can write a sparse6 file by giving the path to the file::
270: (8)                    >>> import tempfile
271: (8)                    >>> with tempfile.NamedTemporaryFile(delete=False) as f:
272: (8)                    ...     nx.write_sparse6(nx.path_graph(2), f.name)
273: (8)                    ...     print(f.read())
274: (8)                    b'>>sparse6<<:An\\n'
275: (4)                You can also write a sparse6 file by giving an open file-like object::
276: (8)                    >>> with tempfile.NamedTemporaryFile() as f:
277: (8)                    ...     nx.write_sparse6(nx.path_graph(2), f)
278: (8)                    ...     _ = f.seek(0)
279: (8)                    ...     print(f.read())
280: (8)                    b'>>sparse6<<:An\\n'
281: (4)                See Also
282: (4)                --------
283: (4)                read_sparse6, from_sparse6_bytes
284: (4)                Notes
285: (4)                -----
286: (4)                The format does not support edge or node labels.
287: (4)                References
288: (4)                ----------
289: (4)                .. [1] Sparse6 specification
290: (11)                      <https://users.cecs.anu.edu.au/~bdm/data/formats.html>
291: (4)                """
292: (4)                if nodes is not None:
293: (8)                    G = G.subgraph(nodes)
294: (4)                G = nx.convert_node_labels_to_integers(G, ordering="sorted")
295: (4)                for b in _generate_sparse6_bytes(G, nodes, header):
296: (8)                    path.write(b)
```

File 513 - text.py:

```
1: (0)              """
2: (0)              Text-based visual representations of graphs
3: (0)              """
4: (0)              import sys
5: (0)              import warnings
6: (0)              from collections import defaultdict
7: (0)              import networkx as nx
8: (0)              from networkx.utils import open_file
9: (0)              __all__ = ["forest_str", "generate_network_text", "write_network_text"]
10: (0)             class BaseGlyphs:
11: (4)                 @classmethod
```

```
(4)                 def as_dict(cls):
(8)                     return {
(12)                        a: getattr(cls, a)
(12)                        for a in dir(cls)
(12)                        if not a.startswith("_") and a != "as_dict"
(8)                     }
(0)             class AsciiBaseGlyphs(BaseGlyphs):
(4)                 empty: str = "+"
(4)                 newtree_last: str = "+-- "
(4)                 newtree_mid: str = "+-- "
(4)                 endof_forest: str = "    "
(4)                 within_forest: str = ":   "
(4)                 within_tree: str = "|   "
(0)             class AsciiDirectedGlyphs(AsciiBaseGlyphs):
(4)                 last: str = "L-> "
(4)                 mid: str = "|-> "
(4)                 backedge: str = "<-"
(4)                 vertical_edge: str = "!"
(0)             class AsciiUndirectedGlyphs(AsciiBaseGlyphs):
(4)                 last: str = "L-- "
(4)                 mid: str = "|-- "
(4)                 backedge: str = "-"
(4)                 vertical_edge: str = "|"
(0)             class UtfBaseGlyphs(BaseGlyphs):
(4)                 empty: str = "╙"
(4)                 newtree_last: str = "╙── "
(4)                 newtree_mid: str = "╟── "
(4)                 endof_forest: str = "    "
(4)                 within_forest: str = "╎   "
(4)                 within_tree: str = "│   "
(0)             class UtfDirectedGlyphs(UtfBaseGlyphs):
(4)                 last: str = "└─╼ "
(4)                 mid: str = "├─╼ "
(4)                 backedge: str = "╼"
(4)                 vertical_edge: str = "╽"
(0)             class UtfUndirectedGlyphs(UtfBaseGlyphs):
(4)                 last: str = "└── "
(4)                 mid: str = "├── "
(4)                 backedge: str = "─"
(4)                 vertical_edge: str = "│"
(0)             def generate_network_text(
(4)                 graph,
(4)                 with_labels=True,
(4)                 sources=None,
(4)                 max_depth=None,
(4)                 ascii_only=False,
(4)                 vertical_chains=False,
(0)             ):
(4)                 """Generate lines in the "network text" format
(4)                 This works via a depth-first traversal of the graph and writing a line for
(4)                 each unique node encountered. Non-tree edges are written to the right of
(4)                 each node, and connection to a non-tree edge is indicated with an
ellipsis.
(4)                 This representation works best when the input graph is a forest, but any
(4)                 graph can be represented.
(4)                 This notation is original to networkx, although it is simple enough that
it
(4)                 may be known in existing literature. See #5602 for details. The procedure
(4)                 is summarized as follows:
(4)                 1. Given a set of source nodes (which can be specified, or automatically
(4)                 discovered via finding the (strongly) connected components and choosing
one
(4)                 node with minimum degree from each), we traverse the graph in depth first
(4)                 order.
(4)                 2. Each reachable node will be printed exactly once on it's own line.
(4)                 3. Edges are indicated in one of four ways:
(8)                     a. a parent "L-style" connection on the upper left. This corresponds
to
(8)                     a traversal in the directed DFS tree.
```

```
            b. a backref "<-style" connection shown directly on the right. For
            directed graphs, these are drawn for any incoming edges to a node that
            is not a parent edge. For undirected graphs, these are drawn for only
            the non-parent edges that have already been represented (The edges that

            have not been represented will be handled in the recursive case).
            c. a child "L-style" connection on the lower right. Drawing of the
            children are handled recursively.
            d. if ``vertical_chains`` is true, and a parent node only has one child

            a "vertical-style" edge is drawn between them.
        4. The children of each node (wrt the directed DFS tree) are drawn
        underneath and to the right of it. In the case that a child node has already

        been drawn the connection is replaced with an ellipsis ("...") to indicate
        that there is one or more connections represented elsewhere.
        5. If a maximum depth is specified, an edge to nodes past this maximum
        depth will be represented by an ellipsis.
        6. If a a node has a truthy "collapse" value, then we do not traverse past
        that node.
        Parameters
        ----------
        graph : nx.DiGraph | nx.Graph
            Graph to represent
        with_labels : bool | str
            If True will use the "label" attribute of a node to display if it
            exists otherwise it will use the node value itself. If given as a
            string, then that attribute name will be used instead of "label".
            Defaults to True.
        sources : List
            Specifies which nodes to start traversal from. Note: nodes that are not

            reachable from one of these sources may not be shown. If unspecified,
            the minimal set of nodes needed to reach all others will be used.
        max_depth : int | None
            The maximum depth to traverse before stopping. Defaults to None.
        ascii_only : Boolean
            If True only ASCII characters are used to construct the visualization
        vertical_chains : Boolean
            If True, chains of nodes will be drawn vertically when possible.
        Yields
        ------
        str : a line of generated text
        Examples
        --------
        >>> graph = nx.path_graph(10)
        >>> graph.add_node("A")
        >>> graph.add_node("B")
        >>> graph.add_node("C")
        >>> graph.add_node("D")
        >>> graph.add_edge(9, "A")
        >>> graph.add_edge(9, "B")
        >>> graph.add_edge(9, "C")
        >>> graph.add_edge("C", "D")
        >>> graph.add_edge("C", "E")
        >>> graph.add_edge("C", "F")
        >>> nx.write_network_text(graph)
        ╙── 0
            └── 1
                └── 2
                    └── 3
                        └── 4
                            └── 5
                                └── 6
                                    └── 7
                                        └── 8
                                            └── 9
                                                ├── A
                                                ├── B
```

```
                                                        └── C
                                                            ├── D
                                                            ├── E
                                                            └── F
                >>> nx.write_network_text(graph, vertical_chains=True)
                ╙── 0
                    │
                    1
                    │
                    2
                    │
                    3
                    │
                    4
                    │
                    5
                    │
                    6
                    │
                    7
                    │
                    8
                    │
                    9
                    ├── A
                    ├── B
                    └── C
                        ├── D
                        ├── E
                        └── F
                """
                from typing import Any, NamedTuple
                class StackFrame(NamedTuple):
                    parent: Any
                    node: Any
                    indents: list
                    this_islast: bool
                    this_vertical: bool
                collapse_attr = "collapse"
                is_directed = graph.is_directed()
                if is_directed:
                    glyphs = AsciiDirectedGlyphs if ascii_only else UtfDirectedGlyphs
                    succ = graph.succ
                    pred = graph.pred
                else:
                    glyphs = AsciiUndirectedGlyphs if ascii_only else UtfUndirectedGlyphs
                    succ = graph.adj
                    pred = graph.adj
                if isinstance(with_labels, str):
                    label_attr = with_labels
                elif with_labels:
                    label_attr = "label"
                else:
                    label_attr = None
                if max_depth == 0:
                    yield glyphs.empty + " ..."
                elif len(graph.nodes) == 0:
                    yield glyphs.empty
                else:
                    if sources is None:
                        sources = _find_sources(graph)
                    last_idx = len(sources) - 1
                    stack = [
                        StackFrame(None, node, [], (idx == last_idx), False)
                        for idx, node in enumerate(sources)
                    ][::-1]
                    num_skipped_children = defaultdict(lambda: 0)
                    seen_nodes = set()
                    while stack:
```

```
(12)                        parent, node, indents, this_islast, this_vertical = stack.pop()
(12)                        if node is not Ellipsis:
(16)                            skip = node in seen_nodes
(16)                            if skip:
(20)                                num_skipped_children[parent] += 1
(16)                            if this_islast:
(20)                                if num_skipped_children[parent] and parent is not None:
(24)                                    next_islast = True
(24)                                    try_frame = StackFrame(
(28)                                        node, Ellipsis, indents, next_islast, False
(24)                                    )
(24)                                    stack.append(try_frame)
(24)                                    next_islast = False
(24)                                    try_frame = StackFrame(
(28)                                        parent, node, indents, next_islast, this_vertical
(24)                                    )
(24)                                    stack.append(try_frame)
(24)                                    continue
(16)                            if skip:
(20)                                continue
(16)                            seen_nodes.add(node)
(12)                        if not indents:
(16)                            if this_islast:
(20)                                this_vertical = False
(20)                                this_prefix = indents + [glyphs.newtree_last]
(20)                                next_prefix = indents + [glyphs.endof_forest]
(16)                            else:
(20)                                this_prefix = indents + [glyphs.newtree_mid]
(20)                                next_prefix = indents + [glyphs.within_forest]
(12)                        else:
(16)                            if this_vertical:
(20)                                this_prefix = indents
(20)                                next_prefix = indents
(16)                            else:
(20)                                if this_islast:
(24)                                    this_prefix = indents + [glyphs.last]
(24)                                    next_prefix = indents + [glyphs.endof_forest]
(20)                                else:
(24)                                    this_prefix = indents + [glyphs.mid]
(24)                                    next_prefix = indents + [glyphs.within_tree]
(12)                        if node is Ellipsis:
(16)                            label = " ..."
(16)                            suffix = ""
(16)                            children = []
(12)                        else:
(16)                            if label_attr is not None:
(20)                                label = str(graph.nodes[node].get(label_attr, node))
(16)                            else:
(20)                                label = str(node)
(16)                            if collapse_attr is not None:
(20)                                collapse = graph.nodes[node].get(collapse_attr, False)
(16)                            else:
(20)                                collapse = False
(16)                            if is_directed:
(20)                                children = list(succ[node])
(20)                                handled_parents = {parent}
(16)                            else:
(20)                                children = [
(24)                                    child for child in succ[node] if child not in
seen_nodes
(20)                                ]
(20)                                handled_parents = {*children, parent}
(16)                            if max_depth is not None and len(indents) == max_depth - 1:
(20)                                if children:
(24)                                    children = [Ellipsis]
(20)                                handled_parents = {parent}
(16)                            if collapse:
(20)                                if children:
(24)                                    children = [Ellipsis]
```

```
279: (20)                                handled_parents = {parent}
280: (16)                            other_parents = [p for p in pred[node] if p not in
handled_parents]
281: (16)                            if other_parents:
282: (20)                                if label_attr is not None:
283: (24)                                    other_parents_labels = ", ".join(
284: (28)                                        [
285: (32)                                            str(graph.nodes[p].get(label_attr, p))
286: (32)                                            for p in other_parents
287: (28)                                        ]
288: (24)                                    )
289: (20)                                else:
290: (24)                                    other_parents_labels = ", ".join(
291: (28)                                        [str(p) for p in other_parents]
292: (24)                                    )
293: (20)                                suffix = " ".join(["", glyphs.backedge,
other_parents_labels])
294: (16)                            else:
295: (20)                                suffix = ""
296: (12)                        if this_vertical:
297: (16)                            yield "".join(this_prefix + [glyphs.vertical_edge])
298: (12)                        yield "".join(this_prefix + [label, suffix])
299: (12)                        if vertical_chains:
300: (16)                            if is_directed:
301: (20)                                num_children = len(set(children))
302: (16)                            else:
303: (20)                                num_children = len(set(children) - {parent})
304: (16)                            next_is_vertical = num_children == 1
305: (12)                        else:
306: (16)                            next_is_vertical = False
307: (12)                        for idx, child in enumerate(children[::-1]):
308: (16)                            next_islast = idx == 0
309: (16)                            try_frame = StackFrame(
310: (20)                                node, child, next_prefix, next_islast, next_is_vertical
311: (16)                            )
312: (16)                            stack.append(try_frame)
313: (0)            @open_file(1, "w")
314: (0)            def write_network_text(
315: (4)                graph,
316: (4)                path=None,
317: (4)                with_labels=True,
318: (4)                sources=None,
319: (4)                max_depth=None,
320: (4)                ascii_only=False,
321: (4)                end="\n",
322: (4)                vertical_chains=False,
323: (0)            ):
324: (4)                """Creates a nice text representation of a graph
325: (4)                This works via a depth-first traversal of the graph and writing a line for
326: (4)                each unique node encountered. Non-tree edges are written to the right of
327: (4)                each node, and connection to a non-tree edge is indicated with an
ellipsis.
328: (4)                This representation works best when the input graph is a forest, but any
329: (4)                graph can be represented.
330: (4)                Parameters
331: (4)                ----------
332: (4)                graph : nx.DiGraph | nx.Graph
333: (8)                    Graph to represent
334: (4)                path : string or file or callable or None
335: (7)                   Filename or file handle for data output.
336: (7)                   if a function, then it will be called for each generated line.
337: (7)                   if None, this will default to "sys.stdout.write"
338: (4)                with_labels : bool | str
339: (8)                    If True will use the "label" attribute of a node to display if it
340: (8)                    exists otherwise it will use the node value itself. If given as a
341: (8)                    string, then that attribute name will be used instead of "label".
342: (8)                    Defaults to True.
343: (4)                sources : List
344: (8)                    Specifies which nodes to start traversal from. Note: nodes that are
```

```
not
                    reachable from one of these sources may not be shown. If unspecified,
                    the minimal set of nodes needed to reach all others will be used.
                max_depth : int | None
                    The maximum depth to traverse before stopping. Defaults to None.
                ascii_only : Boolean
                    If True only ASCII characters are used to construct the visualization
                end : string
                    The line ending character
                vertical_chains : Boolean
                    If True, chains of nodes will be drawn vertically when possible.
                Examples
                --------
                >>> graph = nx.balanced_tree(r=2, h=2, create_using=nx.DiGraph)
                >>> nx.write_network_text(graph)
                ╙── 0
                    ├─╼ 1
                    │   ├─╼ 3
                    │   └─╼ 4
                    └─╼ 2
                        ├─╼ 5
                        └─╼ 6
                >>> # A near tree with one non-tree edge
                >>> graph.add_edge(5, 1)
                >>> nx.write_network_text(graph)
                ╙── 0
                    ├─╼ 1 ━ 5
                    │   ├─╼ 3
                    │   └─╼ 4
                    └─╼ 2
                        ├─╼ 5
                        │   └─╼  ...
                        └─╼ 6
                >>> graph = nx.cycle_graph(5)
                >>> nx.write_network_text(graph)
                ╙── 0
                    ├── 1
                    │   └── 2
                    │       └── 3
                    │           └── 4 — 0
                    └──  ...
                >>> graph = nx.cycle_graph(5, nx.DiGraph)
                >>> nx.write_network_text(graph, vertical_chains=True)
                ╙── 0 ━ 4
                    ╽
                    1
                    ╽
                    2
                    ╽
                    3
                    ╽
                    4
                    └─╼  ...
                >>> nx.write_network_text(graph, vertical_chains=True, ascii_only=True)
                +-- 0 <- 4
                    !
                    1
                    !
                    2
                    !
                    3
                    !
                    4
                    L->  ...
                >>> graph = nx.generators.barbell_graph(4, 2)
                >>> nx.write_network_text(graph, vertical_chains=False)
                ╙── 4
                    ├── 5
                    │   └── 6
```

```
(8)                     │       ├── 7
(8)                     │       │   ├── 8 - 6
(8)                     │       │   │   └── 9 - 6, 7
(8)                     │       │   └── ...
(8)                     │       └── ...
(8)                     └── 3
(12)                        ├── 0
(12)                        │   ├── 1 - 3
(12)                        │   │   └── 2 - 0, 3
(12)                        │   └── ...
(12)                        └── ...
(4)                 >>> nx.write_network_text(graph, vertical_chains=True)
(4)                 ╙── 4
(8)                     ├── 5
(8)                     │
(8)                     6
(8)                     ├── 7
(8)                     │   ├── 8 - 6
(8)                     │   │
(8)                     │   9 - 6, 7
(8)                     │   └── ...
(8)                     └── ...
(8)                     └── 3
(12)                        ├── 0
(12)                        │   ├── 1 - 3
(12)                        │   │
(12)                        │   2 - 0, 3
(12)                        │   └── ...
(12)                        └── ...
(4)                 >>> graph = nx.complete_graph(5, create_using=nx.Graph)
(4)                 >>> nx.write_network_text(graph)
(4)                 ╙── 0
(8)                     ├── 1
(8)                     │   ├── 2 - 0
(8)                     │   │   ├── 3 - 0, 1
(8)                     │   │   │   └── 4 - 0, 1, 2
(8)                     │   │   └── ...
(8)                     │   └── ...
(8)                     └── ...
(4)                 >>> graph = nx.complete_graph(3, create_using=nx.DiGraph)
(4)                 >>> nx.write_network_text(graph)
(4)                 ╙── 0 ╾ 1, 2
(8)                     ├─╼ 1 ╾ 2
(8)                     │   ├─╼ 2 ╾ 0
(8)                     │   │   └─╼ ...
(8)                     │   └─╼ ...
(8)                     └─╼ ...
(4)                 """
(4)                 if path is None:
(8)                     _write = sys.stdout.write
(4)                 elif hasattr(path, "write"):
(8)                     _write = path.write
(4)                 elif callable(path):
(8)                     _write = path
(4)                 else:
(8)                     raise TypeError(type(path))
(4)                 for line in generate_network_text(
(8)                     graph,
(8)                     with_labels=with_labels,
(8)                     sources=sources,
(8)                     max_depth=max_depth,
(8)                     ascii_only=ascii_only,
(8)                     vertical_chains=vertical_chains,
(4)                 ):
(8)                     _write(line + end)
(0)             def _find_sources(graph):
(4)                 """
(4)                 Determine a minimal set of nodes such that the entire graph is reachable
(4)                 """
```

```
    if graph.is_directed():
        sccs = list(nx.strongly_connected_components(graph))
        scc_graph = nx.condensation(graph, sccs)
        supernode_to_nodes = {sn: [] for sn in scc_graph.nodes()}
        mapping = scc_graph.graph["mapping"]
        for n in graph.nodes:
            sn = mapping[n]
            supernode_to_nodes[sn].append(n)
        sources = []
        for sn in scc_graph.nodes():
            if scc_graph.in_degree[sn] == 0:
                scc = supernode_to_nodes[sn]
                node = min(scc, key=lambda n: graph.in_degree[n])
                sources.append(node)
    else:
        sources = [
            min(cc, key=lambda n: graph.degree[n])
            for cc in nx.connected_components(graph)
        ]
        sources = sorted(sources, key=lambda n: graph.degree[n])
    return sources
def forest_str(graph, with_labels=True, sources=None, write=None, ascii_only=False):
    """Creates a nice utf8 representation of a forest
    This function has been superseded by
    :func:`nx.readwrite.text.generate_network_text`, which should be used
    instead.
    Parameters
    ----------
    graph : nx.DiGraph | nx.Graph
        Graph to represent (must be a tree, forest, or the empty graph)
    with_labels : bool
        If True will use the "label" attribute of a node to display if it
        exists otherwise it will use the node value itself. Defaults to True.
    sources : List
        Mainly relevant for undirected forests, specifies which nodes to list
        first. If unspecified the root nodes of each tree will be used for
        directed forests; for undirected forests this defaults to the nodes
        with the smallest degree.
    write : callable
        Function to use to write to, if None new lines are appended to
        a list and returned. If set to the `print` function, lines will
        be written to stdout as they are generated. If specified,
        this function will return None. Defaults to None.
    ascii_only : Boolean
        If True only ASCII characters are used to construct the visualization
    Returns
    -------
    str | None :
        utf8 representation of the tree / forest
    Examples
    --------
    >>> graph = nx.balanced_tree(r=2, h=3, create_using=nx.DiGraph)
    >>> print(nx.forest_str(graph))
    ╙╼ 0
        ├╼ 1
        │   ├╼ 3
        │   │   ├╼ 7
        │   │   └╼ 8
        │   └╼ 4
        │       ├╼ 9
        │       └╼ 10
        └╼ 2
            ├╼ 5
            │   ├╼ 11
            │   └╼ 12
            └╼ 6
                ├╼ 13
                └╼ 14
```

```
    >>> graph = nx.balanced_tree(r=1, h=2, create_using=nx.Graph)
    >>> print(nx.forest_str(graph))
    ╙── 0
        └── 1
            └── 2
    >>> print(nx.forest_str(graph, ascii_only=True))
    +-- 0
        L-- 1
            L-- 2
    """
    msg = (
        "\nforest_str is deprecated as of version 3.1 and will be removed "
        "in version 3.3. Use generate_network_text or write_network_text "
        "instead.\n"
    )
    warnings.warn(msg, DeprecationWarning)
    if len(graph.nodes) > 0:
        if not nx.is_forest(graph):
            raise nx.NetworkXNotImplemented("input must be a forest or the empty graph")
    printbuf = []
    if write is None:
        _write = printbuf.append
    else:
        _write = write
    write_network_text(
        graph,
        _write,
        with_labels=with_labels,
        sources=sources,
        ascii_only=ascii_only,
        end="",
    )
    if write is None:
        return "\n".join(printbuf)
def _parse_network_text(lines):
    """Reconstructs a graph from a network text representation.
    This is mainly used for testing.  Network text is for display, not
    serialization, as such this cannot parse all network text representations
    because node labels can be ambiguous with the glyphs and indentation used
    to represent edge structure. Additionally, there is no way to determine if
    disconnected graphs were originally directed or undirected.
    Parameters
    ----------
    lines : list or iterator of strings
        Input data in network text format
    Returns
    -------
    G: NetworkX graph
        The graph corresponding to the lines in network text format.
    """
    from itertools import chain
    from typing import Any, NamedTuple, Union
    class ParseStackFrame(NamedTuple):
        node: Any
        indent: int
        has_vertical_child: int | None
    initial_line_iter = iter(lines)
    is_ascii = None
    is_directed = None
    initial_lines = []
    try:
        first_line = next(initial_line_iter)
    except StopIteration:
        ...
    else:
        initial_lines.append(first_line)
        first_char = first_line[0]
        if first_char in {
```

```
(12)                        UtfBaseGlyphs.empty,
(12)                        UtfBaseGlyphs.newtree_mid[0],
(12)                        UtfBaseGlyphs.newtree_last[0],
(8)                     }:
(12)                        is_ascii = False
(8)                     elif first_char in {
(12)                        AsciiBaseGlyphs.empty,
(12)                        AsciiBaseGlyphs.newtree_mid[0],
(12)                        AsciiBaseGlyphs.newtree_last[0],
(8)                     }:
(12)                        is_ascii = True
(8)                     else:
(12)                        raise AssertionError(f"Unexpected first character: {first_char}")
(4)                 if is_ascii:
(8)                     directed_glyphs = AsciiDirectedGlyphs.as_dict()
(8)                     undirected_glyphs = AsciiUndirectedGlyphs.as_dict()
(4)                 else:
(8)                     directed_glyphs = UtfDirectedGlyphs.as_dict()
(8)                     undirected_glyphs = UtfUndirectedGlyphs.as_dict()
(4)                 directed_items = set(directed_glyphs.values())
(4)                 undirected_items = set(undirected_glyphs.values())
(4)                 unambiguous_directed_items = []
(4)                 for item in directed_items:
(8)                     other_items = undirected_items
(8)                     other_supersets = [other for other in other_items if item in other]
(8)                     if not other_supersets:
(12)                        unambiguous_directed_items.append(item)
(4)                 unambiguous_undirected_items = []
(4)                 for item in undirected_items:
(8)                     other_items = directed_items
(8)                     other_supersets = [other for other in other_items if item in other]
(8)                     if not other_supersets:
(12)                        unambiguous_undirected_items.append(item)
(4)                 for line in initial_line_iter:
(8)                     initial_lines.append(line)
(8)                     if any(item in line for item in unambiguous_undirected_items):
(12)                        is_directed = False
(12)                        break
(8)                     elif any(item in line for item in unambiguous_directed_items):
(12)                        is_directed = True
(12)                        break
(4)                 if is_directed is None:
(8)                     is_directed = False
(4)                 glyphs = directed_glyphs if is_directed else undirected_glyphs
(4)                 backedge_symbol = " " + glyphs["backedge"] + " "
(4)                 parsing_line_iter = chain(initial_lines, initial_line_iter)
(4)                 edges = []
(4)                 nodes = []
(4)                 is_empty = None
(4)                 noparent = object()  # sentinel value
(4)                 stack = [ParseStackFrame(noparent, -1, None)]
(4)                 for line in parsing_line_iter:
(8)                     if line == glyphs["empty"]:
(12)                        is_empty = True
(12)                        continue
(8)                     if backedge_symbol in line:
(12)                        node_part, backedge_part = line.split(backedge_symbol)
(12)                        backedge_nodes = [u.strip() for u in backedge_part.split(", ")]
(12)                        node_part = node_part.rstrip()
(12)                        prefix, node = node_part.rsplit(" ", 1)
(12)                        node = node.strip()
(12)                        edges.extend([(u, node) for u in backedge_nodes])
(8)                     else:
(12)                        prefix, node = line.rsplit(" ", 1)
(12)                        node = node.strip()
(8)                     prev = stack.pop()
(8)                     if node in glyphs["vertical_edge"]:
(12)                        modified_prev = ParseStackFrame(
(16)                            prev.node,
```

```
687: (16)                            prev.indent,
688: (16)                            True,
689: (12)                        )
690: (12)                        stack.append(modified_prev)
691: (12)                        continue
692: (8)                    indent = len(prefix)
693: (8)                    curr = ParseStackFrame(node, indent, None)
694: (8)                    if prev.has_vertical_child:
695: (12)                        ...
696: (8)                    else:
697: (12)                        while curr.indent <= prev.indent:
698: (16)                            prev = stack.pop()
699: (8)                    if node == "...":
700: (12)                        stack.append(prev)
701: (8)                    else:
702: (12)                        stack.append(prev)
703: (12)                        stack.append(curr)
704: (12)                        nodes.append(curr.node)
705: (12)                        if prev.node is not noparent:
706: (16)                            edges.append((prev.node, curr.node))
707: (4)                if is_empty:
708: (8)                    assert len(nodes) == 0
709: (4)                cls = nx.DiGraph if is_directed else nx.Graph
710: (4)                new = cls()
711: (4)                new.add_nodes_from(nodes)
712: (4)                new.add_edges_from(edges)
713: (4)                return new
```

File 514 - __init__.py:

```
1: (0)              """
2: (0)              *********
3: (0)              JSON data
4: (0)              *********
5: (0)              Generate and parse JSON serializable data for NetworkX graphs.
6: (0)              These formats are suitable for use with the d3.js examples https://d3js.org/
7: (0)              The three formats that you can generate with NetworkX are:
8: (1)               - node-link like in the d3.js example https://bl.ocks.org/mbostock/4062045
9: (1)               - tree like in the d3.js example https://bl.ocks.org/mbostock/4063550
10: (1)              - adjacency like in the d3.js example https://bost.ocks.org/mike/miserables/
11: (0)             """
12: (0)             from networkx.readwrite.json_graph.node_link import *
13: (0)             from networkx.readwrite.json_graph.adjacency import *
14: (0)             from networkx.readwrite.json_graph.tree import *
15: (0)             from networkx.readwrite.json_graph.cytoscape import *
```

File 515 - adjacency.py:

```
1: (0)              import networkx as nx
2: (0)              __all__ = ["adjacency_data", "adjacency_graph"]
3: (0)              _attrs = {"id": "id", "key": "key"}
4: (0)              def adjacency_data(G, attrs=_attrs):
5: (4)                  """Returns data in adjacency format that is suitable for JSON
serialization
6: (4)                  and use in JavaScript documents.
7: (4)                  Parameters
8: (4)                  ----------
9: (4)                  G : NetworkX graph
10: (4)                 attrs : dict
11: (8)                     A dictionary that contains two keys 'id' and 'key'. The corresponding
12: (8)                     values provide the attribute names for storing NetworkX-internal graph
13: (8)                     data. The values should be unique. Default value:
14: (8)                     :samp:`dict(id='id', key='key')`.
15: (8)                     If some user-defined graph data use these attribute names as data
keys,
```

```
16: (8)                     they may be silently dropped.
17: (4)                 Returns
18: (4)                 -------
19: (4)                 data : dict
20: (7)                    A dictionary with adjacency formatted data.
21: (4)                 Raises
22: (4)                 ------
23: (4)                 NetworkXError
24: (8)                     If values in attrs are not unique.
25: (4)                 Examples
26: (4)                 --------
27: (4)                 >>> from networkx.readwrite import json_graph
28: (4)                 >>> G = nx.Graph([(1, 2)])
29: (4)                 >>> data = json_graph.adjacency_data(G)
30: (4)                 To serialize with json
31: (4)                 >>> import json
32: (4)                 >>> s = json.dumps(data)
33: (4)                 Notes
34: (4)                 -----
35: (4)                 Graph, node, and link attributes will be written when using this format
36: (4)                 but attribute keys must be strings if you want to serialize the resulting
37: (4)                 data with JSON.
38: (4)                 The default value of attrs will be changed in a future release of
NetworkX.
39: (4)                 See Also
40: (4)                 --------
41: (4)                 adjacency_graph, node_link_data, tree_data
42: (4)                 """
43: (4)                 multigraph = G.is_multigraph()
44: (4)                 id_ = attrs["id"]
45: (4)                 key = None if not multigraph else attrs["key"]
46: (4)                 if id_ == key:
47: (8)                     raise nx.NetworkXError("Attribute names are not unique.")
48: (4)                 data = {}
49: (4)                 data["directed"] = G.is_directed()
50: (4)                 data["multigraph"] = multigraph
51: (4)                 data["graph"] = list(G.graph.items())
52: (4)                 data["nodes"] = []
53: (4)                 data["adjacency"] = []
54: (4)                 for n, nbrdict in G.adjacency():
55: (8)                     data["nodes"].append({**G.nodes[n], id_: n})
56: (8)                     adj = []
57: (8)                     if multigraph:
58: (12)                        for nbr, keys in nbrdict.items():
59: (16)                            for k, d in keys.items():
60: (20)                                adj.append({**d, id_: nbr, key: k})
61: (8)                     else:
62: (12)                        for nbr, d in nbrdict.items():
63: (16)                            adj.append({**d, id_: nbr})
64: (8)                     data["adjacency"].append(adj)
65: (4)                 return data
66: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
67: (0)             def adjacency_graph(data, directed=False, multigraph=True, attrs=_attrs):
68: (4)                 """Returns graph from adjacency data format.
69: (4)                 Parameters
70: (4)                 ----------
71: (4)                 data : dict
72: (8)                     Adjacency list formatted graph data
73: (4)                 directed : bool
74: (8)                     If True, and direction not specified in data, return a directed graph.
75: (4)                 multigraph : bool
76: (8)                     If True, and multigraph not specified in data, return a multigraph.
77: (4)                 attrs : dict
78: (8)                     A dictionary that contains two keys 'id' and 'key'. The corresponding
79: (8)                     values provide the attribute names for storing NetworkX-internal graph
80: (8)                     data. The values should be unique. Default value:
81: (8)                     :samp:`dict(id='id', key='key')`.
82: (4)                 Returns
83: (4)                 -------
```

```
84: (4)                 G : NetworkX graph
85: (7)                    A NetworkX graph object
86: (4)                 Examples
87: (4)                 --------
88: (4)                 >>> from networkx.readwrite import json_graph
89: (4)                 >>> G = nx.Graph([(1, 2)])
90: (4)                 >>> data = json_graph.adjacency_data(G)
91: (4)                 >>> H = json_graph.adjacency_graph(data)
92: (4)                 Notes
93: (4)                 -----
94: (4)                 The default value of attrs will be changed in a future release of
NetworkX.
95: (4)                 See Also
96: (4)                 --------
97: (4)                 adjacency_graph, node_link_data, tree_data
98: (4)                 """
99: (4)                 multigraph = data.get("multigraph", multigraph)
100: (4)                directed = data.get("directed", directed)
101: (4)                if multigraph:
102: (8)                    graph = nx.MultiGraph()
103: (4)                else:
104: (8)                    graph = nx.Graph()
105: (4)                if directed:
106: (8)                    graph = graph.to_directed()
107: (4)                id_ = attrs["id"]
108: (4)                key = None if not multigraph else attrs["key"]
109: (4)                graph.graph = dict(data.get("graph", []))
110: (4)                mapping = []
111: (4)                for d in data["nodes"]:
112: (8)                    node_data = d.copy()
113: (8)                    node = node_data.pop(id_)
114: (8)                    mapping.append(node)
115: (8)                    graph.add_node(node)
116: (8)                    graph.nodes[node].update(node_data)
117: (4)                for i, d in enumerate(data["adjacency"]):
118: (8)                    source = mapping[i]
119: (8)                    for tdata in d:
120: (12)                       target_data = tdata.copy()
121: (12)                       target = target_data.pop(id_)
122: (12)                       if not multigraph:
123: (16)                           graph.add_edge(source, target)
124: (16)                           graph[source][target].update(target_data)
125: (12)                       else:
126: (16)                           ky = target_data.pop(key, None)
127: (16)                           graph.add_edge(source, target, key=ky)
128: (16)                           graph[source][target][ky].update(target_data)
129: (4)                return graph

----------------------------------------

File 516 - cytoscape.py:

1: (0)              import networkx as nx
2: (0)              __all__ = ["cytoscape_data", "cytoscape_graph"]
3: (0)              def cytoscape_data(G, name="name", ident="id"):
4: (4)                  """Returns data in Cytoscape JSON format (cyjs).
5: (4)                  Parameters
6: (4)                  ----------
7: (4)                  G : NetworkX Graph
8: (8)                      The graph to convert to cytoscape format
9: (4)                  name : string
10: (8)                     A string which is mapped to the 'name' node element in cyjs format.
11: (8)                     Must not have the same value as `ident`.
12: (4)                 ident : string
13: (8)                     A string which is mapped to the 'id' node element in cyjs format.
14: (8)                     Must not have the same value as `name`.
15: (4)                 Returns
16: (4)                 -------
17: (4)                 data: dict
```

```
(8)                     A dictionary with cyjs formatted data.
(4)                 Raises
(4)                 ------
(4)                 NetworkXError
(8)                     If the values for `name` and `ident` are identical.
(4)                 See Also
(4)                 --------
(4)                 cytoscape_graph: convert a dictionary in cyjs format to a graph
(4)                 References
(4)                 ----------
(4)                 .. [1] Cytoscape user's manual:
(7)                    http://manual.cytoscape.org/en/stable/index.html
(4)                 Examples
(4)                 --------
(4)                 >>> G = nx.path_graph(2)
(4)                 >>> nx.cytoscape_data(G)  # doctest: +SKIP
(4)                 {'data': [],
(5)                  'directed': False,
(5)                  'multigraph': False,
(5)                  'elements': {'nodes': [{'data': {'id': '0', 'value': 0, 'name': '0'}},
(7)                    {'data': {'id': '1', 'value': 1, 'name': '1'}}],
(6)                   'edges': [{'data': {'source': 0, 'target': 1}}]}}
(4)                 """
(4)                 if name == ident:
(8)                     raise nx.NetworkXError("name and ident must be different.")
(4)                 jsondata = {"data": list(G.graph.items())}
(4)                 jsondata["directed"] = G.is_directed()
(4)                 jsondata["multigraph"] = G.is_multigraph()
(4)                 jsondata["elements"] = {"nodes": [], "edges": []}
(4)                 nodes = jsondata["elements"]["nodes"]
(4)                 edges = jsondata["elements"]["edges"]
(4)                 for i, j in G.nodes.items():
(8)                     n = {"data": j.copy()}
(8)                     n["data"]["id"] = j.get(ident) or str(i)
(8)                     n["data"]["value"] = i
(8)                     n["data"]["name"] = j.get(name) or str(i)
(8)                     nodes.append(n)
(4)                 if G.is_multigraph():
(8)                     for e in G.edges(keys=True):
(12)                        n = {"data": G.adj[e[0]][e[1]][e[2]].copy()}
(12)                        n["data"]["source"] = e[0]
(12)                        n["data"]["target"] = e[1]
(12)                        n["data"]["key"] = e[2]
(12)                        edges.append(n)
(4)                 else:
(8)                     for e in G.edges():
(12)                        n = {"data": G.adj[e[0]][e[1]].copy()}
(12)                        n["data"]["source"] = e[0]
(12)                        n["data"]["target"] = e[1]
(12)                        edges.append(n)
(4)                 return jsondata
(0)             @nx._dispatchable(graphs=None, returns_graph=True)
(0)             def cytoscape_graph(data, name="name", ident="id"):
(4)                 """
(4)                 Create a NetworkX graph from a dictionary in cytoscape JSON format.
(4)                 Parameters
(4)                 ----------
(4)                 data : dict
(8)                     A dictionary of data conforming to cytoscape JSON format.
(4)                 name : string
(8)                     A string which is mapped to the 'name' node element in cyjs format.
(8)                     Must not have the same value as `ident`.
(4)                 ident : string
(8)                     A string which is mapped to the 'id' node element in cyjs format.
(8)                     Must not have the same value as `name`.
(4)                 Returns
(4)                 -------
(4)                 graph : a NetworkX graph instance
(8)                     The `graph` can be an instance of `Graph`, `DiGraph`, `MultiGraph`, or
```

```
 87: (8)                    `MultiDiGraph` depending on the input data.
 88: (4)                Raises
 89: (4)                ------
 90: (4)                NetworkXError
 91: (8)                    If the `name` and `ident` attributes are identical.
 92: (4)                See Also
 93: (4)                --------
 94: (4)                cytoscape_data: convert a NetworkX graph to a dict in cyjs format
 95: (4)                References
 96: (4)                ----------
 97: (4)                .. [1] Cytoscape user's manual:
 98: (7)                   http://manual.cytoscape.org/en/stable/index.html
 99: (4)                Examples
 100: (4)               --------
 101: (4)               >>> data_dict = {
 102: (4)               ...     "data": [],
 103: (4)               ...     "directed": False,
 104: (4)               ...     "multigraph": False,
 105: (4)               ...     "elements": {
 106: (4)               ...         "nodes": [
 107: (4)               ...             {"data": {"id": "0", "value": 0, "name": "0"}},
 108: (4)               ...             {"data": {"id": "1", "value": 1, "name": "1"}},
 109: (4)               ...         ],
 110: (4)               ...         "edges": [{"data": {"source": 0, "target": 1}}],
 111: (4)               ...     },
 112: (4)               ... }
 113: (4)               >>> G = nx.cytoscape_graph(data_dict)
 114: (4)               >>> G.name
 115: (4)               ''
 116: (4)               >>> G.nodes()
 117: (4)               NodeView((0, 1))
 118: (4)               >>> G.nodes(data=True)[0]
 119: (4)               {'id': '0', 'value': 0, 'name': '0'}
 120: (4)               >>> G.edges(data=True)
 121: (4)               EdgeDataView([(0, 1, {'source': 0, 'target': 1})])
 122: (4)               """
 123: (4)               if name == ident:
 124: (8)                   raise nx.NetworkXError("name and ident must be different.")
 125: (4)               multigraph = data.get("multigraph")
 126: (4)               directed = data.get("directed")
 127: (4)               if multigraph:
 128: (8)                   graph = nx.MultiGraph()
 129: (4)               else:
 130: (8)                   graph = nx.Graph()
 131: (4)               if directed:
 132: (8)                   graph = graph.to_directed()
 133: (4)               graph.graph = dict(data.get("data"))
 134: (4)               for d in data["elements"]["nodes"]:
 135: (8)                   node_data = d["data"].copy()
 136: (8)                   node = d["data"]["value"]
 137: (8)                   if d["data"].get(name):
 138: (12)                      node_data[name] = d["data"].get(name)
 139: (8)                   if d["data"].get(ident):
 140: (12)                      node_data[ident] = d["data"].get(ident)
 141: (8)                   graph.add_node(node)
 142: (8)                   graph.nodes[node].update(node_data)
 143: (4)               for d in data["elements"]["edges"]:
 144: (8)                   edge_data = d["data"].copy()
 145: (8)                   sour = d["data"]["source"]
 146: (8)                   targ = d["data"]["target"]
 147: (8)                   if multigraph:
 148: (12)                      key = d["data"].get("key", 0)
 149: (12)                      graph.add_edge(sour, targ, key=key)
 150: (12)                      graph.edges[sour, targ, key].update(edge_data)
 151: (8)                   else:
 152: (12)                      graph.add_edge(sour, targ)
 153: (12)                      graph.edges[sour, targ].update(edge_data)
 154: (4)               return graph
```

```
----------------------------------------

File 517 - node_link.py:

1: (0)              from itertools import chain, count
2: (0)              import networkx as nx
3: (0)              __all__ = ["node_link_data", "node_link_graph"]
4: (0)              _attrs = {
5: (4)                  "source": "source",
6: (4)                  "target": "target",
7: (4)                  "name": "id",
8: (4)                  "key": "key",
9: (4)                  "link": "links",
10: (0)             }
11: (0)             def _to_tuple(x):
12: (4)                 """Converts lists to tuples, including nested lists.
13: (4)                 All other non-list inputs are passed through unmodified. This function is
14: (4)                 intended to be used to convert potentially nested lists from json files
15: (4)                 into valid nodes.
16: (4)                 Examples
17: (4)                 --------
18: (4)                 >>> _to_tuple([1, 2, [3, 4]])
19: (4)                 (1, 2, (3, 4))
20: (4)                 """
21: (4)                 if not isinstance(x, tuple | list):
22: (8)                     return x
23: (4)                 return tuple(map(_to_tuple, x))
24: (0)             def node_link_data(
25: (4)                 G,
26: (4)                 *,
27: (4)                 source="source",
28: (4)                 target="target",
29: (4)                 name="id",
30: (4)                 key="key",
31: (4)                 link="links",
32: (0)             ):
33: (4)                 """Returns data in node-link format that is suitable for JSON
serialization
34: (4)                 and use in JavaScript documents.
35: (4)                 Parameters
36: (4)                 ----------
37: (4)                 G : NetworkX graph
38: (4)                 source : string
39: (8)                     A string that provides the 'source' attribute name for storing
NetworkX-internal graph data.
40: (4)                 target : string
41: (8)                     A string that provides the 'target' attribute name for storing
NetworkX-internal graph data.
42: (4)                 name : string
43: (8)                     A string that provides the 'name' attribute name for storing NetworkX-
internal graph data.
44: (4)                 key : string
45: (8)                     A string that provides the 'key' attribute name for storing NetworkX-
internal graph data.
46: (4)                 link : string
47: (8)                     A string that provides the 'link' attribute name for storing NetworkX-
internal graph data.
48: (4)                 Returns
49: (4)                 -------
50: (4)                 data : dict
51: (7)                    A dictionary with node-link formatted data.
52: (4)                 Raises
53: (4)                 ------
54: (4)                 NetworkXError
55: (8)                     If the values of 'source', 'target' and 'key' are not unique.
56: (4)                 Examples
57: (4)                 --------
58: (4)                 >>> G = nx.Graph([("A", "B")])
59: (4)                 >>> data1 = nx.node_link_data(G)
```

```
60: (4)                 >>> data1
61: (4)                 {'directed': False, 'multigraph': False, 'graph': {}, 'nodes': [{'id':
'A'}, {'id': 'B'}], 'links': [{'source': 'A', 'target': 'B'}]}
62: (4)                 To serialize with JSON
63: (4)                 >>> import json
64: (4)                 >>> s1 = json.dumps(data1)
65: (4)                 >>> s1
66: (4)                 '{"directed": false, "multigraph": false, "graph": {}, "nodes": [{"id":
"A"}, {"id": "B"}], "links": [{"source": "A", "target": "B"}]}'
67: (4)                 A graph can also be serialized by passing `node_link_data` as an encoder
function. The two methods are equivalent.
68: (4)                 >>> s1 = json.dumps(G, default=nx.node_link_data)
69: (4)                 >>> s1
70: (4)                 '{"directed": false, "multigraph": false, "graph": {}, "nodes": [{"id":
"A"}, {"id": "B"}], "links": [{"source": "A", "target": "B"}]}'
71: (4)                 The attribute names for storing NetworkX-internal graph data can
72: (4)                 be specified as keyword options.
73: (4)                 >>> H = nx.gn_graph(2)
74: (4)                 >>> data2 = nx.node_link_data(H, link="edges", source="from", target="to")
75: (4)                 >>> data2
76: (4)                 {'directed': True, 'multigraph': False, 'graph': {}, 'nodes': [{'id': 0},
{'id': 1}], 'edges': [{'from': 1, 'to': 0}]}
77: (4)                 Notes
78: (4)                 -----
79: (4)                 Graph, node, and link attributes are stored in this format.  Note that
80: (4)                 attribute keys will be converted to strings in order to comply with JSON.
81: (4)                 Attribute 'key' is only used for multigraphs.
82: (4)                 To use `node_link_data` in conjunction with `node_link_graph`,
83: (4)                 the keyword names for the attributes must match.
84: (4)                 See Also
85: (4)                 --------
86: (4)                 node_link_graph, adjacency_data, tree_data
87: (4)                 """
88: (4)                 multigraph = G.is_multigraph()
89: (4)                 key = None if not multigraph else key
90: (4)                 if len({source, target, key}) < 3:
91: (8)                     raise nx.NetworkXError("Attribute names are not unique.")
92: (4)                 data = {
93: (8)                     "directed": G.is_directed(),
94: (8)                     "multigraph": multigraph,
95: (8)                     "graph": G.graph,
96: (8)                     "nodes": [{**G.nodes[n], name: n} for n in G],
97: (4)                 }
98: (4)                 if multigraph:
99: (8)                     data[link] = [
100: (12)                       {**d, source: u, target: v, key: k}
101: (12)                       for u, v, k, d in G.edges(keys=True, data=True)
102: (8)                    ]
103: (4)                else:
104: (8)                    data[link] = [{**d, source: u, target: v} for u, v, d in
G.edges(data=True)]
105: (4)                return data
106: (0)            @nx._dispatchable(graphs=None, returns_graph=True)
107: (0)            def node_link_graph(
108: (4)                data,
109: (4)                directed=False,
110: (4)                multigraph=True,
111: (4)                *,
112: (4)                source="source",
113: (4)                target="target",
114: (4)                name="id",
115: (4)                key="key",
116: (4)                link="links",
117: (0)            ):
118: (4)                """Returns graph from node-link data format.
119: (4)                Useful for de-serialization from JSON.
120: (4)                Parameters
121: (4)                ----------
122: (4)                data : dict
```

```
(8)                     node-link formatted graph data
(4)                 directed : bool
(8)                     If True, and direction not specified in data, return a directed graph.
(4)                 multigraph : bool
(8)                     If True, and multigraph not specified in data, return a multigraph.
(4)                 source : string
(8)                     A string that provides the 'source' attribute name for storing
NetworkX-internal graph data.
(4)                 target : string
(8)                     A string that provides the 'target' attribute name for storing
NetworkX-internal graph data.
(4)                 name : string
(8)                     A string that provides the 'name' attribute name for storing NetworkX-
internal graph data.
(4)                 key : string
(8)                     A string that provides the 'key' attribute name for storing NetworkX-
internal graph data.
(4)                 link : string
(8)                     A string that provides the 'link' attribute name for storing NetworkX-
internal graph data.
(4)                 Returns
(4)                 -------
(4)                 G : NetworkX graph
(8)                     A NetworkX graph object
(4)                 Examples
(4)                 --------
(4)                 Create data in node-link format by converting a graph.
(4)                 >>> G = nx.Graph([("A", "B")])
(4)                 >>> data = nx.node_link_data(G)
(4)                 >>> data
(4)                 {'directed': False, 'multigraph': False, 'graph': {}, 'nodes': [{'id':
'A'}, {'id': 'B'}], 'links': [{'source': 'A', 'target': 'B'}]}
(4)                 Revert data in node-link format to a graph.
(4)                 >>> H = nx.node_link_graph(data)
(4)                 >>> print(H.edges)
(4)                 [('A', 'B')]
(4)                 To serialize and deserialize a graph with JSON,
(4)                 >>> import json
(4)                 >>> d = json.dumps(node_link_data(G))
(4)                 >>> H = node_link_graph(json.loads(d))
(4)                 >>> print(G.edges, H.edges)
(4)                 [('A', 'B')] [('A', 'B')]
(4)                 Notes
(4)                 -----
(4)                 Attribute 'key' is only used for multigraphs.
(4)                 To use `node_link_data` in conjunction with `node_link_graph`,
(4)                 the keyword names for the attributes must match.
(4)                 See Also
(4)                 --------
(4)                 node_link_data, adjacency_data, tree_data
(4)                 """
(4)                 multigraph = data.get("multigraph", multigraph)
(4)                 directed = data.get("directed", directed)
(4)                 if multigraph:
(8)                     graph = nx.MultiGraph()
(4)                 else:
(8)                     graph = nx.Graph()
(4)                 if directed:
(8)                     graph = graph.to_directed()
(4)                 key = None if not multigraph else key
(4)                 graph.graph = data.get("graph", {})
(4)                 c = count()
(4)                 for d in data["nodes"]:
(8)                     node = _to_tuple(d.get(name, next(c)))
(8)                     nodedata = {str(k): v for k, v in d.items() if k != name}
(8)                     graph.add_node(node, **nodedata)
(4)                 for d in data[link]:
(8)                     src = tuple(d[source]) if isinstance(d[source], list) else d[source]
(8)                     tgt = tuple(d[target]) if isinstance(d[target], list) else d[target]
```

```
186: (8)                    if not multigraph:
187: (12)                       edgedata = {str(k): v for k, v in d.items() if k != source and k
!= target}
188: (12)                       graph.add_edge(src, tgt, **edgedata)
189: (8)                    else:
190: (12)                       ky = d.get(key, None)
191: (12)                       edgedata = {
192: (16)                           str(k): v
193: (16)                           for k, v in d.items()
194: (16)                           if k != source and k != target and k != key
195: (12)                       }
196: (12)                       graph.add_edge(src, tgt, ky, **edgedata)
197: (4)                return graph
```

File 518 - tree.py:

```
1: (0)              from itertools import chain
2: (0)              import networkx as nx
3: (0)              __all__ = ["tree_data", "tree_graph"]
4: (0)              def tree_data(G, root, ident="id", children="children"):
5: (4)                  """Returns data in tree format that is suitable for JSON serialization
6: (4)                  and use in JavaScript documents.
7: (4)                  Parameters
8: (4)                  ----------
9: (4)                  G : NetworkX graph
10: (7)                    G must be an oriented tree
11: (4)                 root : node
12: (7)                    The root of the tree
13: (4)                 ident : string
14: (8)                     Attribute name for storing NetworkX-internal graph data. `ident` must
15: (8)                     have a different value than `children`. The default is 'id'.
16: (4)                 children : string
17: (8)                     Attribute name for storing NetworkX-internal graph data. `children`
18: (8)                     must have a different value than `ident`. The default is 'children'.
19: (4)                 Returns
20: (4)                 -------
21: (4)                 data : dict
22: (7)                    A dictionary with node-link formatted data.
23: (4)                 Raises
24: (4)                 ------
25: (4)                 NetworkXError
26: (8)                     If `children` and `ident` attributes are identical.
27: (4)                 Examples
28: (4)                 --------
29: (4)                 >>> from networkx.readwrite import json_graph
30: (4)                 >>> G = nx.DiGraph([(1, 2)])
31: (4)                 >>> data = json_graph.tree_data(G, root=1)
32: (4)                 To serialize with json
33: (4)                 >>> import json
34: (4)                 >>> s = json.dumps(data)
35: (4)                 Notes
36: (4)                 -----
37: (4)                 Node attributes are stored in this format but keys
38: (4)                 for attributes must be strings if you want to serialize with JSON.
39: (4)                 Graph and edge attributes are not stored.
40: (4)                 See Also
41: (4)                 --------
42: (4)                 tree_graph, node_link_data, adjacency_data
43: (4)                 """
44: (4)                 if G.number_of_nodes() != G.number_of_edges() + 1:
45: (8)                     raise TypeError("G is not a tree.")
46: (4)                 if not G.is_directed():
47: (8)                     raise TypeError("G is not directed.")
48: (4)                 if not nx.is_weakly_connected(G):
49: (8)                     raise TypeError("G is not weakly connected.")
50: (4)                 if ident == children:
51: (8)                     raise nx.NetworkXError("The values for `id` and `children` must be
```

```
different.")
52: (4)                 def add_children(n, G):
53: (8)                     nbrs = G[n]
54: (8)                     if len(nbrs) == 0:
55: (12)                        return []
56: (8)                     children_ = []
57: (8)                     for child in nbrs:
58: (12)                        d = {**G.nodes[child], ident: child}
59: (12)                        c = add_children(child, G)
60: (12)                        if c:
61: (16)                            d[children] = c
62: (12)                        children_.append(d)
63: (8)                     return children_
64: (4)                 return {**G.nodes[root], ident: root, children: add_children(root, G)}
65: (0)             @nx._dispatchable(graphs=None, returns_graph=True)
66: (0)             def tree_graph(data, ident="id", children="children"):
67: (4)                 """Returns graph from tree data format.
68: (4)                 Parameters
69: (4)                 ----------
70: (4)                 data : dict
71: (8)                     Tree formatted graph data
72: (4)                 ident : string
73: (8)                     Attribute name for storing NetworkX-internal graph data. `ident` must
74: (8)                     have a different value than `children`. The default is 'id'.
75: (4)                 children : string
76: (8)                     Attribute name for storing NetworkX-internal graph data. `children`
77: (8)                     must have a different value than `ident`. The default is 'children'.
78: (4)                 Returns
79: (4)                 -------
80: (4)                 G : NetworkX DiGraph
81: (4)                 Examples
82: (4)                 --------
83: (4)                 >>> from networkx.readwrite import json_graph
84: (4)                 >>> G = nx.DiGraph([(1, 2)])
85: (4)                 >>> data = json_graph.tree_data(G, root=1)
86: (4)                 >>> H = json_graph.tree_graph(data)
87: (4)                 See Also
88: (4)                 --------
89: (4)                 tree_data, node_link_data, adjacency_data
90: (4)                 """
91: (4)                 graph = nx.DiGraph()
92: (4)                 def add_children(parent, children_):
93: (8)                     for data in children_:
94: (12)                        child = data[ident]
95: (12)                        graph.add_edge(parent, child)
96: (12)                        grandchildren = data.get(children, [])
97: (12)                        if grandchildren:
98: (16)                            add_children(child, grandchildren)
99: (12)                        nodedata = {
100: (16)                           str(k): v for k, v in data.items() if k != ident and k !=
children
101: (12)                       }
102: (12)                       graph.add_node(child, **nodedata)
103: (4)                root = data[ident]
104: (4)                children_ = data.get(children, [])
105: (4)                nodedata = {str(k): v for k, v in data.items() if k != ident and k !=
children}
106: (4)                graph.add_node(root, **nodedata)
107: (4)                add_children(root, children_)
108: (4)                return graph

----------------------------------------

File 519 - __init__.py:

1: (0)

----------------------------------------
```

File 520 - test_adjacency.py:

```
1: (0)              import copy
2: (0)              import json
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.readwrite.json_graph import adjacency_data, adjacency_graph
6: (0)              from networkx.utils import graphs_equal
7: (0)              class TestAdjacency:
8: (4)                  def test_graph(self):
9: (8)                      G = nx.path_graph(4)
10: (8)                     H = adjacency_graph(adjacency_data(G))
11: (8)                     assert graphs_equal(G, H)
12: (4)                 def test_graph_attributes(self):
13: (8)                     G = nx.path_graph(4)
14: (8)                     G.add_node(1, color="red")
15: (8)                     G.add_edge(1, 2, width=7)
16: (8)                     G.graph["foo"] = "bar"
17: (8)                     G.graph[1] = "one"
18: (8)                     H = adjacency_graph(adjacency_data(G))
19: (8)                     assert graphs_equal(G, H)
20: (8)                     assert H.graph["foo"] == "bar"
21: (8)                     assert H.nodes[1]["color"] == "red"
22: (8)                     assert H[1][2]["width"] == 7
23: (8)                     d = json.dumps(adjacency_data(G))
24: (8)                     H = adjacency_graph(json.loads(d))
25: (8)                     assert graphs_equal(G, H)
26: (8)                     assert H.graph["foo"] == "bar"
27: (8)                     assert H.graph[1] == "one"
28: (8)                     assert H.nodes[1]["color"] == "red"
29: (8)                     assert H[1][2]["width"] == 7
30: (4)                 def test_digraph(self):
31: (8)                     G = nx.DiGraph()
32: (8)                     nx.add_path(G, [1, 2, 3])
33: (8)                     H = adjacency_graph(adjacency_data(G))
34: (8)                     assert H.is_directed()
35: (8)                     assert graphs_equal(G, H)
36: (4)                 def test_multidigraph(self):
37: (8)                     G = nx.MultiDiGraph()
38: (8)                     nx.add_path(G, [1, 2, 3])
39: (8)                     H = adjacency_graph(adjacency_data(G))
40: (8)                     assert H.is_directed()
41: (8)                     assert H.is_multigraph()
42: (8)                     assert graphs_equal(G, H)
43: (4)                 def test_multigraph(self):
44: (8)                     G = nx.MultiGraph()
45: (8)                     G.add_edge(1, 2, key="first")
46: (8)                     G.add_edge(1, 2, key="second", color="blue")
47: (8)                     H = adjacency_graph(adjacency_data(G))
48: (8)                     assert graphs_equal(G, H)
49: (8)                     assert H[1][2]["second"]["color"] == "blue"
50: (4)                 def test_input_data_is_not_modified_when_building_graph(self):
51: (8)                     G = nx.path_graph(4)
52: (8)                     input_data = adjacency_data(G)
53: (8)                     orig_data = copy.deepcopy(input_data)
54: (8)                     assert graphs_equal(G, adjacency_graph(input_data))
55: (8)                     assert input_data == orig_data
56: (4)                 def test_adjacency_form_json_serialisable(self):
57: (8)                     G = nx.path_graph(4)
58: (8)                     H = adjacency_graph(json.loads(json.dumps(adjacency_data(G))))
59: (8)                     assert graphs_equal(G, H)
60: (4)                 def test_exception(self):
61: (8)                     with pytest.raises(nx.NetworkXError):
62: (12)                        G = nx.MultiDiGraph()
63: (12)                        attrs = {"id": "node", "key": "node"}
64: (12)                        adjacency_data(G, attrs)
```

File 521 - test_cytoscape.py:

```
1: (0)              import copy
2: (0)              import json
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.readwrite.json_graph import cytoscape_data, cytoscape_graph
6: (0)              def test_graph():
7: (4)                  G = nx.path_graph(4)
8: (4)                  H = cytoscape_graph(cytoscape_data(G))
9: (4)                  assert nx.is_isomorphic(G, H)
10: (0)             def test_input_data_is_not_modified_when_building_graph():
11: (4)                 G = nx.path_graph(4)
12: (4)                 input_data = cytoscape_data(G)
13: (4)                 orig_data = copy.deepcopy(input_data)
14: (4)                 cytoscape_graph(input_data)
15: (4)                 assert input_data == orig_data
16: (0)             def test_graph_attributes():
17: (4)                 G = nx.path_graph(4)
18: (4)                 G.add_node(1, color="red")
19: (4)                 G.add_edge(1, 2, width=7)
20: (4)                 G.graph["foo"] = "bar"
21: (4)                 G.graph[1] = "one"
22: (4)                 G.add_node(3, name="node", id="123")
23: (4)                 H = cytoscape_graph(cytoscape_data(G))
24: (4)                 assert H.graph["foo"] == "bar"
25: (4)                 assert H.nodes[1]["color"] == "red"
26: (4)                 assert H[1][2]["width"] == 7
27: (4)                 assert H.nodes[3]["name"] == "node"
28: (4)                 assert H.nodes[3]["id"] == "123"
29: (4)                 d = json.dumps(cytoscape_data(G))
30: (4)                 H = cytoscape_graph(json.loads(d))
31: (4)                 assert H.graph["foo"] == "bar"
32: (4)                 assert H.graph[1] == "one"
33: (4)                 assert H.nodes[1]["color"] == "red"
34: (4)                 assert H[1][2]["width"] == 7
35: (4)                 assert H.nodes[3]["name"] == "node"
36: (4)                 assert H.nodes[3]["id"] == "123"
37: (0)             def test_digraph():
38: (4)                 G = nx.DiGraph()
39: (4)                 nx.add_path(G, [1, 2, 3])
40: (4)                 H = cytoscape_graph(cytoscape_data(G))
41: (4)                 assert H.is_directed()
42: (4)                 assert nx.is_isomorphic(G, H)
43: (0)             def test_multidigraph():
44: (4)                 G = nx.MultiDiGraph()
45: (4)                 nx.add_path(G, [1, 2, 3])
46: (4)                 H = cytoscape_graph(cytoscape_data(G))
47: (4)                 assert H.is_directed()
48: (4)                 assert H.is_multigraph()
49: (0)             def test_multigraph():
50: (4)                 G = nx.MultiGraph()
51: (4)                 G.add_edge(1, 2, key="first")
52: (4)                 G.add_edge(1, 2, key="second", color="blue")
53: (4)                 H = cytoscape_graph(cytoscape_data(G))
54: (4)                 assert nx.is_isomorphic(G, H)
55: (4)                 assert H[1][2]["second"]["color"] == "blue"
56: (0)             def test_exception():
57: (4)                 with pytest.raises(nx.NetworkXError):
58: (8)                     G = nx.MultiDiGraph()
59: (8)                     cytoscape_data(G, name="foo", ident="foo")
```

File 522 - test_node_link.py:

```
1: (0)              import json
2: (0)              import pytest
3: (0)              import networkx as nx
```

```
4: (0)              from networkx.readwrite.json_graph import node_link_data, node_link_graph
5: (0)              class TestNodeLink:
6: (4)                  def test_custom_attrs_dep(self):
7: (8)                      G = nx.path_graph(4)
8: (8)                      G.add_node(1, color="red")
9: (8)                      G.add_edge(1, 2, width=7)
10: (8)                     G.graph[1] = "one"
11: (8)                     G.graph["foo"] = "bar"
12: (8)                     attrs = {
13: (12)                        "source": "c_source",
14: (12)                        "target": "c_target",
15: (12)                        "name": "c_id",
16: (12)                        "key": "c_key",
17: (12)                        "link": "c_links",
18: (8)                     }
19: (8)                     H = node_link_graph(node_link_data(G, **attrs), multigraph=False,
**attrs)
20: (8)                     assert nx.is_isomorphic(G, H)
21: (8)                     assert H.graph["foo"] == "bar"
22: (8)                     assert H.nodes[1]["color"] == "red"
23: (8)                     assert H[1][2]["width"] == 7
24: (8)                     attrs = {
25: (12)                        "link": "c_links",
26: (12)                        "source": "c_source",
27: (12)                        "target": "c_target",
28: (8)                     }
29: (8)                     H = node_link_graph(node_link_data(G, **attrs), multigraph=False,
**attrs)
30: (8)                     assert nx.is_isomorphic(G, H)
31: (8)                     assert H.graph["foo"] == "bar"
32: (8)                     assert H.nodes[1]["color"] == "red"
33: (8)                     assert H[1][2]["width"] == 7
34: (4)                 def test_exception_dep(self):
35: (8)                     with pytest.raises(nx.NetworkXError):
36: (12)                        G = nx.MultiDiGraph()
37: (12)                        node_link_data(G, name="node", source="node", target="node",
key="node")
38: (4)                 def test_graph(self):
39: (8)                     G = nx.path_graph(4)
40: (8)                     H = node_link_graph(node_link_data(G))
41: (8)                     assert nx.is_isomorphic(G, H)
42: (4)                 def test_graph_attributes(self):
43: (8)                     G = nx.path_graph(4)
44: (8)                     G.add_node(1, color="red")
45: (8)                     G.add_edge(1, 2, width=7)
46: (8)                     G.graph[1] = "one"
47: (8)                     G.graph["foo"] = "bar"
48: (8)                     H = node_link_graph(node_link_data(G))
49: (8)                     assert H.graph["foo"] == "bar"
50: (8)                     assert H.nodes[1]["color"] == "red"
51: (8)                     assert H[1][2]["width"] == 7
52: (8)                     d = json.dumps(node_link_data(G))
53: (8)                     H = node_link_graph(json.loads(d))
54: (8)                     assert H.graph["foo"] == "bar"
55: (8)                     assert H.graph["1"] == "one"
56: (8)                     assert H.nodes[1]["color"] == "red"
57: (8)                     assert H[1][2]["width"] == 7
58: (4)                 def test_digraph(self):
59: (8)                     G = nx.DiGraph()
60: (8)                     H = node_link_graph(node_link_data(G))
61: (8)                     assert H.is_directed()
62: (4)                 def test_multigraph(self):
63: (8)                     G = nx.MultiGraph()
64: (8)                     G.add_edge(1, 2, key="first")
65: (8)                     G.add_edge(1, 2, key="second", color="blue")
66: (8)                     H = node_link_graph(node_link_data(G))
67: (8)                     assert nx.is_isomorphic(G, H)
68: (8)                     assert H[1][2]["second"]["color"] == "blue"
69: (4)                 def test_graph_with_tuple_nodes(self):
```

```
70: (8)                     G = nx.Graph()
71: (8)                     G.add_edge((0, 0), (1, 0), color=[255, 255, 0])
72: (8)                     d = node_link_data(G)
73: (8)                     dumped_d = json.dumps(d)
74: (8)                     dd = json.loads(dumped_d)
75: (8)                     H = node_link_graph(dd)
76: (8)                     assert H.nodes[(0, 0)] == G.nodes[(0, 0)]
77: (8)                     assert H[(0, 0)][(1, 0)]["color"] == [255, 255, 0]
78: (4)                 def test_unicode_keys(self):
79: (8)                     q = "qualité"
80: (8)                     G = nx.Graph()
81: (8)                     G.add_node(1, **{q: q})
82: (8)                     s = node_link_data(G)
83: (8)                     output = json.dumps(s, ensure_ascii=False)
84: (8)                     data = json.loads(output)
85: (8)                     H = node_link_graph(data)
86: (8)                     assert H.nodes[1][q] == q
87: (4)                 def test_exception(self):
88: (8)                     with pytest.raises(nx.NetworkXError):
89: (12)                        G = nx.MultiDiGraph()
90: (12)                        attrs = {"name": "node", "source": "node", "target": "node",
"key": "node"}
91: (12)                        node_link_data(G, **attrs)
92: (4)                 def test_string_ids(self):
93: (8)                     q = "qualité"
94: (8)                     G = nx.DiGraph()
95: (8)                     G.add_node("A")
96: (8)                     G.add_node(q)
97: (8)                     G.add_edge("A", q)
98: (8)                     data = node_link_data(G)
99: (8)                     assert data["links"][0]["source"] == "A"
100: (8)                    assert data["links"][0]["target"] == q
101: (8)                    H = node_link_graph(data)
102: (8)                    assert nx.is_isomorphic(G, H)
103: (4)                def test_custom_attrs(self):
104: (8)                    G = nx.path_graph(4)
105: (8)                    G.add_node(1, color="red")
106: (8)                    G.add_edge(1, 2, width=7)
107: (8)                    G.graph[1] = "one"
108: (8)                    G.graph["foo"] = "bar"
109: (8)                    attrs = {
110: (12)                       "source": "c_source",
111: (12)                       "target": "c_target",
112: (12)                       "name": "c_id",
113: (12)                       "key": "c_key",
114: (12)                       "link": "c_links",
115: (8)                    }
116: (8)                    H = node_link_graph(node_link_data(G, **attrs), multigraph=False,
**attrs)
117: (8)                    assert nx.is_isomorphic(G, H)
118: (8)                    assert H.graph["foo"] == "bar"
119: (8)                    assert H.nodes[1]["color"] == "red"
120: (8)                    assert H[1][2]["width"] == 7
```

File 523 - test_tree.py:

```
1: (0)              import json
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.readwrite.json_graph import tree_data, tree_graph
5: (0)              def test_graph():
6: (4)                  G = nx.DiGraph()
7: (4)                  G.add_nodes_from([1, 2, 3], color="red")
8: (4)                  G.add_edge(1, 2, foo=7)
9: (4)                  G.add_edge(1, 3, foo=10)
10: (4)                 G.add_edge(3, 4, foo=10)
11: (4)                 H = tree_graph(tree_data(G, 1))
```

```
12: (4)                 assert nx.is_isomorphic(G, H)
13: (0)             def test_graph_attributes():
14: (4)                 G = nx.DiGraph()
15: (4)                 G.add_nodes_from([1, 2, 3], color="red")
16: (4)                 G.add_edge(1, 2, foo=7)
17: (4)                 G.add_edge(1, 3, foo=10)
18: (4)                 G.add_edge(3, 4, foo=10)
19: (4)                 H = tree_graph(tree_data(G, 1))
20: (4)                 assert H.nodes[1]["color"] == "red"
21: (4)                 d = json.dumps(tree_data(G, 1))
22: (4)                 H = tree_graph(json.loads(d))
23: (4)                 assert H.nodes[1]["color"] == "red"
24: (0)             def test_exceptions():
25: (4)                 with pytest.raises(TypeError, match="is not a tree."):
26: (8)                     G = nx.complete_graph(3)
27: (8)                     tree_data(G, 0)
28: (4)                 with pytest.raises(TypeError, match="is not directed."):
29: (8)                     G = nx.path_graph(3)
30: (8)                     tree_data(G, 0)
31: (4)                 with pytest.raises(TypeError, match="is not weakly connected."):
32: (8)                     G = nx.path_graph(3, create_using=nx.DiGraph)
33: (8)                     G.add_edge(2, 0)
34: (8)                     G.add_node(3)
35: (8)                     tree_data(G, 0)
36: (4)                 with pytest.raises(nx.NetworkXError, match="must be different."):
37: (8)                     G = nx.MultiDiGraph()
38: (8)                     G.add_node(0)
39: (8)                     tree_data(G, 0, ident="node", children="node")

----------------------------------------

File 524 - __init__.py:

1: (0)

----------------------------------------

File 525 - test_adjlist.py:

1: (0)              """
2: (4)                  Unit tests for adjlist.
3: (0)              """
4: (0)              import io
5: (0)              import pytest
6: (0)              import networkx as nx
7: (0)              from networkx.utils import edges_equal, graphs_equal, nodes_equal
8: (0)              class TestAdjlist:
9: (4)                  @classmethod
10: (4)                 def setup_class(cls):
11: (8)                     cls.G = nx.Graph(name="test")
12: (8)                     e = [("a", "b"), ("b", "c"), ("c", "d"), ("d", "e"), ("e", "f"), ("a",
"f")]
13: (8)                     cls.G.add_edges_from(e)
14: (8)                     cls.G.add_node("g")
15: (8)                     cls.DG = nx.DiGraph(cls.G)
16: (8)                     cls.XG = nx.MultiGraph()
17: (8)                     cls.XG.add_weighted_edges_from([(1, 2, 5), (1, 2, 5), (1, 2, 1), (3,
3, 42)])
18: (8)                     cls.XDG = nx.MultiDiGraph(cls.XG)
19: (4)                 def test_read_multiline_adjlist_1(self):
20: (8)                     s = b"""# comment line
21: (0)             1 2
22: (0)             2
23: (0)             3
24: (0)             """
25: (8)                     bytesIO = io.BytesIO(s)
26: (8)                     G = nx.read_multiline_adjlist(bytesIO)
27: (8)                     adj = {"1": {"3": {}, "2": {}}, "3": {"1": {}}, "2": {"1": {}}}
28: (8)                     assert graphs_equal(G, nx.Graph(adj))
```

```
 29: (4)                 def test_unicode(self, tmp_path):
 30: (8)                     G = nx.Graph()
 31: (8)                     name1 = chr(2344) + chr(123) + chr(6543)
 32: (8)                     name2 = chr(5543) + chr(1543) + chr(324)
 33: (8)                     G.add_edge(name1, "Radiohead", **{name2: 3})
 34: (8)                     fname = tmp_path / "adjlist.txt"
 35: (8)                     nx.write_multiline_adjlist(G, fname)
 36: (8)                     H = nx.read_multiline_adjlist(fname)
 37: (8)                     assert graphs_equal(G, H)
 38: (4)                 def test_latin1_err(self, tmp_path):
 39: (8)                     G = nx.Graph()
 40: (8)                     name1 = chr(2344) + chr(123) + chr(6543)
 41: (8)                     name2 = chr(5543) + chr(1543) + chr(324)
 42: (8)                     G.add_edge(name1, "Radiohead", **{name2: 3})
 43: (8)                     fname = tmp_path / "adjlist.txt"
 44: (8)                     with pytest.raises(UnicodeEncodeError):
 45: (12)                        nx.write_multiline_adjlist(G, fname, encoding="latin-1")
 46: (4)                 def test_latin1(self, tmp_path):
 47: (8)                     G = nx.Graph()
 48: (8)                     name1 = "Bj" + chr(246) + "rk"
 49: (8)                     name2 = chr(220) + "ber"
 50: (8)                     G.add_edge(name1, "Radiohead", **{name2: 3})
 51: (8)                     fname = tmp_path / "adjlist.txt"
 52: (8)                     nx.write_multiline_adjlist(G, fname, encoding="latin-1")
 53: (8)                     H = nx.read_multiline_adjlist(fname, encoding="latin-1")
 54: (8)                     assert graphs_equal(G, H)
 55: (4)                 def test_parse_adjlist(self):
 56: (8)                     lines = ["1 2 5", "2 3 4", "3 5", "4", "5"]
 57: (8)                     nx.parse_adjlist(lines, nodetype=int)  # smoke test
 58: (8)                     with pytest.raises(TypeError):
 59: (12)                        nx.parse_adjlist(lines, nodetype="int")
 60: (8)                     lines = ["1 2 5", "2 b", "c"]
 61: (8)                     with pytest.raises(TypeError):
 62: (12)                        nx.parse_adjlist(lines, nodetype=int)
 63: (4)                 def test_adjlist_graph(self, tmp_path):
 64: (8)                     G = self.G
 65: (8)                     fname = tmp_path / "adjlist.txt"
 66: (8)                     nx.write_adjlist(G, fname)
 67: (8)                     H = nx.read_adjlist(fname)
 68: (8)                     H2 = nx.read_adjlist(fname)
 69: (8)                     assert H is not H2  # they should be different graphs
 70: (8)                     assert nodes_equal(list(H), list(G))
 71: (8)                     assert edges_equal(list(H.edges()), list(G.edges()))
 72: (4)                 def test_adjlist_digraph(self, tmp_path):
 73: (8)                     G = self.DG
 74: (8)                     fname = tmp_path / "adjlist.txt"
 75: (8)                     nx.write_adjlist(G, fname)
 76: (8)                     H = nx.read_adjlist(fname, create_using=nx.DiGraph())
 77: (8)                     H2 = nx.read_adjlist(fname, create_using=nx.DiGraph())
 78: (8)                     assert H is not H2  # they should be different graphs
 79: (8)                     assert nodes_equal(list(H), list(G))
 80: (8)                     assert edges_equal(list(H.edges()), list(G.edges()))
 81: (4)                 def test_adjlist_integers(self, tmp_path):
 82: (8)                     fname = tmp_path / "adjlist.txt"
 83: (8)                     G = nx.convert_node_labels_to_integers(self.G)
 84: (8)                     nx.write_adjlist(G, fname)
 85: (8)                     H = nx.read_adjlist(fname, nodetype=int)
 86: (8)                     H2 = nx.read_adjlist(fname, nodetype=int)
 87: (8)                     assert H is not H2  # they should be different graphs
 88: (8)                     assert nodes_equal(list(H), list(G))
 89: (8)                     assert edges_equal(list(H.edges()), list(G.edges()))
 90: (4)                 def test_adjlist_multigraph(self, tmp_path):
 91: (8)                     G = self.XG
 92: (8)                     fname = tmp_path / "adjlist.txt"
 93: (8)                     nx.write_adjlist(G, fname)
 94: (8)                     H = nx.read_adjlist(fname, nodetype=int, create_using=nx.MultiGraph())
 95: (8)                     H2 = nx.read_adjlist(fname, nodetype=int,
create_using=nx.MultiGraph())
 96: (8)                     assert H is not H2  # they should be different graphs
```

```
 97: (8)                     assert nodes_equal(list(H), list(G))
 98: (8)                     assert edges_equal(list(H.edges()), list(G.edges()))
 99: (4)                 def test_adjlist_multidigraph(self, tmp_path):
 100: (8)                    G = self.XDG
 101: (8)                    fname = tmp_path / "adjlist.txt"
 102: (8)                    nx.write_adjlist(G, fname)
 103: (8)                    H = nx.read_adjlist(fname, nodetype=int,
 create_using=nx.MultiDiGraph())
 104: (8)                    H2 = nx.read_adjlist(fname, nodetype=int,
 create_using=nx.MultiDiGraph())
 105: (8)                    assert H is not H2  # they should be different graphs
 106: (8)                    assert nodes_equal(list(H), list(G))
 107: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
 108: (4)                def test_adjlist_delimiter(self):
 109: (8)                    fh = io.BytesIO()
 110: (8)                    G = nx.path_graph(3)
 111: (8)                    nx.write_adjlist(G, fh, delimiter=":")
 112: (8)                    fh.seek(0)
 113: (8)                    H = nx.read_adjlist(fh, nodetype=int, delimiter=":")
 114: (8)                    assert nodes_equal(list(H), list(G))
 115: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
 116: (0)            class TestMultilineAdjlist:
 117: (4)                @classmethod
 118: (4)                def setup_class(cls):
 119: (8)                    cls.G = nx.Graph(name="test")
 120: (8)                    e = [("a", "b"), ("b", "c"), ("c", "d"), ("d", "e"), ("e", "f"), ("a",
 "f")]
 121: (8)                    cls.G.add_edges_from(e)
 122: (8)                    cls.G.add_node("g")
 123: (8)                    cls.DG = nx.DiGraph(cls.G)
 124: (8)                    cls.DG.remove_edge("b", "a")
 125: (8)                    cls.DG.remove_edge("b", "c")
 126: (8)                    cls.XG = nx.MultiGraph()
 127: (8)                    cls.XG.add_weighted_edges_from([(1, 2, 5), (1, 2, 5), (1, 2, 1), (3,
 3, 42)])
 128: (8)                    cls.XDG = nx.MultiDiGraph(cls.XG)
 129: (4)                def test_parse_multiline_adjlist(self):
 130: (8)                    lines = [
 131: (12)                       "1 2",
 132: (12)                       "b {'weight':3, 'name': 'Frodo'}",
 133: (12)                       "c {}",
 134: (12)                       "d 1",
 135: (12)                       "e {'weight':6, 'name': 'Saruman'}",
 136: (8)                    ]
 137: (8)                    nx.parse_multiline_adjlist(iter(lines))  # smoke test
 138: (8)                    with pytest.raises(TypeError):
 139: (12)                       nx.parse_multiline_adjlist(iter(lines), nodetype=int)
 140: (8)                    nx.parse_multiline_adjlist(iter(lines), edgetype=str)  # smoke test
 141: (8)                    with pytest.raises(TypeError):
 142: (12)                       nx.parse_multiline_adjlist(iter(lines), nodetype=int)
 143: (8)                    lines = ["1 a"]
 144: (8)                    with pytest.raises(TypeError):
 145: (12)                       nx.parse_multiline_adjlist(iter(lines))
 146: (8)                    lines = ["a 2"]
 147: (8)                    with pytest.raises(TypeError):
 148: (12)                       nx.parse_multiline_adjlist(iter(lines), nodetype=int)
 149: (8)                    lines = ["1 2"]
 150: (8)                    with pytest.raises(TypeError):
 151: (12)                       nx.parse_multiline_adjlist(iter(lines))
 152: (8)                    lines = ["1 2", "2 {}"]
 153: (8)                    with pytest.raises(TypeError):
 154: (12)                       nx.parse_multiline_adjlist(iter(lines))
 155: (4)                def test_multiline_adjlist_graph(self, tmp_path):
 156: (8)                    G = self.G
 157: (8)                    fname = tmp_path / "adjlist.txt"
 158: (8)                    nx.write_multiline_adjlist(G, fname)
 159: (8)                    H = nx.read_multiline_adjlist(fname)
 160: (8)                    H2 = nx.read_multiline_adjlist(fname)
 161: (8)                    assert H is not H2  # they should be different graphs
```

```
162: (8)                    assert nodes_equal(list(H), list(G))
163: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
164: (4)                def test_multiline_adjlist_digraph(self, tmp_path):
165: (8)                    G = self.DG
166: (8)                    fname = tmp_path / "adjlist.txt"
167: (8)                    nx.write_multiline_adjlist(G, fname)
168: (8)                    H = nx.read_multiline_adjlist(fname, create_using=nx.DiGraph())
169: (8)                    H2 = nx.read_multiline_adjlist(fname, create_using=nx.DiGraph())
170: (8)                    assert H is not H2  # they should be different graphs
171: (8)                    assert nodes_equal(list(H), list(G))
172: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
173: (4)                def test_multiline_adjlist_integers(self, tmp_path):
174: (8)                    fname = tmp_path / "adjlist.txt"
175: (8)                    G = nx.convert_node_labels_to_integers(self.G)
176: (8)                    nx.write_multiline_adjlist(G, fname)
177: (8)                    H = nx.read_multiline_adjlist(fname, nodetype=int)
178: (8)                    H2 = nx.read_multiline_adjlist(fname, nodetype=int)
179: (8)                    assert H is not H2  # they should be different graphs
180: (8)                    assert nodes_equal(list(H), list(G))
181: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
182: (4)                def test_multiline_adjlist_multigraph(self, tmp_path):
183: (8)                    G = self.XG
184: (8)                    fname = tmp_path / "adjlist.txt"
185: (8)                    nx.write_multiline_adjlist(G, fname)
186: (8)                    H = nx.read_multiline_adjlist(fname, nodetype=int,
create_using=nx.MultiGraph())
187: (8)                    H2 = nx.read_multiline_adjlist(
188: (12)                       fname, nodetype=int, create_using=nx.MultiGraph()
189: (8)                    )
190: (8)                    assert H is not H2  # they should be different graphs
191: (8)                    assert nodes_equal(list(H), list(G))
192: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
193: (4)                def test_multiline_adjlist_multidigraph(self, tmp_path):
194: (8)                    G = self.XDG
195: (8)                    fname = tmp_path / "adjlist.txt"
196: (8)                    nx.write_multiline_adjlist(G, fname)
197: (8)                    H = nx.read_multiline_adjlist(
198: (12)                       fname, nodetype=int, create_using=nx.MultiDiGraph()
199: (8)                    )
200: (8)                    H2 = nx.read_multiline_adjlist(
201: (12)                       fname, nodetype=int, create_using=nx.MultiDiGraph()
202: (8)                    )
203: (8)                    assert H is not H2  # they should be different graphs
204: (8)                    assert nodes_equal(list(H), list(G))
205: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
206: (4)                def test_multiline_adjlist_delimiter(self):
207: (8)                    fh = io.BytesIO()
208: (8)                    G = nx.path_graph(3)
209: (8)                    nx.write_multiline_adjlist(G, fh, delimiter=":")
210: (8)                    fh.seek(0)
211: (8)                    H = nx.read_multiline_adjlist(fh, nodetype=int, delimiter=":")
212: (8)                    assert nodes_equal(list(H), list(G))
213: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))

----------------------------------------

File 526 - test_edgelist.py:

1: (0)              """
2: (4)                  Unit tests for edgelists.
3: (0)              """
4: (0)              import io
5: (0)              import textwrap
6: (0)              import pytest
7: (0)              import networkx as nx
8: (0)              from networkx.utils import edges_equal, graphs_equal, nodes_equal
9: (0)              edges_no_data = textwrap.dedent(
10: (4)                 """
11: (4)                 1 2
```

```
(4)                 2 3
(4)                 """
(0)             )
(0)             edges_with_values = textwrap.dedent(
(4)                 """
(4)                 1 2 2.0
(4)                 2 3 3.0
(4)                 """
(0)             )
(0)             edges_with_weight = textwrap.dedent(
(4)                 """
(4)                 1 2 {'weight':2.0}
(4)                 2 3 {'weight':3.0}
(4)                 """
(0)             )
(0)             edges_with_multiple_attrs = textwrap.dedent(
(4)                 """
(4)                 1 2 {'weight':2.0, 'color':'green'}
(4)                 2 3 {'weight':3.0, 'color':'red'}
(4)                 """
(0)             )
(0)             edges_with_multiple_attrs_csv = textwrap.dedent(
(4)                 """
(4)                 1, 2, {'weight':2.0, 'color':'green'}
(4)                 2, 3, {'weight':3.0, 'color':'red'}
(4)                 """
(0)             )
(0)             _expected_edges_weights = [(1, 2, {"weight": 2.0}), (2, 3, {"weight": 3.0})]
(0)             _expected_edges_multiattr = [
(4)                 (1, 2, {"weight": 2.0, "color": "green"}),
(4)                 (2, 3, {"weight": 3.0, "color": "red"}),
(0)             ]
(0)             @pytest.mark.parametrize(
(4)                 ("data", "extra_kwargs"),
(4)                 (
(8)                     (edges_no_data, {}),
(8)                     (edges_with_values, {}),
(8)                     (edges_with_weight, {}),
(8)                     (edges_with_multiple_attrs, {}),
(8)                     (edges_with_multiple_attrs_csv, {"delimiter": ","}),
(4)                 ),
(0)             )
(0)             def test_read_edgelist_no_data(data, extra_kwargs):
(4)                 bytesIO = io.BytesIO(data.encode("utf-8"))
(4)                 G = nx.read_edgelist(bytesIO, nodetype=int, data=False, **extra_kwargs)
(4)                 assert edges_equal(G.edges(), [(1, 2), (2, 3)])
(0)             def test_read_weighted_edgelist():
(4)                 bytesIO = io.BytesIO(edges_with_values.encode("utf-8"))
(4)                 G = nx.read_weighted_edgelist(bytesIO, nodetype=int)
(4)                 assert edges_equal(G.edges(data=True), _expected_edges_weights)
(0)             @pytest.mark.parametrize(
(4)                 ("data", "extra_kwargs", "expected"),
(4)                 (
(8)                     (edges_with_weight, {}, _expected_edges_weights),
(8)                     (edges_with_multiple_attrs, {}, _expected_edges_multiattr),
(8)                     (edges_with_multiple_attrs_csv, {"delimiter": ","},
_expected_edges_multiattr),
(4)                 ),
(0)             )
(0)             def test_read_edgelist_with_data(data, extra_kwargs, expected):
(4)                 bytesIO = io.BytesIO(data.encode("utf-8"))
(4)                 G = nx.read_edgelist(bytesIO, nodetype=int, **extra_kwargs)
(4)                 assert edges_equal(G.edges(data=True), expected)
(0)             @pytest.fixture
(0)             def example_graph():
(4)                 G = nx.Graph()
(4)                 G.add_weighted_edges_from([(1, 2, 3.0), (2, 3, 27.0), (3, 4, 3.0)])
(4)                 return G
(0)             def test_parse_edgelist_no_data(example_graph):
```

```
80: (4)                 G = example_graph
81: (4)                 H = nx.parse_edgelist(["1 2", "2 3", "3 4"], nodetype=int)
82: (4)                 assert nodes_equal(G.nodes, H.nodes)
83: (4)                 assert edges_equal(G.edges, H.edges)
84: (0)             def test_parse_edgelist_with_data_dict(example_graph):
85: (4)                 G = example_graph
86: (4)                 H = nx.parse_edgelist(
87: (8)                     ["1 2 {'weight': 3}", "2 3 {'weight': 27}", "3 4 {'weight': 3.0}"],
nodetype=int
88: (4)                 )
89: (4)                 assert nodes_equal(G.nodes, H.nodes)
90: (4)                 assert edges_equal(G.edges(data=True), H.edges(data=True))
91: (0)             def test_parse_edgelist_with_data_list(example_graph):
92: (4)                 G = example_graph
93: (4)                 H = nx.parse_edgelist(
94: (8)                     ["1 2 3", "2 3 27", "3 4 3.0"], nodetype=int, data=(("weight",
float),)
95: (4)                 )
96: (4)                 assert nodes_equal(G.nodes, H.nodes)
97: (4)                 assert edges_equal(G.edges(data=True), H.edges(data=True))
98: (0)             def test_parse_edgelist():
99: (4)                 lines = ["1;2", "2 3", "3 4"]
100: (4)                G = nx.parse_edgelist(lines, nodetype=int)
101: (4)                assert list(G.edges()) == [(2, 3), (3, 4)]
102: (4)                with pytest.raises(TypeError, match="Failed to convert nodes"):
103: (8)                    lines = ["1 2", "2 3", "3 4"]
104: (8)                    nx.parse_edgelist(lines, nodetype="nope")
105: (4)                with pytest.raises(TypeError, match="Failed to convert edge data"):
106: (8)                    lines = ["1 2 3", "2 3", "3 4"]
107: (8)                    nx.parse_edgelist(lines, nodetype=int)
108: (4)                with pytest.raises(IndexError, match="not the same length"):
109: (8)                    lines = ["1 2 3", "2 3 27", "3 4 3.0"]
110: (8)                    nx.parse_edgelist(
111: (12)                       lines, nodetype=int, data=(("weight", float), ("capacity", int))
112: (8)                    )
113: (4)                with pytest.raises(TypeError, match="Failed to convert"):
114: (8)                    lines = ["1 2 't1'", "2 3 't3'", "3 4 't3'"]
115: (8)                    nx.parse_edgelist(lines, nodetype=int, data=(("weight", float),))
116: (0)            def test_comments_None():
117: (4)                edgelist = ["node#1 node#2", "node#2 node#3"]
118: (4)                G = nx.parse_edgelist(edgelist, comments=None)
119: (4)                H = nx.Graph([e.split(" ") for e in edgelist])
120: (4)                assert edges_equal(G.edges, H.edges)
121: (0)            class TestEdgelist:
122: (4)                @classmethod
123: (4)                def setup_class(cls):
124: (8)                    cls.G = nx.Graph(name="test")
125: (8)                    e = [("a", "b"), ("b", "c"), ("c", "d"), ("d", "e"), ("e", "f"), ("a",
"f")]
126: (8)                    cls.G.add_edges_from(e)
127: (8)                    cls.G.add_node("g")
128: (8)                    cls.DG = nx.DiGraph(cls.G)
129: (8)                    cls.XG = nx.MultiGraph()
130: (8)                    cls.XG.add_weighted_edges_from([(1, 2, 5), (1, 2, 5), (1, 2, 1), (3,
3, 42)])
131: (8)                    cls.XDG = nx.MultiDiGraph(cls.XG)
132: (4)                def test_write_edgelist_1(self):
133: (8)                    fh = io.BytesIO()
134: (8)                    G = nx.Graph()
135: (8)                    G.add_edges_from([(1, 2), (2, 3)])
136: (8)                    nx.write_edgelist(G, fh, data=False)
137: (8)                    fh.seek(0)
138: (8)                    assert fh.read() == b"1 2\n2 3\n"
139: (4)                def test_write_edgelist_2(self):
140: (8)                    fh = io.BytesIO()
141: (8)                    G = nx.Graph()
142: (8)                    G.add_edges_from([(1, 2), (2, 3)])
143: (8)                    nx.write_edgelist(G, fh, data=True)
144: (8)                    fh.seek(0)
```

```
(8)         assert fh.read() == b"1 2 {}\n2 3 {}\n"
(4)     def test_write_edgelist_3(self):
(8)         fh = io.BytesIO()
(8)         G = nx.Graph()
(8)         G.add_edge(1, 2, weight=2.0)
(8)         G.add_edge(2, 3, weight=3.0)
(8)         nx.write_edgelist(G, fh, data=True)
(8)         fh.seek(0)
(8)         assert fh.read() == b"1 2 {'weight': 2.0}\n2 3 {'weight': 3.0}\n"
(4)     def test_write_edgelist_4(self):
(8)         fh = io.BytesIO()
(8)         G = nx.Graph()
(8)         G.add_edge(1, 2, weight=2.0)
(8)         G.add_edge(2, 3, weight=3.0)
(8)         nx.write_edgelist(G, fh, data=[("weight")])
(8)         fh.seek(0)
(8)         assert fh.read() == b"1 2 2.0\n2 3 3.0\n"
(4)     def test_unicode(self, tmp_path):
(8)         G = nx.Graph()
(8)         name1 = chr(2344) + chr(123) + chr(6543)
(8)         name2 = chr(5543) + chr(1543) + chr(324)
(8)         G.add_edge(name1, "Radiohead", **{name2: 3})
(8)         fname = tmp_path / "el.txt"
(8)         nx.write_edgelist(G, fname)
(8)         H = nx.read_edgelist(fname)
(8)         assert graphs_equal(G, H)
(4)     def test_latin1_issue(self, tmp_path):
(8)         G = nx.Graph()
(8)         name1 = chr(2344) + chr(123) + chr(6543)
(8)         name2 = chr(5543) + chr(1543) + chr(324)
(8)         G.add_edge(name1, "Radiohead", **{name2: 3})
(8)         fname = tmp_path / "el.txt"
(8)         with pytest.raises(UnicodeEncodeError):
(12)            nx.write_edgelist(G, fname, encoding="latin-1")
(4)     def test_latin1(self, tmp_path):
(8)         G = nx.Graph()
(8)         name1 = "Bj" + chr(246) + "rk"
(8)         name2 = chr(220) + "ber"
(8)         G.add_edge(name1, "Radiohead", **{name2: 3})
(8)         fname = tmp_path / "el.txt"
(8)         nx.write_edgelist(G, fname, encoding="latin-1")
(8)         H = nx.read_edgelist(fname, encoding="latin-1")
(8)         assert graphs_equal(G, H)
(4)     def test_edgelist_graph(self, tmp_path):
(8)         G = self.G
(8)         fname = tmp_path / "el.txt"
(8)         nx.write_edgelist(G, fname)
(8)         H = nx.read_edgelist(fname)
(8)         H2 = nx.read_edgelist(fname)
(8)         assert H is not H2  # they should be different graphs
(8)         G.remove_node("g")  # isolated nodes are not written in edgelist
(8)         assert nodes_equal(list(H), list(G))
(8)         assert edges_equal(list(H.edges()), list(G.edges()))
(4)     def test_edgelist_digraph(self, tmp_path):
(8)         G = self.DG
(8)         fname = tmp_path / "el.txt"
(8)         nx.write_edgelist(G, fname)
(8)         H = nx.read_edgelist(fname, create_using=nx.DiGraph())
(8)         H2 = nx.read_edgelist(fname, create_using=nx.DiGraph())
(8)         assert H is not H2  # they should be different graphs
(8)         G.remove_node("g")  # isolated nodes are not written in edgelist
(8)         assert nodes_equal(list(H), list(G))
(8)         assert edges_equal(list(H.edges()), list(G.edges()))
(4)     def test_edgelist_integers(self, tmp_path):
(8)         G = nx.convert_node_labels_to_integers(self.G)
(8)         fname = tmp_path / "el.txt"
(8)         nx.write_edgelist(G, fname)
(8)         H = nx.read_edgelist(fname, nodetype=int)
(8)         G.remove_nodes_from(list(nx.isolates(G)))
```

```
214: (8)                    assert nodes_equal(list(H), list(G))
215: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
216: (4)                def test_edgelist_multigraph(self, tmp_path):
217: (8)                    G = self.XG
218: (8)                    fname = tmp_path / "el.txt"
219: (8)                    nx.write_edgelist(G, fname)
220: (8)                    H = nx.read_edgelist(fname, nodetype=int,
create_using=nx.MultiGraph())
221: (8)                    H2 = nx.read_edgelist(fname, nodetype=int,
create_using=nx.MultiGraph())
222: (8)                    assert H is not H2  # they should be different graphs
223: (8)                    assert nodes_equal(list(H), list(G))
224: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))
225: (4)                def test_edgelist_multidigraph(self, tmp_path):
226: (8)                    G = self.XDG
227: (8)                    fname = tmp_path / "el.txt"
228: (8)                    nx.write_edgelist(G, fname)
229: (8)                    H = nx.read_edgelist(fname, nodetype=int,
create_using=nx.MultiDiGraph())
230: (8)                    H2 = nx.read_edgelist(fname, nodetype=int,
create_using=nx.MultiDiGraph())
231: (8)                    assert H is not H2  # they should be different graphs
232: (8)                    assert nodes_equal(list(H), list(G))
233: (8)                    assert edges_equal(list(H.edges()), list(G.edges()))

----------------------------------------

File 527 - test_gexf.py:

1: (0)              import io
2: (0)              import time
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              class TestGEXF:
6: (4)                  @classmethod
7: (4)                  def setup_class(cls):
8: (8)                      cls.simple_directed_data = """<?xml version="1.0" encoding="UTF-8"?>
9: (0)              <gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
10: (4)                 <graph mode="static" defaultedgetype="directed">
11: (8)                     <nodes>
12: (12)                        <node id="0" label="Hello" />
13: (12)                        <node id="1" label="Word" />
14: (8)                     </nodes>
15: (8)                     <edges>
16: (12)                        <edge id="0" source="0" target="1" />
17: (8)                     </edges>
18: (4)                 </graph>
19: (0)             </gexf>
20: (0)             """
21: (8)                     cls.simple_directed_graph = nx.DiGraph()
22: (8)                     cls.simple_directed_graph.add_node("0", label="Hello")
23: (8)                     cls.simple_directed_graph.add_node("1", label="World")
24: (8)                     cls.simple_directed_graph.add_edge("0", "1", id="0")
25: (8)                     cls.simple_directed_fh =
io.BytesIO(cls.simple_directed_data.encode("UTF-8"))
26: (8)                     cls.attribute_data = """<?xml version="1.0" encoding="UTF-8"?>\
27: (0)             <gexf xmlns="http://www.gexf.net/1.2draft" xmlns:xsi="http://www.w3.\
28: (0)             org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.gexf.net/\
29: (0)             1.2draft http://www.gexf.net/1.2draft/gexf.xsd" version="1.2">
30: (2)               <meta lastmodifieddate="2009-03-20">
31: (4)                 <creator>Gephi.org</creator>
32: (4)                 <description>A Web network</description>
33: (2)               </meta>
34: (2)               <graph defaultedgetype="directed">
35: (4)                 <attributes class="node">
36: (6)                   <attribute id="0" title="url" type="string"/>
37: (6)                   <attribute id="1" title="indegree" type="integer"/>
38: (6)                   <attribute id="2" title="frog" type="boolean">
39: (8)                     <default>true</default>
```

```
  40: (6)                    </attribute>
  41: (4)                  </attributes>
  42: (4)                  <nodes>
  43: (6)                    <node id="0" label="Gephi">
  44: (8)                      <attvalues>
  45: (10)                       <attvalue for="0" value="https://gephi.org"/>
  46: (10)                       <attvalue for="1" value="1"/>
  47: (10)                       <attvalue for="2" value="false"/>
  48: (8)                      </attvalues>
  49: (6)                    </node>
  50: (6)                    <node id="1" label="Webatlas">
  51: (8)                      <attvalues>
  52: (10)                       <attvalue for="0" value="http://webatlas.fr"/>
  53: (10)                       <attvalue for="1" value="2"/>
  54: (10)                       <attvalue for="2" value="false"/>
  55: (8)                      </attvalues>
  56: (6)                    </node>
  57: (6)                    <node id="2" label="RTGI">
  58: (8)                      <attvalues>
  59: (10)                       <attvalue for="0" value="http://rtgi.fr"/>
  60: (10)                       <attvalue for="1" value="1"/>
  61: (10)                       <attvalue for="2" value="true"/>
  62: (8)                      </attvalues>
  63: (6)                    </node>
  64: (6)                    <node id="3" label="BarabasiLab">
  65: (8)                      <attvalues>
  66: (10)                       <attvalue for="0" value="http://barabasilab.com"/>
  67: (10)                       <attvalue for="1" value="1"/>
  68: (10)                       <attvalue for="2" value="true"/>
  69: (8)                      </attvalues>
  70: (6)                    </node>
  71: (4)                  </nodes>
  72: (4)                  <edges>
  73: (6)                    <edge id="0" source="0" target="1" label="foo"/>
  74: (6)                    <edge id="1" source="0" target="2"/>
  75: (6)                    <edge id="2" source="1" target="0"/>
  76: (6)                    <edge id="3" source="2" target="1"/>
  77: (6)                    <edge id="4" source="0" target="3"/>
  78: (4)                  </edges>
  79: (2)                </graph>
  80: (0)              </gexf>
  81: (0)              """
  82: (8)                      cls.attribute_graph = nx.DiGraph()
  83: (8)                      cls.attribute_graph.graph["node_default"] = {"frog": True}
  84: (8)                      cls.attribute_graph.add_node(
  85: (12)                         "0", label="Gephi", url="https://gephi.org", indegree=1,
frog=False
  86: (8)                      )
  87: (8)                      cls.attribute_graph.add_node(
  88: (12)                         "1", label="Webatlas", url="http://webatlas.fr", indegree=2,
frog=False
  89: (8)                      )
  90: (8)                      cls.attribute_graph.add_node(
  91: (12)                         "2", label="RTGI", url="http://rtgi.fr", indegree=1, frog=True
  92: (8)                      )
  93: (8)                      cls.attribute_graph.add_node(
  94: (12)                         "3",
  95: (12)                         label="BarabasiLab",
  96: (12)                         url="http://barabasilab.com",
  97: (12)                         indegree=1,
  98: (12)                         frog=True,
  99: (8)                      )
  100: (8)                     cls.attribute_graph.add_edge("0", "1", id="0", label="foo")
  101: (8)                     cls.attribute_graph.add_edge("0", "2", id="1")
  102: (8)                     cls.attribute_graph.add_edge("1", "0", id="2")
  103: (8)                     cls.attribute_graph.add_edge("2", "1", id="3")
  104: (8)                     cls.attribute_graph.add_edge("0", "3", id="4")
  105: (8)                     cls.attribute_fh = io.BytesIO(cls.attribute_data.encode("UTF-8"))
  106: (8)                     cls.simple_undirected_data = """<?xml version="1.0" encoding="UTF-8"?>
```

```
(0)             <gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
(4)                 <graph mode="static" defaultedgetype="undirected">
(8)                     <nodes>
(12)                        <node id="0" label="Hello" />
(12)                        <node id="1" label="Word" />
(8)                     </nodes>
(8)                     <edges>
(12)                        <edge id="0" source="0" target="1" />
(8)                     </edges>
(4)                 </graph>
(0)             </gexf>
(0)             """
(8)                     cls.simple_undirected_graph = nx.Graph()
(8)                     cls.simple_undirected_graph.add_node("0", label="Hello")
(8)                     cls.simple_undirected_graph.add_node("1", label="World")
(8)                     cls.simple_undirected_graph.add_edge("0", "1", id="0")
(8)                     cls.simple_undirected_fh = io.BytesIO(
(12)                        cls.simple_undirected_data.encode("UTF-8")
(8)                     )
(4)                 def test_read_simple_directed_graphml(self):
(8)                     G = self.simple_directed_graph
(8)                     H = nx.read_gexf(self.simple_directed_fh)
(8)                     assert sorted(G.nodes()) == sorted(H.nodes())
(8)                     assert sorted(G.edges()) == sorted(H.edges())
(8)                     assert sorted(G.edges(data=True)) == sorted(H.edges(data=True))
(8)                     self.simple_directed_fh.seek(0)
(4)                 def test_write_read_simple_directed_graphml(self):
(8)                     G = self.simple_directed_graph
(8)                     fh = io.BytesIO()
(8)                     nx.write_gexf(G, fh)
(8)                     fh.seek(0)
(8)                     H = nx.read_gexf(fh)
(8)                     assert sorted(G.nodes()) == sorted(H.nodes())
(8)                     assert sorted(G.edges()) == sorted(H.edges())
(8)                     assert sorted(G.edges(data=True)) == sorted(H.edges(data=True))
(8)                     self.simple_directed_fh.seek(0)
(4)                 def test_read_simple_undirected_graphml(self):
(8)                     G = self.simple_undirected_graph
(8)                     H = nx.read_gexf(self.simple_undirected_fh)
(8)                     assert sorted(G.nodes()) == sorted(H.nodes())
(8)                     assert sorted(sorted(e) for e in G.edges()) == sorted(
(12)                        sorted(e) for e in H.edges()
(8)                     )
(8)                     self.simple_undirected_fh.seek(0)
(4)                 def test_read_attribute_graphml(self):
(8)                     G = self.attribute_graph
(8)                     H = nx.read_gexf(self.attribute_fh)
(8)                     assert sorted(G.nodes(True)) == sorted(H.nodes(data=True))
(8)                     ge = sorted(G.edges(data=True))
(8)                     he = sorted(H.edges(data=True))
(8)                     for a, b in zip(ge, he):
(12)                        assert a == b
(8)                     self.attribute_fh.seek(0)
(4)                 def test_directed_edge_in_undirected(self):
(8)                     s = """<?xml version="1.0" encoding="UTF-8"?>
(0)             <gexf xmlns="http://www.gexf.net/1.2draft" version='1.2'>
(4)                 <graph mode="static" defaultedgetype="undirected" name="">
(8)                     <nodes>
(12)                        <node id="0" label="Hello" />
(12)                        <node id="1" label="Word" />
(8)                     </nodes>
(8)                     <edges>
(12)                        <edge id="0" source="0" target="1" type="directed"/>
(8)                     </edges>
(4)                 </graph>
(0)             </gexf>
(0)             """
(8)                     fh = io.BytesIO(s.encode("UTF-8"))
(8)                     pytest.raises(nx.NetworkXError, nx.read_gexf, fh)
```

```
(4)     def test_undirected_edge_in_directed(self):
(8)         s = """<?xml version="1.0" encoding="UTF-8"?>
(0)     <gexf xmlns="http://www.gexf.net/1.2draft" version='1.2'>
(4)         <graph mode="static" defaultedgetype="directed" name="">
(8)             <nodes>
(12)                <node id="0" label="Hello" />
(12)                <node id="1" label="Word" />
(8)             </nodes>
(8)             <edges>
(12)                <edge id="0" source="0" target="1" type="undirected"/>
(8)             </edges>
(4)         </graph>
(0)     </gexf>
(0)     """
(8)         fh = io.BytesIO(s.encode("UTF-8"))
(8)         pytest.raises(nx.NetworkXError, nx.read_gexf, fh)
(4)     def test_key_raises(self):
(8)         s = """<?xml version="1.0" encoding="UTF-8"?>
(0)     <gexf xmlns="http://www.gexf.net/1.2draft" version='1.2'>
(4)         <graph mode="static" defaultedgetype="directed" name="">
(8)             <nodes>
(12)                <node id="0" label="Hello">
(14)                  <attvalues>
(16)                    <attvalue for='0' value='1'/>
(14)                  </attvalues>
(12)                </node>
(12)                <node id="1" label="Word" />
(8)             </nodes>
(8)             <edges>
(12)                <edge id="0" source="0" target="1" type="undirected"/>
(8)             </edges>
(4)         </graph>
(0)     </gexf>
(0)     """
(8)         fh = io.BytesIO(s.encode("UTF-8"))
(8)         pytest.raises(nx.NetworkXError, nx.read_gexf, fh)
(4)     def test_relabel(self):
(8)         s = """<?xml version="1.0" encoding="UTF-8"?>
(0)     <gexf xmlns="http://www.gexf.net/1.2draft" version='1.2'>
(4)         <graph mode="static" defaultedgetype="directed" name="">
(8)             <nodes>
(12)                <node id="0" label="Hello" />
(12)                <node id="1" label="Word" />
(8)             </nodes>
(8)             <edges>
(12)                <edge id="0" source="0" target="1"/>
(8)             </edges>
(4)         </graph>
(0)     </gexf>
(0)     """
(8)         fh = io.BytesIO(s.encode("UTF-8"))
(8)         G = nx.read_gexf(fh, relabel=True)
(8)         assert sorted(G.nodes()) == ["Hello", "Word"]
(4)     def test_default_attribute(self):
(8)         G = nx.Graph()
(8)         G.add_node(1, label="1", color="green")
(8)         nx.add_path(G, [0, 1, 2, 3])
(8)         G.add_edge(1, 2, foo=3)
(8)         G.graph["node_default"] = {"color": "yellow"}
(8)         G.graph["edge_default"] = {"foo": 7}
(8)         fh = io.BytesIO()
(8)         nx.write_gexf(G, fh)
(8)         fh.seek(0)
(8)         H = nx.read_gexf(fh, node_type=int)
(8)         assert sorted(G.nodes()) == sorted(H.nodes())
(8)         assert sorted(sorted(e) for e in G.edges()) == sorted(
(12)            sorted(e) for e in H.edges()
(8)         )
(8)         del H.graph["mode"]
```

```
(8)                 assert G.graph == H.graph
(4)             def test_serialize_ints_to_strings(self):
(8)                 G = nx.Graph()
(8)                 G.add_node(1, id=7, label=77)
(8)                 fh = io.BytesIO()
(8)                 nx.write_gexf(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert list(H) == [7]
(8)                 assert H.nodes[7]["label"] == "77"
(4)             def test_write_with_node_attributes(self):
(8)                 G = nx.Graph()
(8)                 G.add_edges_from([(0, 1), (1, 2), (2, 3)])
(8)                 for i in range(4):
(12)                    G.nodes[i]["id"] = i
(12)                    G.nodes[i]["label"] = i
(12)                    G.nodes[i]["pid"] = i
(12)                    G.nodes[i]["start"] = i
(12)                    G.nodes[i]["end"] = i + 1
(8)                 expected = f"""<gexf xmlns="http://www.gexf.net/1.2draft" xmlns:xsi\
(0)             ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=\
(0)             "http://www.gexf.net/1.2draft http://www.gexf.net/1.2draft/\
(0)             gexf.xsd" version="1.2">
(2)               <meta lastmodifieddate="{time.strftime('%Y-%m-%d')}">
(4)                 <creator>NetworkX {nx.__version__}</creator>
(2)               </meta>
(2)               <graph defaultedgetype="undirected" mode="dynamic" name=""
timeformat="long">
(4)                 <nodes>
(6)                   <node id="0" label="0" pid="0" start="0" end="1" />
(6)                   <node id="1" label="1" pid="1" start="1" end="2" />
(6)                   <node id="2" label="2" pid="2" start="2" end="3" />
(6)                   <node id="3" label="3" pid="3" start="3" end="4" />
(4)                 </nodes>
(4)                 <edges>
(6)                   <edge source="0" target="1" id="0" />
(6)                   <edge source="1" target="2" id="1" />
(6)                   <edge source="2" target="3" id="2" />
(4)                 </edges>
(2)               </graph>
(0)             </gexf>"""
(8)                 obtained = "\n".join(nx.generate_gexf(G))
(8)                 assert expected == obtained
(4)             def test_edge_id_construct(self):
(8)                 G = nx.Graph()
(8)                 G.add_edges_from([(0, 1, {"id": 0}), (1, 2, {"id": 2}), (2, 3)])
(8)                 expected = f"""<gexf xmlns="http://www.gexf.net/1.2draft" xmlns:xsi\
(0)             ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.\
(0)             gexf.net/1.2draft http://www.gexf.net/1.2draft/gexf.xsd" version="1.2">
(2)               <meta lastmodifieddate="{time.strftime('%Y-%m-%d')}">
(4)                 <creator>NetworkX {nx.__version__}</creator>
(2)               </meta>
(2)               <graph defaultedgetype="undirected" mode="static" name="">
(4)                 <nodes>
(6)                   <node id="0" label="0" />
(6)                   <node id="1" label="1" />
(6)                   <node id="2" label="2" />
(6)                   <node id="3" label="3" />
(4)                 </nodes>
(4)                 <edges>
(6)                   <edge source="0" target="1" id="0" />
(6)                   <edge source="1" target="2" id="2" />
(6)                   <edge source="2" target="3" id="1" />
(4)                 </edges>
(2)               </graph>
(0)             </gexf>"""
(8)                 obtained = "\n".join(nx.generate_gexf(G))
(8)                 assert expected == obtained
(4)             def test_numpy_type(self):
```

```
(8)                 np = pytest.importorskip("numpy")
(8)                 G = nx.path_graph(4)
(8)                 nx.set_node_attributes(G, {n: n for n in np.arange(4)}, "number")
(8)                 G[0][1]["edge-number"] = np.float64(1.1)
(8)                 expected = f"""<gexf xmlns="http://www.gexf.net/1.2draft"\
(1)          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation\
(0)         ="http://www.gexf.net/1.2draft http://www.gexf.net/1.2draft/gexf.xsd"\
(1)          version="1.2">
(2)           <meta lastmodifieddate="{time.strftime('%Y-%m-%d')}">
(4)             <creator>NetworkX {nx.__version__}</creator>
(2)           </meta>
(2)           <graph defaultedgetype="undirected" mode="static" name="">
(4)             <attributes mode="static" class="edge">
(6)               <attribute id="1" title="edge-number" type="float" />
(4)             </attributes>
(4)             <attributes mode="static" class="node">
(6)               <attribute id="0" title="number" type="int" />
(4)             </attributes>
(4)             <nodes>
(6)               <node id="0" label="0">
(8)                 <attvalues>
(10)                 <attvalue for="0" value="0" />
(8)                 </attvalues>
(6)               </node>
(6)               <node id="1" label="1">
(8)                 <attvalues>
(10)                 <attvalue for="0" value="1" />
(8)                 </attvalues>
(6)               </node>
(6)               <node id="2" label="2">
(8)                 <attvalues>
(10)                 <attvalue for="0" value="2" />
(8)                 </attvalues>
(6)               </node>
(6)               <node id="3" label="3">
(8)                 <attvalues>
(10)                 <attvalue for="0" value="3" />
(8)                 </attvalues>
(6)               </node>
(4)             </nodes>
(4)             <edges>
(6)               <edge source="0" target="1" id="0">
(8)                 <attvalues>
(10)                 <attvalue for="1" value="1.1" />
(8)                 </attvalues>
(6)               </edge>
(6)               <edge source="1" target="2" id="1" />
(6)               <edge source="2" target="3" id="2" />
(4)             </edges>
(2)           </graph>
(0)         </gexf>"""
(8)                 obtained = "\n".join(nx.generate_gexf(G))
(8)                 assert expected == obtained
(4)             def test_bool(self):
(8)                 G = nx.Graph()
(8)                 G.add_node(1, testattr=True)
(8)                 fh = io.BytesIO()
(8)                 nx.write_gexf(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert H.nodes[1]["testattr"]
(4)             def test_specials(self):
(8)                 from math import isnan
(8)                 inf, nan = float("inf"), float("nan")
(8)                 G = nx.Graph()
(8)                 G.add_node(1, testattr=inf, strdata="inf", key="a")
(8)                 G.add_node(2, testattr=nan, strdata="nan", key="b")
(8)                 G.add_node(3, testattr=-inf, strdata="-inf", key="c")
(8)                 fh = io.BytesIO()
```

```
(8)                 nx.write_gexf(G, fh)
(8)                 fh.seek(0)
(8)                 filetext = fh.read()
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert b"INF" in filetext
(8)                 assert b"NaN" in filetext
(8)                 assert b"-INF" in filetext
(8)                 assert H.nodes[1]["testattr"] == inf
(8)                 assert isnan(H.nodes[2]["testattr"])
(8)                 assert H.nodes[3]["testattr"] == -inf
(8)                 assert H.nodes[1]["strdata"] == "inf"
(8)                 assert H.nodes[2]["strdata"] == "nan"
(8)                 assert H.nodes[3]["strdata"] == "-inf"
(8)                 assert H.nodes[1]["networkx_key"] == "a"
(8)                 assert H.nodes[2]["networkx_key"] == "b"
(8)                 assert H.nodes[3]["networkx_key"] == "c"
(4)             def test_simple_list(self):
(8)                 G = nx.Graph()
(8)                 list_value = [(1, 2, 3), (9, 1, 2)]
(8)                 G.add_node(1, key=list_value)
(8)                 fh = io.BytesIO()
(8)                 nx.write_gexf(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert H.nodes[1]["networkx_key"] == list_value
(4)             def test_dynamic_mode(self):
(8)                 G = nx.Graph()
(8)                 G.add_node(1, label="1", color="green")
(8)                 G.graph["mode"] = "dynamic"
(8)                 fh = io.BytesIO()
(8)                 nx.write_gexf(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert sorted(G.nodes()) == sorted(H.nodes())
(8)                 assert sorted(sorted(e) for e in G.edges()) == sorted(
(12)                    sorted(e) for e in H.edges()
(8)                 )
(4)             def test_multigraph_with_missing_attributes(self):
(8)                 G = nx.MultiGraph()
(8)                 G.add_node(0, label="1", color="green")
(8)                 G.add_node(1, label="2", color="green")
(8)                 G.add_edge(0, 1, id="0", weight=3, type="undirected", start=0, end=1)
(8)                 G.add_edge(0, 1, id="1", label="foo", start=0, end=1)
(8)                 G.add_edge(0, 1)
(8)                 fh = io.BytesIO()
(8)                 nx.write_gexf(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert sorted(G.nodes()) == sorted(H.nodes())
(8)                 assert sorted(sorted(e) for e in G.edges()) == sorted(
(12)                    sorted(e) for e in H.edges()
(8)                 )
(4)             def test_missing_viz_attributes(self):
(8)                 G = nx.Graph()
(8)                 G.add_node(0, label="1", color="green")
(8)                 G.nodes[0]["viz"] = {"size": 54}
(8)                 G.nodes[0]["viz"]["position"] = {"x": 0, "y": 1, "z": 0}
(8)                 G.nodes[0]["viz"]["color"] = {"r": 0, "g": 0, "b": 256}
(8)                 G.nodes[0]["viz"]["shape"] = "http://random.url"
(8)                 G.nodes[0]["viz"]["thickness"] = 2
(8)                 fh = io.BytesIO()
(8)                 nx.write_gexf(G, fh, version="1.1draft")
(8)                 fh.seek(0)
(8)                 H = nx.read_gexf(fh, node_type=int)
(8)                 assert sorted(G.nodes()) == sorted(H.nodes())
(8)                 assert sorted(sorted(e) for e in G.edges()) == sorted(
(12)                    sorted(e) for e in H.edges()
(8)                 )
```

```
451: (8)                    fh = io.BytesIO()
452: (8)                    nx.write_gexf(G, fh, version="1.2draft")
453: (8)                    fh.seek(0)
454: (8)                    H = nx.read_gexf(fh, node_type=int)
455: (8)                    assert H.nodes[0]["viz"]["color"]["a"] == 1.0
456: (8)                    G = nx.Graph()
457: (8)                    G.add_node(0, label="1", color="green")
458: (8)                    G.nodes[0]["viz"] = {"size": 54}
459: (8)                    G.nodes[0]["viz"]["position"] = {"x": 0, "y": 1, "z": 0}
460: (8)                    G.nodes[0]["viz"]["color"] = {"r": 0, "g": 0, "b": 256, "a": 0.5}
461: (8)                    G.nodes[0]["viz"]["shape"] = "ftp://random.url"
462: (8)                    G.nodes[0]["viz"]["thickness"] = 2
463: (8)                    fh = io.BytesIO()
464: (8)                    nx.write_gexf(G, fh)
465: (8)                    fh.seek(0)
466: (8)                    H = nx.read_gexf(fh, node_type=int)
467: (8)                    assert sorted(G.nodes()) == sorted(H.nodes())
468: (8)                    assert sorted(sorted(e) for e in G.edges()) == sorted(
469: (12)                       sorted(e) for e in H.edges()
470: (8)                    )
471: (4)                def test_slice_and_spell(self):
472: (8)                    G = nx.Graph()
473: (8)                    G.add_node(0, label="1", color="green")
474: (8)                    G.nodes[0]["spells"] = [(1, 2)]
475: (8)                    fh = io.BytesIO()
476: (8)                    nx.write_gexf(G, fh)
477: (8)                    fh.seek(0)
478: (8)                    H = nx.read_gexf(fh, node_type=int)
479: (8)                    assert sorted(G.nodes()) == sorted(H.nodes())
480: (8)                    assert sorted(sorted(e) for e in G.edges()) == sorted(
481: (12)                       sorted(e) for e in H.edges()
482: (8)                    )
483: (8)                    G = nx.Graph()
484: (8)                    G.add_node(0, label="1", color="green")
485: (8)                    G.nodes[0]["slices"] = [(1, 2)]
486: (8)                    fh = io.BytesIO()
487: (8)                    nx.write_gexf(G, fh, version="1.1draft")
488: (8)                    fh.seek(0)
489: (8)                    H = nx.read_gexf(fh, node_type=int)
490: (8)                    assert sorted(G.nodes()) == sorted(H.nodes())
491: (8)                    assert sorted(sorted(e) for e in G.edges()) == sorted(
492: (12)                       sorted(e) for e in H.edges()
493: (8)                    )
494: (4)                def test_add_parent(self):
495: (8)                    G = nx.Graph()
496: (8)                    G.add_node(0, label="1", color="green", parents=[1, 2])
497: (8)                    fh = io.BytesIO()
498: (8)                    nx.write_gexf(G, fh)
499: (8)                    fh.seek(0)
500: (8)                    H = nx.read_gexf(fh, node_type=int)
501: (8)                    assert sorted(G.nodes()) == sorted(H.nodes())
502: (8)                    assert sorted(sorted(e) for e in G.edges()) == sorted(
503: (12)                       sorted(e) for e in H.edges()
504: (8)                    )
```

File 528 - test_gml.py:

```
1: (0)              import codecs
2: (0)              import io
3: (0)              import math
4: (0)              from ast import literal_eval
5: (0)              from contextlib import contextmanager
6: (0)              from textwrap import dedent
7: (0)              import pytest
8: (0)              import networkx as nx
9: (0)              from networkx.readwrite.gml import literal_destringizer, literal_stringizer
10: (0)             class TestGraph:
```

```
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.simple_data = """Creator "me"
(0)             Version "xx"
(0)             graph [
(1)              comment "This is a sample graph"
(1)              directed 1
(1)              IsPlanar 1
(1)              pos  [ x 0 y 1 ]
(1)              node [
(3)                id 1
(3)                label "Node 1"
(3)                pos [ x 1 y 1 ]
(1)              ]
(1)              node [
(4)                 id 2
(4)                 pos [ x 1 y 2 ]
(4)                 label "Node 2"
(4)                 ]
(2)               node [
(4)                 id 3
(4)                 label "Node 3"
(4)                 pos [ x 1 y 3 ]
(2)               ]
(2)               edge [
(4)                 source 1
(4)                 target 2
(4)                 label "Edge from node 1 to node 2"
(4)                 color [line "blue" thickness 3]
(2)               ]
(2)               edge [
(4)                 source 2
(4)                 target 3
(4)                 label "Edge from node 2 to node 3"
(2)               ]
(2)               edge [
(4)                 source 3
(4)                 target 1
(4)                 label "Edge from node 3 to node 1"
(2)               ]
(0)             ]
(0)             """
(4)                 def test_parse_gml_cytoscape_bug(self):
(8)                     cytoscape_example = """
(0)             Creator "Cytoscape"
(0)             Version 1.0
(0)             graph   [
(4)                 node    [
(8)                     root_index  -3
(8)                     id  -3
(8)                     graphics    [
(12)                        x   -96.0
(12)                        y   -67.0
(12)                        w   40.0
(12)                        h   40.0
(12)                        fill    "#ff9999"
(12)                        type    "ellipse"
(12)                        outline "#666666"
(12)                        outline_width   1.5
(8)                     ]
(8)                     label   "node2"
(4)                 ]
(4)                 node    [
(8)                     root_index  -2
(8)                     id  -2
(8)                     graphics    [
(12)                        x   63.0
(12)                        y   37.0
(12)                        w   40.0
```

```
(12)                        h   40.0
(12)                        fill    "#ff9999"
(12)                        type    "ellipse"
(12)                        outline "#666666"
(12)                        outline_width   1.5
(8)                     ]
(8)                     label   "node1"
(4)                 ]
(4)                 node    [
(8)                     root_index  -1
(8)                     id  -1
(8)                     graphics    [
(12)                        x   -31.0
(12)                        y   -17.0
(12)                        w   40.0
(12)                        h   40.0
(12)                        fill    "#ff9999"
(12)                        type    "ellipse"
(12)                        outline "#666666"
(12)                        outline_width   1.5
(8)                     ]
(8)                     label   "node0"
(4)                 ]
(4)                 edge    [
(8)                     root_index  -2
(8)                     target  -2
(8)                     source  -1
(8)                     graphics    [
(12)                        width   1.5
(12)                        fill    "#0000ff"
(12)                        type    "line"
(12)                        Line    [
(12)                        ]
(12)                        source_arrow    0
(12)                        target_arrow    3
(8)                     ]
(8)                     label   "DirectedEdge"
(4)                 ]
(4)                 edge    [
(8)                     root_index  -1
(8)                     target  -1
(8)                     source  -3
(8)                     graphics    [
(12)                        width   1.5
(12)                        fill    "#0000ff"
(12)                        type    "line"
(12)                        Line    [
(12)                        ]
(12)                        source_arrow    0
(12)                        target_arrow    3
(8)                     ]
(8)                     label   "DirectedEdge"
(4)                 ]
(0)             ]
(0)             """
(8)                     nx.parse_gml(cytoscape_example)
(4)                 def test_parse_gml(self):
(8)                     G = nx.parse_gml(self.simple_data, label="label")
(8)                     assert sorted(G.nodes()) == ["Node 1", "Node 2", "Node 3"]
(8)                     assert sorted(G.edges()) == [
(12)                        ("Node 1", "Node 2"),
(12)                        ("Node 2", "Node 3"),
(12)                        ("Node 3", "Node 1"),
(8)                     ]
(8)                     assert sorted(G.edges(data=True)) == [
(12)                        (
(16)                            "Node 1",
(16)                            "Node 2",
(16)                            {
```

```
(20)                                "color": {"line": "blue", "thickness": 3},
(20)                                "label": "Edge from node 1 to node 2",
(16)                            },
(12)                        ),
(12)                        ("Node 2", "Node 3", {"label": "Edge from node 2 to node 3"}),
(12)                        ("Node 3", "Node 1", {"label": "Edge from node 3 to node 1"}),
(8)                     ]
(4)                 def test_read_gml(self, tmp_path):
(8)                     fname = tmp_path / "test.gml"
(8)                     with open(fname, "w") as fh:
(12)                        fh.write(self.simple_data)
(8)                     Gin = nx.read_gml(fname, label="label")
(8)                     G = nx.parse_gml(self.simple_data, label="label")
(8)                     assert sorted(G.nodes(data=True)) == sorted(Gin.nodes(data=True))
(8)                     assert sorted(G.edges(data=True)) == sorted(Gin.edges(data=True))
(4)                 def test_labels_are_strings(self):
(8)                     answer = """graph [
(2)               node [
(4)                 id 0
(4)                 label "1203"
(2)               ]
(0)             ]"""
(8)                     G = nx.Graph()
(8)                     G.add_node(1203)
(8)                     data = "\n".join(nx.generate_gml(G, stringizer=literal_stringizer))
(8)                     assert data == answer
(4)                 def test_relabel_duplicate(self):
(8)                     data = """
(0)             graph
(0)             [
(8)                     label   ""
(8)                     directed        1
(8)                     node
(8)                     [
(16)                            id      0
(16)                            label   "same"
(8)                     ]
(8)                     node
(8)                     [
(16)                            id      1
(16)                            label   "same"
(8)                     ]
(0)             ]
(0)             """
(8)                     fh = io.BytesIO(data.encode("UTF-8"))
(8)                     fh.seek(0)
(8)                     pytest.raises(nx.NetworkXError, nx.read_gml, fh, label="label")
(4)                 @pytest.mark.parametrize("stringizer", (None, literal_stringizer))
(4)                 def test_tuplelabels(self, stringizer):
(8)                     G = nx.Graph()
(8)                     G.add_edge((0, 1), (1, 0))
(8)                     data = "\n".join(nx.generate_gml(G, stringizer=stringizer))
(8)                     answer = """graph [
(2)               node [
(4)                 id 0
(4)                 label "(0,1)"
(2)               ]
(2)               node [
(4)                 id 1
(4)                 label "(1,0)"
(2)               ]
(2)               edge [
(4)                 source 0
(4)                 target 1
(2)               ]
(0)             ]"""
(8)                     assert data == answer
(4)                 def test_quotes(self, tmp_path):
(8)                     G = nx.path_graph(1)
```

```
(8)                 G.name = "path_graph(1)"
(8)                 attr = 'This is "quoted" and this is a copyright: ' + chr(169)
(8)                 G.nodes[0]["demo"] = attr
(8)                 with open(tmp_path / "test.gml", "w+b") as fobj:
(12)                    nx.write_gml(G, fobj)
(12)                    fobj.seek(0)
(12)                    data = fobj.read().strip().decode("ascii")
(8)                 answer = """graph [
(2)               name "path_graph(1)"
(2)               node [
(4)                 id 0
(4)                 label "0"
(4)                 demo "This is &#34;quoted&#34; and this is a copyright: &#169;"
(2)               ]
(0)             ]"""
(8)                 assert data == answer
(4)             def test_unicode_node(self, tmp_path):
(8)                 node = "node" + chr(169)
(8)                 G = nx.Graph()
(8)                 G.add_node(node)
(8)                 with open(tmp_path / "test.gml", "w+b") as fobj:
(12)                    nx.write_gml(G, fobj)
(12)                    fobj.seek(0)
(12)                    data = fobj.read().strip().decode("ascii")
(8)                 answer = """graph [
(2)               node [
(4)                 id 0
(4)                 label "node&#169;"
(2)               ]
(0)             ]"""
(8)                 assert data == answer
(4)             def test_float_label(self, tmp_path):
(8)                 node = 1.0
(8)                 G = nx.Graph()
(8)                 G.add_node(node)
(8)                 with open(tmp_path / "test.gml", "w+b") as fobj:
(12)                    nx.write_gml(G, fobj)
(12)                    fobj.seek(0)
(12)                    data = fobj.read().strip().decode("ascii")
(8)                 answer = """graph [
(2)               node [
(4)                 id 0
(4)                 label "1.0"
(2)               ]
(0)             ]"""
(8)                 assert data == answer
(4)             def test_special_float_label(self, tmp_path):
(8)                 special_floats = [float("nan"), float("+inf"), float("-inf")]
(8)                 try:
(12)                    import numpy as np
(12)                    special_floats += [np.nan, np.inf, np.inf * -1]
(8)                 except ImportError:
(12)                    special_floats += special_floats
(8)                 G = nx.cycle_graph(len(special_floats))
(8)                 attrs = dict(enumerate(special_floats))
(8)                 nx.set_node_attributes(G, attrs, "nodefloat")
(8)                 edges = list(G.edges)
(8)                 attrs = {edges[i]: value for i, value in enumerate(special_floats)}
(8)                 nx.set_edge_attributes(G, attrs, "edgefloat")
(8)                 with open(tmp_path / "test.gml", "w+b") as fobj:
(12)                    nx.write_gml(G, fobj)
(12)                    fobj.seek(0)
(12)                    data = fobj.read().strip().decode("ascii")
(12)                    answer = """graph [
(2)               node [
(4)                 id 0
(4)                 label "0"
(4)                 nodefloat NAN
(2)               ]
```

```
(2)             node [
(4)               id 1
(4)               label "1"
(4)               nodefloat +INF
(2)             ]
(2)             node [
(4)               id 2
(4)               label "2"
(4)               nodefloat -INF
(2)             ]
(2)             node [
(4)               id 3
(4)               label "3"
(4)               nodefloat NAN
(2)             ]
(2)             node [
(4)               id 4
(4)               label "4"
(4)               nodefloat +INF
(2)             ]
(2)             node [
(4)               id 5
(4)               label "5"
(4)               nodefloat -INF
(2)             ]
(2)             edge [
(4)               source 0
(4)               target 1
(4)               edgefloat NAN
(2)             ]
(2)             edge [
(4)               source 0
(4)               target 5
(4)               edgefloat +INF
(2)             ]
(2)             edge [
(4)               source 1
(4)               target 2
(4)               edgefloat -INF
(2)             ]
(2)             edge [
(4)               source 2
(4)               target 3
(4)               edgefloat NAN
(2)             ]
(2)             edge [
(4)               source 3
(4)               target 4
(4)               edgefloat +INF
(2)             ]
(2)             edge [
(4)               source 4
(4)               target 5
(4)               edgefloat -INF
(2)             ]
(0)           ]"""
(12)                      assert data == answer
(12)                      fobj.seek(0)
(12)                      graph = nx.read_gml(fobj)
(12)                      for indx, value in enumerate(special_floats):
(16)                          node_value = graph.nodes[str(indx)]["nodefloat"]
(16)                          if math.isnan(value):
(20)                              assert math.isnan(node_value)
(16)                          else:
(20)                              assert node_value == value
(16)                          edge = edges[indx]
(16)                          string_edge = (str(edge[0]), str(edge[1]))
(16)                          edge_value = graph.edges[string_edge]["edgefloat"]
(16)                          if math.isnan(value):
```

```
(20)                                assert math.isnan(edge_value)
(16)                            else:
(20)                                assert edge_value == value
(4)                 def test_name(self):
(8)                     G = nx.parse_gml('graph [ name "x" node [ id 0 label "x" ] ]')
(8)                     assert "x" == G.graph["name"]
(8)                     G = nx.parse_gml('graph [ node [ id 0 label "x" ] ]')
(8)                     assert "" == G.name
(8)                     assert "name" not in G.graph
(4)                 def test_graph_types(self):
(8)                     for directed in [None, False, True]:
(12)                        for multigraph in [None, False, True]:
(16)                            gml = "graph ["
(16)                            if directed is not None:
(20)                                gml += " directed " + str(int(directed))
(16)                            if multigraph is not None:
(20)                                gml += " multigraph " + str(int(multigraph))
(16)                            gml += ' node [ id 0 label "0" ]'
(16)                            gml += " edge [ source 0 target 0 ]"
(16)                            gml += " ]"
(16)                            G = nx.parse_gml(gml)
(16)                            assert bool(directed) == G.is_directed()
(16)                            assert bool(multigraph) == G.is_multigraph()
(16)                            gml = "graph [\n"
(16)                            if directed is True:
(20)                                gml += "  directed 1\n"
(16)                            if multigraph is True:
(20)                                gml += "  multigraph 1\n"
(16)                            gml += """  node [
(4)                 id 0
(4)                 label "0"
(2)               ]
(2)               edge [
(4)                 source 0
(4)                 target 0
(0)             """
(16)                            if multigraph:
(20)                                gml += "    key 0\n"
(16)                            gml += "  ]\n]"
(16)                            assert gml == "\n".join(nx.generate_gml(G))
(4)                 def test_data_types(self):
(8)                     data = [
(12)                        True,
(12)                        False,
(12)                        10**20,
(12)                        -2e33,
(12)                        "'",
(12)                        '"&&amp;&&#34;"',
(12)                        [{(b"\xfd",): "\x7f", chr(0x4444): (1, 2)}, (2, "3")],
(8)                     ]
(8)                     data.append(chr(0x14444))
(8)                     data.append(literal_eval("{2.3j, 1 - 2.3j, ()}"))
(8)                     G = nx.Graph()
(8)                     G.name = data
(8)                     G.graph["data"] = data
(8)                     G.add_node(0, int=-1, data={"data": data})
(8)                     G.add_edge(0, 0, float=-2.5, data=data)
(8)                     gml = "\n".join(nx.generate_gml(G, stringizer=literal_stringizer))
(8)                     G = nx.parse_gml(gml, destringizer=literal_destringizer)
(8)                     assert data == G.name
(8)                     assert {"name": data, "data": data} == G.graph
(8)                     assert list(G.nodes(data=True)) == [(0, {"int": -1, "data": {"data":
data}})]
(8)                     assert list(G.edges(data=True)) == [(0, 0, {"float": -2.5, "data":
data})]
(8)                     G = nx.Graph()
(8)                     G.graph["data"] = "frozenset([1, 2, 3])"
(8)                     G = nx.parse_gml(nx.generate_gml(G), destringizer=literal_eval)
(8)                     assert G.graph["data"] == "frozenset([1, 2, 3])"
```

```
(4)                 def test_escape_unescape(self):
(8)                     gml = """graph [
(2)               name "&amp;&#34;&#xf;&#x4444;&#1234567890;&#x1234567890abcdef;&unknown;"
(0)             ]"""
(8)                     G = nx.parse_gml(gml)
(8)                     assert (
(12)                        '&"\x0f' + chr(0x4444) +
"&#1234567890;&#x1234567890abcdef;&unknown;"
(12)                        == G.name
(8)                     )
(8)                     gml = "\n".join(nx.generate_gml(G))
(8)                     alnu = "#1234567890;&#38;#x1234567890abcdef"
(8)                     answer = (
(12)                        """graph [
(2)               name "&#38;&#34;&#15;&#17476;&#38;"""
(12)                        + alnu
(12)                        + """;&#38;unknown;"
(0)             ]"""
(8)                     )
(8)                     assert answer == gml
(4)                 def test_exceptions(self, tmp_path):
(8)                     pytest.raises(ValueError, literal_destringizer, "(")
(8)                     pytest.raises(ValueError, literal_destringizer, "frozenset([1, 2,
3])")
(8)                     pytest.raises(ValueError, literal_destringizer, literal_destringizer)
(8)                     pytest.raises(ValueError, literal_stringizer, frozenset([1, 2, 3]))
(8)                     pytest.raises(ValueError, literal_stringizer, literal_stringizer)
(8)                     with open(tmp_path / "test.gml", "w+b") as f:
(12)                        f.write(codecs.BOM_UTF8 + b"graph[]")
(12)                        f.seek(0)
(12)                        pytest.raises(nx.NetworkXError, nx.read_gml, f)
(8)                     def assert_parse_error(gml):
(12)                        pytest.raises(nx.NetworkXError, nx.parse_gml, gml)
(8)                     assert_parse_error(["graph [\n\n", "]"])
(8)                     assert_parse_error("")
(8)                     assert_parse_error('Creator ""')
(8)                     assert_parse_error("0")
(8)                     assert_parse_error("graph ]")
(8)                     assert_parse_error("graph [ 1 ]")
(8)                     assert_parse_error("graph [ 1.E+2 ]")
(8)                     assert_parse_error('graph [ "A" ]')
(8)                     assert_parse_error("graph [ ] graph ]")
(8)                     assert_parse_error("graph [ ] graph [ ]")
(8)                     assert_parse_error("graph [ data [1, 2, 3] ]")
(8)                     assert_parse_error("graph [ node [ ] ]")
(8)                     assert_parse_error("graph [ node [ id 0 ] ]")
(8)                     nx.parse_gml('graph [ node [ id "a" ] ]', label="id")
(8)                     assert_parse_error("graph [ node [ id 0 label 0 ] node [ id 0 label 1
] ]")
(8)                     assert_parse_error("graph [ node [ id 0 label 0 ] node [ id 1 label 0
] ]")
(8)                     assert_parse_error("graph [ node [ id 0 label 0 ] edge [ ] ]")
(8)                     assert_parse_error("graph [ node [ id 0 label 0 ] edge [ source 0 ]
]")
(8)                     nx.parse_gml("graph [edge [ source 0 target 0 ] node [ id 0 label 0 ]
]")
(8)                     assert_parse_error("graph [ node [ id 0 label 0 ] edge [ source 1
target 0 ] ]")
(8)                     assert_parse_error("graph [ node [ id 0 label 0 ] edge [ source 0
target 1 ] ]")
(8)                     assert_parse_error(
(12)                        "graph [ node [ id 0 label 0 ] node [ id 1 label 1 ] "
(12)                        "edge [ source 0 target 1 ] edge [ source 1 target 0 ] ]"
(8)                     )
(8)                     nx.parse_gml(
(12)                        "graph [ node [ id 0 label 0 ] node [ id 1 label 1 ] "
(12)                        "edge [ source 0 target 1 ] edge [ source 1 target 0 ] "
(12)                        "directed 1 ]"
(8)                     )
```

```
(8)                     nx.parse_gml(
(12)                        "graph [ node [ id 0 label 0 ] node [ id 1 label 1 ] "
(12)                        "edge [ source 0 target 1 ] edge [ source 0 target 1 ]"
(12)                        "multigraph 1 ]"
(8)                     )
(8)                     nx.parse_gml(
(12)                        "graph [ node [ id 0 label 0 ] node [ id 1 label 1 ] "
(12)                        "edge [ source 0 target 1 key 0 ] edge [ source 0 target 1 ]"
(12)                        "multigraph 1 ]"
(8)                     )
(8)                     assert_parse_error(
(12)                        "graph [ node [ id 0 label 0 ] node [ id 1 label 1 ] "
(12)                        "edge [ source 0 target 1 key 0 ] edge [ source 0 target 1 key 0
]"
(12)                        "multigraph 1 ]"
(8)                     )
(8)                     nx.parse_gml(
(12)                        "graph [ node [ id 0 label 0 ] node [ id 1 label 1 ] "
(12)                        "edge [ source 0 target 1 key 0 ] edge [ source 1 target 0 key 0
]"
(12)                        "directed 1 multigraph 1 ]"
(8)                     )
(8)                     nx.parse_gml("graph [edge [ source a target a ] node [ id a label b ]
]")
(8)                     nx.parse_gml(
(12)                        "graph [ node [ id n42 label 0 ] node [ id x43 label 1 ]"
(12)                        "edge [ source n42 target x43 key 0 ]"
(12)                        "edge [ source x43 target n42 key 0 ]"
(12)                        "directed 1 multigraph 1 ]"
(8)                     )
(8)                     assert_parse_error(
(12)                        "graph [edge [ source '\u4200' target '\u4200' ] "
(12)                        + "node [ id '\u4200' label b ] ]"
(8)                     )
(8)                     def assert_generate_error(*args, **kwargs):
(12)                        pytest.raises(
(16)                            nx.NetworkXError, lambda: list(nx.generate_gml(*args,
**kwargs))
(12)                        )
(8)                     G = nx.Graph()
(8)                     G.graph[3] = 3
(8)                     assert_generate_error(G)
(8)                     G = nx.Graph()
(8)                     G.graph["3"] = 3
(8)                     assert_generate_error(G)
(8)                     G = nx.Graph()
(8)                     G.graph["data"] = frozenset([1, 2, 3])
(8)                     assert_generate_error(G, stringizer=literal_stringizer)
(4)                 def test_label_kwarg(self):
(8)                     G = nx.parse_gml(self.simple_data, label="id")
(8)                     assert sorted(G.nodes) == [1, 2, 3]
(8)                     labels = [G.nodes[n]["label"] for n in sorted(G.nodes)]
(8)                     assert labels == ["Node 1", "Node 2", "Node 3"]
(8)                     G = nx.parse_gml(self.simple_data, label=None)
(8)                     assert sorted(G.nodes) == [1, 2, 3]
(8)                     labels = [G.nodes[n]["label"] for n in sorted(G.nodes)]
(8)                     assert labels == ["Node 1", "Node 2", "Node 3"]
(4)                 def test_outofrange_integers(self, tmp_path):
(8)                     G = nx.Graph()
(8)                     numbers = {
(12)                        "toosmall": (-(2**31)) - 1,
(12)                        "small": -(2**31),
(12)                        "med1": -4,
(12)                        "med2": 0,
(12)                        "med3": 17,
(12)                        "big": (2**31) - 1,
(12)                        "toobig": 2**31,
(8)                     }
(8)                     G.add_node("Node", **numbers)
```

```
549: (8)                    fname = tmp_path / "test.gml"
550: (8)                    nx.write_gml(G, fname)
551: (8)                    G2 = nx.read_gml(fname)
552: (8)                    for attr, value in G2.nodes["Node"].items():
553: (12)                       if attr == "toosmall" or attr == "toobig":
554: (16)                           assert type(value) == str
555: (12)                       else:
556: (16)                           assert type(value) == int
557: (4)                def test_multiline(self):
558: (8)                    multiline_example = """
559: (0)            graph
560: (0)            [
561: (4)                node
562: (4)                [
563: (8)                    id 0
564: (8)                    label "multiline node"
565: (8)                    label2 "multiline1
566: (4)                multiline2
567: (4)                multiline3"
568: (8)                    alt_name "id 0"
569: (4)                ]
570: (0)            ]
571: (0)            """
572: (8)                    G = nx.parse_gml(multiline_example)
573: (8)                    assert G.nodes["multiline node"] == {
574: (12)                       "label2": "multiline1 multiline2 multiline3",
575: (12)                       "alt_name": "id 0",
576: (8)                    }
577: (0)            @contextmanager
578: (0)            def byte_file():
579: (4)                _file_handle = io.BytesIO()
580: (4)                yield _file_handle
581: (4)                _file_handle.seek(0)
582: (0)            class TestPropertyLists:
583: (4)                def test_writing_graph_with_multi_element_property_list(self):
584: (8)                    g = nx.Graph()
585: (8)                    g.add_node("n1", properties=["element", 0, 1, 2.5, True, False])
586: (8)                    with byte_file() as f:
587: (12)                       nx.write_gml(g, f)
588: (8)                    result = f.read().decode()
589: (8)                    assert result == dedent(
590: (12)                       """\
591: (12)                       graph [
592: (14)                         node [
593: (16)                           id 0
594: (16)                           label "n1"
595: (16)                           properties "element"
596: (16)                           properties 0
597: (16)                           properties 1
598: (16)                           properties 2.5
599: (16)                           properties 1
600: (16)                           properties 0
601: (14)                         ]
602: (12)                       ]
603: (8)                    """
604: (8)                    )
605: (4)                def test_writing_graph_with_one_element_property_list(self):
606: (8)                    g = nx.Graph()
607: (8)                    g.add_node("n1", properties=["element"])
608: (8)                    with byte_file() as f:
609: (12)                       nx.write_gml(g, f)
610: (8)                    result = f.read().decode()
611: (8)                    assert result == dedent(
612: (12)                       """\
613: (12)                       graph [
614: (14)                         node [
615: (16)                           id 0
616: (16)                           label "n1"
617: (16)                           properties "_networkx_list_start"
```

```
618: (16)                            properties "element"
619: (14)                          ]
620: (12)                        ]
621: (8)                    """
622: (8)                    )
623: (4)                def test_reading_graph_with_list_property(self):
624: (8)                    with byte_file() as f:
625: (12)                       f.write(
626: (16)                           dedent(
627: (20)                               """
628: (14)                          graph [
629: (16)                            node [
630: (18)                              id 0
631: (18)                              label "n1"
632: (18)                              properties "element"
633: (18)                              properties 0
634: (18)                              properties 1
635: (18)                              properties 2.5
636: (16)                            ]
637: (14)                          ]
638: (12)                       """
639: (16)                           ).encode("ascii")
640: (12)                       )
641: (12)                       f.seek(0)
642: (12)                       graph = nx.read_gml(f)
643: (8)                    assert graph.nodes(data=True)["n1"] == {"properties": ["element", 0,
1, 2.5]}
644: (4)                def test_reading_graph_with_single_element_list_property(self):
645: (8)                    with byte_file() as f:
646: (12)                       f.write(
647: (16)                           dedent(
648: (20)                               """
649: (14)                          graph [
650: (16)                            node [
651: (18)                              id 0
652: (18)                              label "n1"
653: (18)                              properties "_networkx_list_start"
654: (18)                              properties "element"
655: (16)                            ]
656: (14)                          ]
657: (12)                       """
658: (16)                           ).encode("ascii")
659: (12)                       )
660: (12)                       f.seek(0)
661: (12)                       graph = nx.read_gml(f)
662: (8)                    assert graph.nodes(data=True)["n1"] == {"properties": ["element"]}
663: (0)            @pytest.mark.parametrize("coll", ([], ()))
664: (0)            def test_stringize_empty_list_tuple(coll):
665: (4)                G = nx.path_graph(2)
666: (4)                G.nodes[0]["test"] = coll  # test serializing an empty collection
667: (4)                f = io.BytesIO()
668: (4)                nx.write_gml(G, f)  # Smoke test - should not raise
669: (4)                f.seek(0)
670: (4)                H = nx.read_gml(f)
671: (4)                assert H.nodes["0"]["test"] == coll  # Check empty list round-trips
properly
672: (4)                H = nx.relabel_nodes(H, {"0": 0, "1": 1})
673: (4)                assert nx.utils.graphs_equal(G, H)
674: (4)                f.seek(0)
675: (4)                H = nx.read_gml(f, destringizer=int)
676: (4)                assert nx.utils.graphs_equal(G, H)

----------------------------------------

File 529 - test_graph6.py:

1: (0)              from io import BytesIO
2: (0)              import pytest
3: (0)              import networkx as nx
```

```
4: (0)              import networkx.readwrite.graph6 as g6
5: (0)              from networkx.utils import edges_equal, nodes_equal
6: (0)              class TestGraph6Utils:
7: (4)                  def test_n_data_n_conversion(self):
8: (8)                      for i in [0, 1, 42, 62, 63, 64, 258047, 258048, 7744773, 68719476735]:
9: (12)                         assert g6.data_to_n(g6.n_to_data(i))[0] == i
10: (12)                        assert g6.data_to_n(g6.n_to_data(i))[1] == []
11: (12)                        assert g6.data_to_n(g6.n_to_data(i) + [42, 43])[1] == [42, 43]
12: (0)             class TestFromGraph6Bytes:
13: (4)                 def test_from_graph6_bytes(self):
14: (8)                     data = b"DF{"
15: (8)                     G = nx.from_graph6_bytes(data)
16: (8)                     assert nodes_equal(G.nodes(), [0, 1, 2, 3, 4])
17: (8)                     assert edges_equal(
18: (12)                        G.edges(), [(0, 3), (0, 4), (1, 3), (1, 4), (2, 3), (2, 4), (3,
4)]
19: (8)                     )
20: (4)                 def test_read_equals_from_bytes(self):
21: (8)                     data = b"DF{"
22: (8)                     G = nx.from_graph6_bytes(data)
23: (8)                     fh = BytesIO(data)
24: (8)                     Gin = nx.read_graph6(fh)
25: (8)                     assert nodes_equal(G.nodes(), Gin.nodes())
26: (8)                     assert edges_equal(G.edges(), Gin.edges())
27: (0)             class TestReadGraph6:
28: (4)                 def test_read_many_graph6(self):
29: (8)                     """Test for reading many graphs from a file into a list."""
30: (8)                     data = b"DF{\nD`{\nDqK\nD~{\n"
31: (8)                     fh = BytesIO(data)
32: (8)                     glist = nx.read_graph6(fh)
33: (8)                     assert len(glist) == 4
34: (8)                     for G in glist:
35: (12)                        assert sorted(G) == list(range(5))
36: (0)             class TestWriteGraph6:
37: (4)                 """Unit tests for writing a graph to a file in graph6 format."""
38: (4)                 def test_null_graph(self):
39: (8)                     result = BytesIO()
40: (8)                     nx.write_graph6(nx.null_graph(), result)
41: (8)                     assert result.getvalue() == b">>graph6<<?\n"
42: (4)                 def test_trivial_graph(self):
43: (8)                     result = BytesIO()
44: (8)                     nx.write_graph6(nx.trivial_graph(), result)
45: (8)                     assert result.getvalue() == b">>graph6<<@\n"
46: (4)                 def test_complete_graph(self):
47: (8)                     result = BytesIO()
48: (8)                     nx.write_graph6(nx.complete_graph(4), result)
49: (8)                     assert result.getvalue() == b">>graph6<<C~\n"
50: (4)                 def test_large_complete_graph(self):
51: (8)                     result = BytesIO()
52: (8)                     nx.write_graph6(nx.complete_graph(67), result, header=False)
53: (8)                     assert result.getvalue() == b"~?@B" + b"~" * 368 + b"w\n"
54: (4)                 def test_no_header(self):
55: (8)                     result = BytesIO()
56: (8)                     nx.write_graph6(nx.complete_graph(4), result, header=False)
57: (8)                     assert result.getvalue() == b"C~\n"
58: (4)                 def test_complete_bipartite_graph(self):
59: (8)                     result = BytesIO()
60: (8)                     G = nx.complete_bipartite_graph(6, 9)
61: (8)                     nx.write_graph6(G, result, header=False)
62: (8)                     assert result.getvalue() == b"N??F~z{~Fw^_~?~?^_?\n"
63: (4)                 @pytest.mark.parametrize("G", (nx.MultiGraph(), nx.DiGraph()))
64: (4)                 def test_no_directed_or_multi_graphs(self, G):
65: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
66: (12)                        nx.write_graph6(G, BytesIO())
67: (4)                 def test_length(self):
68: (8)                     for i in list(range(13)) + [31, 47, 62, 63, 64, 72]:
69: (12)                        g = nx.random_graphs.gnm_random_graph(i, i * i // 4, seed=i)
70: (12)                        gstr = BytesIO()
71: (12)                        nx.write_graph6(g, gstr, header=False)
```

```
 72: (12)                        gstr = gstr.getvalue().rstrip()
 73: (12)                        assert len(gstr) == ((i - 1) * i // 2 + 5) // 6 + (1 if i < 63
else 4)
 74: (4)                 def test_roundtrip(self):
 75: (8)                     for i in list(range(13)) + [31, 47, 62, 63, 64, 72]:
 76: (12)                        G = nx.random_graphs.gnm_random_graph(i, i * i // 4, seed=i)
 77: (12)                        f = BytesIO()
 78: (12)                        nx.write_graph6(G, f)
 79: (12)                        f.seek(0)
 80: (12)                        H = nx.read_graph6(f)
 81: (12)                        assert nodes_equal(G.nodes(), H.nodes())
 82: (12)                        assert edges_equal(G.edges(), H.edges())
 83: (4)                 def test_write_path(self, tmp_path):
 84: (8)                     with open(tmp_path / "test.g6", "w+b") as f:
 85: (12)                        g6.write_graph6_file(nx.null_graph(), f)
 86: (12)                        f.seek(0)
 87: (12)                        assert f.read() == b">>graph6<<?\n"
 88: (4)                 @pytest.mark.parametrize("edge", ((0, 1), (1, 2), (1, 42)))
 89: (4)                 def test_relabeling(self, edge):
 90: (8)                     G = nx.Graph([edge])
 91: (8)                     f = BytesIO()
 92: (8)                     nx.write_graph6(G, f)
 93: (8)                     f.seek(0)
 94: (8)                     assert f.read() == b">>graph6<<A_\n"
 95: (0)             class TestToGraph6Bytes:
 96: (4)                 def test_null_graph(self):
 97: (8)                     G = nx.null_graph()
 98: (8)                     assert g6.to_graph6_bytes(G) == b">>graph6<<?\n"
 99: (4)                 def test_trivial_graph(self):
100: (8)                     G = nx.trivial_graph()
101: (8)                     assert g6.to_graph6_bytes(G) == b">>graph6<<@\n"
102: (4)                 def test_complete_graph(self):
103: (8)                     assert g6.to_graph6_bytes(nx.complete_graph(4)) == b">>graph6<<C~\n"
104: (4)                 def test_large_complete_graph(self):
105: (8)                     G = nx.complete_graph(67)
106: (8)                     assert g6.to_graph6_bytes(G, header=False) == b"~?@B" + b"~" * 368 +
b"w\n"
107: (4)                 def test_no_header(self):
108: (8)                     G = nx.complete_graph(4)
109: (8)                     assert g6.to_graph6_bytes(G, header=False) == b"C~\n"
110: (4)                 def test_complete_bipartite_graph(self):
111: (8)                     G = nx.complete_bipartite_graph(6, 9)
112: (8)                     assert g6.to_graph6_bytes(G, header=False) == b"N??F~z{~Fw^_~?~?^_?\n"
113: (4)                 @pytest.mark.parametrize("G", (nx.MultiGraph(), nx.DiGraph()))
114: (4)                 def test_no_directed_or_multi_graphs(self, G):
115: (8)                     with pytest.raises(nx.NetworkXNotImplemented):
116: (12)                        g6.to_graph6_bytes(G)
117: (4)                 def test_length(self):
118: (8)                     for i in list(range(13)) + [31, 47, 62, 63, 64, 72]:
119: (12)                        G = nx.random_graphs.gnm_random_graph(i, i * i // 4, seed=i)
120: (12)                        gstr = g6.to_graph6_bytes(G, header=False).rstrip()
121: (12)                        assert len(gstr) == ((i - 1) * i // 2 + 5) // 6 + (1 if i < 63
else 4)
122: (4)                 def test_roundtrip(self):
123: (8)                     for i in list(range(13)) + [31, 47, 62, 63, 64, 72]:
124: (12)                        G = nx.random_graphs.gnm_random_graph(i, i * i // 4, seed=i)
125: (12)                        data = g6.to_graph6_bytes(G)
126: (12)                        H = nx.from_graph6_bytes(data.rstrip())
127: (12)                        assert nodes_equal(G.nodes(), H.nodes())
128: (12)                        assert edges_equal(G.edges(), H.edges())
129: (4)                 @pytest.mark.parametrize("edge", ((0, 1), (1, 2), (1, 42)))
130: (4)                 def test_relabeling(self, edge):
131: (8)                     G = nx.Graph([edge])
132: (8)                     assert g6.to_graph6_bytes(G) == b">>graph6<<A_\n"

----------------------------------------

File 530 - test_graphml.py:
```

```
(0)             import io
(0)             import pytest
(0)             import networkx as nx
(0)             from networkx.readwrite.graphml import GraphMLWriter
(0)             from networkx.utils import edges_equal, nodes_equal
(0)             class BaseGraphML:
(4)                 @classmethod
(4)                 def setup_class(cls):
(8)                     cls.simple_directed_data = """<?xml version="1.0" encoding="UTF-8"?>
(0)             <!-- This file was written by the JAVA GraphML Library.-->
(0)             <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                      xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                      http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)               <graph id="G" edgedefault="directed">
(4)                 <node id="n0"/>
(4)                 <node id="n1"/>
(4)                 <node id="n2"/>
(4)                 <node id="n3"/>
(4)                 <node id="n4"/>
(4)                 <node id="n5"/>
(4)                 <node id="n6"/>
(4)                 <node id="n7"/>
(4)                 <node id="n8"/>
(4)                 <node id="n9"/>
(4)                 <node id="n10"/>
(4)                 <edge id="foo" source="n0" target="n2"/>
(4)                 <edge source="n1" target="n2"/>
(4)                 <edge source="n2" target="n3"/>
(4)                 <edge source="n3" target="n5"/>
(4)                 <edge source="n3" target="n4"/>
(4)                 <edge source="n4" target="n6"/>
(4)                 <edge source="n6" target="n5"/>
(4)                 <edge source="n5" target="n7"/>
(4)                 <edge source="n6" target="n8"/>
(4)                 <edge source="n8" target="n7"/>
(4)                 <edge source="n8" target="n9"/>
(2)               </graph>
(0)             </graphml>"""
(8)                     cls.simple_directed_graph = nx.DiGraph()
(8)                     cls.simple_directed_graph.add_node("n10")
(8)                     cls.simple_directed_graph.add_edge("n0", "n2", id="foo")
(8)                     cls.simple_directed_graph.add_edge("n0", "n2")
(8)                     cls.simple_directed_graph.add_edges_from(
(12)                        [
(16)                            ("n1", "n2"),
(16)                            ("n2", "n3"),
(16)                            ("n3", "n5"),
(16)                            ("n3", "n4"),
(16)                            ("n4", "n6"),
(16)                            ("n6", "n5"),
(16)                            ("n5", "n7"),
(16)                            ("n6", "n8"),
(16)                            ("n8", "n7"),
(16)                            ("n8", "n9"),
(12)                        ]
(8)                     )
(8)                     cls.simple_directed_fh =
io.BytesIO(cls.simple_directed_data.encode("UTF-8"))
(8)                     cls.attribute_data = """<?xml version="1.0" encoding="UTF-8"?>
(0)             <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(6)                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(6)                   xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(8)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)               <key id="d0" for="node" attr.name="color" attr.type="string">
(4)                 <default>yellow</default>
(2)               </key>
(2)               <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
(2)               <graph id="G" edgedefault="directed">
```

```
 69: (4)                  <node id="n0">
 70: (6)                    <data key="d0">green</data>
 71: (4)                  </node>
 72: (4)                  <node id="n1"/>
 73: (4)                  <node id="n2">
 74: (6)                    <data key="d0">blue</data>
 75: (4)                  </node>
 76: (4)                  <node id="n3">
 77: (6)                    <data key="d0">red</data>
 78: (4)                  </node>
 79: (4)                  <node id="n4"/>
 80: (4)                  <node id="n5">
 81: (6)                    <data key="d0">turquoise</data>
 82: (4)                  </node>
 83: (4)                  <edge id="e0" source="n0" target="n2">
 84: (6)                    <data key="d1">1.0</data>
 85: (4)                  </edge>
 86: (4)                  <edge id="e1" source="n0" target="n1">
 87: (6)                    <data key="d1">1.0</data>
 88: (4)                  </edge>
 89: (4)                  <edge id="e2" source="n1" target="n3">
 90: (6)                    <data key="d1">2.0</data>
 91: (4)                  </edge>
 92: (4)                  <edge id="e3" source="n3" target="n2"/>
 93: (4)                  <edge id="e4" source="n2" target="n4"/>
 94: (4)                  <edge id="e5" source="n3" target="n5"/>
 95: (4)                  <edge id="e6" source="n5" target="n4">
 96: (6)                    <data key="d1">1.1</data>
 97: (4)                  </edge>
 98: (2)                </graph>
 99: (0)              </graphml>
 100: (0)             """
 101: (8)                     cls.attribute_graph = nx.DiGraph(id="G")
 102: (8)                     cls.attribute_graph.graph["node_default"] = {"color": "yellow"}
 103: (8)                     cls.attribute_graph.add_node("n0", color="green")
 104: (8)                     cls.attribute_graph.add_node("n2", color="blue")
 105: (8)                     cls.attribute_graph.add_node("n3", color="red")
 106: (8)                     cls.attribute_graph.add_node("n4")
 107: (8)                     cls.attribute_graph.add_node("n5", color="turquoise")
 108: (8)                     cls.attribute_graph.add_edge("n0", "n2", id="e0", weight=1.0)
 109: (8)                     cls.attribute_graph.add_edge("n0", "n1", id="e1", weight=1.0)
 110: (8)                     cls.attribute_graph.add_edge("n1", "n3", id="e2", weight=2.0)
 111: (8)                     cls.attribute_graph.add_edge("n3", "n2", id="e3")
 112: (8)                     cls.attribute_graph.add_edge("n2", "n4", id="e4")
 113: (8)                     cls.attribute_graph.add_edge("n3", "n5", id="e5")
 114: (8)                     cls.attribute_graph.add_edge("n5", "n4", id="e6", weight=1.1)
 115: (8)                     cls.attribute_fh = io.BytesIO(cls.attribute_data.encode("UTF-8"))
 116: (8)                     cls.node_attribute_default_data = """<?xml version="1.0"
encoding="UTF-8"?>
 117: (8)                     <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 118: (14)                          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 119: (14)                          xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
 120: (16)                            http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
 121: (10)                       <key id="d0" for="node" attr.name="boolean_attribute"
attr.type="boolean"><default>false</default></key>
 122: (10)                       <key id="d1" for="node" attr.name="int_attribute" attr.type="int">
<default>0</default></key>
 123: (10)                       <key id="d2" for="node" attr.name="long_attribute" attr.type="long">
<default>0</default></key>
 124: (10)                       <key id="d3" for="node" attr.name="float_attribute"
attr.type="float"><default>0.0</default></key>
 125: (10)                       <key id="d4" for="node" attr.name="double_attribute"
attr.type="double"><default>0.0</default></key>
 126: (10)                       <key id="d5" for="node" attr.name="string_attribute"
attr.type="string"><default>Foo</default></key>
 127: (10)                       <graph id="G" edgedefault="directed">
 128: (12)                         <node id="n0"/>
 129: (12)                         <node id="n1"/>
 130: (12)                         <edge id="e0" source="n0" target="n1"/>
```

```
131: (10)                    </graph>
132: (8)                   </graphml>
133: (8)                   """
134: (8)                   cls.node_attribute_default_graph = nx.DiGraph(id="G")
135: (8)                   cls.node_attribute_default_graph.graph["node_default"] = {
136: (12)                      "boolean_attribute": False,
137: (12)                      "int_attribute": 0,
138: (12)                      "long_attribute": 0,
139: (12)                      "float_attribute": 0.0,
140: (12)                      "double_attribute": 0.0,
141: (12)                      "string_attribute": "Foo",
142: (8)                   }
143: (8)                   cls.node_attribute_default_graph.add_node("n0")
144: (8)                   cls.node_attribute_default_graph.add_node("n1")
145: (8)                   cls.node_attribute_default_graph.add_edge("n0", "n1", id="e0")
146: (8)                   cls.node_attribute_default_fh = io.BytesIO(
147: (12)                      cls.node_attribute_default_data.encode("UTF-8")
148: (8)                   )
149: (8)                   cls.attribute_named_key_ids_data = """<?xml version='1.0'
encoding='utf-8'?>
150: (0)           <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
151: (9)                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
152: (9)                    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
153: (9)                    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
154: (2)             <key id="edge_prop" for="edge" attr.name="edge_prop" attr.type="string"/>
155: (2)             <key id="prop2" for="node" attr.name="prop2" attr.type="string"/>
156: (2)             <key id="prop1" for="node" attr.name="prop1" attr.type="string"/>
157: (2)             <graph edgedefault="directed">
158: (4)               <node id="0">
159: (6)                 <data key="prop1">val1</data>
160: (6)                 <data key="prop2">val2</data>
161: (4)               </node>
162: (4)               <node id="1">
163: (6)                 <data key="prop1">val_one</data>
164: (6)                 <data key="prop2">val2</data>
165: (4)               </node>
166: (4)               <edge source="0" target="1">
167: (6)                 <data key="edge_prop">edge_value</data>
168: (4)               </edge>
169: (2)             </graph>
170: (0)           </graphml>
171: (0)           """
172: (8)                   cls.attribute_named_key_ids_graph = nx.DiGraph()
173: (8)                   cls.attribute_named_key_ids_graph.add_node("0", prop1="val1",
prop2="val2")
174: (8)                   cls.attribute_named_key_ids_graph.add_node("1", prop1="val_one",
prop2="val2")
175: (8)                   cls.attribute_named_key_ids_graph.add_edge("0", "1",
edge_prop="edge_value")
176: (8)                   fh = io.BytesIO(cls.attribute_named_key_ids_data.encode("UTF-8"))
177: (8)                   cls.attribute_named_key_ids_fh = fh
178: (8)                   cls.attribute_numeric_type_data = """<?xml version='1.0'
encoding='utf-8'?>
179: (0)           <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
180: (9)                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
181: (9)                    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
182: (9)                    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
183: (2)             <key attr.name="weight" attr.type="double" for="node" id="d1" />
184: (2)             <key attr.name="weight" attr.type="double" for="edge" id="d0" />
185: (2)             <graph edgedefault="directed">
186: (4)               <node id="n0">
187: (6)                 <data key="d1">1</data>
188: (4)               </node>
189: (4)               <node id="n1">
190: (6)                 <data key="d1">2.0</data>
191: (4)               </node>
192: (4)               <edge source="n0" target="n1">
193: (6)                 <data key="d0">1</data>
194: (4)               </edge>
```

```
195: (4)                <edge source="n1" target="n0">
196: (6)                  <data key="d0">k</data>
197: (4)                </edge>
198: (4)                <edge source="n1" target="n1">
199: (6)                  <data key="d0">1.0</data>
200: (4)                </edge>
201: (2)              </graph>
202: (0)            </graphml>
203: (0)            """
204: (8)                    cls.attribute_numeric_type_graph = nx.DiGraph()
205: (8)                    cls.attribute_numeric_type_graph.add_node("n0", weight=1)
206: (8)                    cls.attribute_numeric_type_graph.add_node("n1", weight=2.0)
207: (8)                    cls.attribute_numeric_type_graph.add_edge("n0", "n1", weight=1)
208: (8)                    cls.attribute_numeric_type_graph.add_edge("n1", "n1", weight=1.0)
209: (8)                    fh = io.BytesIO(cls.attribute_numeric_type_data.encode("UTF-8"))
210: (8)                    cls.attribute_numeric_type_fh = fh
211: (8)                    cls.simple_undirected_data = """<?xml version="1.0" encoding="UTF-8"?>
212: (0)            <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
213: (9)                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
214: (9)                     xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
215: (9)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
216: (2)              <graph id="G">
217: (4)                <node id="n0"/>
218: (4)                <node id="n1"/>
219: (4)                <node id="n2"/>
220: (4)                <node id="n10"/>
221: (4)                <edge id="foo" source="n0" target="n2"/>
222: (4)                <edge source="n1" target="n2"/>
223: (4)                <edge source="n2" target="n3"/>
224: (2)              </graph>
225: (0)            </graphml>"""
226: (8)                    cls.simple_undirected_graph = nx.Graph()
227: (8)                    cls.simple_undirected_graph.add_node("n10")
228: (8)                    cls.simple_undirected_graph.add_edge("n0", "n2", id="foo")
229: (8)                    cls.simple_undirected_graph.add_edges_from([("n1", "n2"), ("n2",
"n3")])
230: (8)                    fh = io.BytesIO(cls.simple_undirected_data.encode("UTF-8"))
231: (8)                    cls.simple_undirected_fh = fh
232: (8)                    cls.undirected_multigraph_data = """<?xml version="1.0" encoding="UTF-
8"?>
233: (0)            <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
234: (9)                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
235: (9)                     xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
236: (9)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
237: (2)              <graph id="G">
238: (4)                <node id="n0"/>
239: (4)                <node id="n1"/>
240: (4)                <node id="n2"/>
241: (4)                <node id="n10"/>
242: (4)                <edge id="e0" source="n0" target="n2"/>
243: (4)                <edge id="e1" source="n1" target="n2"/>
244: (4)                <edge id="e2" source="n2" target="n1"/>
245: (2)              </graph>
246: (0)            </graphml>"""
247: (8)                    cls.undirected_multigraph = nx.MultiGraph()
248: (8)                    cls.undirected_multigraph.add_node("n10")
249: (8)                    cls.undirected_multigraph.add_edge("n0", "n2", id="e0")
250: (8)                    cls.undirected_multigraph.add_edge("n1", "n2", id="e1")
251: (8)                    cls.undirected_multigraph.add_edge("n2", "n1", id="e2")
252: (8)                    fh = io.BytesIO(cls.undirected_multigraph_data.encode("UTF-8"))
253: (8)                    cls.undirected_multigraph_fh = fh
254: (8)                    cls.undirected_multigraph_no_multiedge_data = """<?xml version="1.0"
encoding="UTF-8"?>
255: (0)            <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
256: (9)                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
257: (9)                     xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
258: (9)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
259: (2)              <graph id="G">
260: (4)                <node id="n0"/>
```

```
(4)                <node id="n1"/>
(4)                <node id="n2"/>
(4)                <node id="n10"/>
(4)                <edge id="e0" source="n0" target="n2"/>
(4)                <edge id="e1" source="n1" target="n2"/>
(4)                <edge id="e2" source="n2" target="n3"/>
(2)              </graph>
(0)            </graphml>"""
(8)                    cls.undirected_multigraph_no_multiedge = nx.MultiGraph()
(8)                    cls.undirected_multigraph_no_multiedge.add_node("n10")
(8)                    cls.undirected_multigraph_no_multiedge.add_edge("n0", "n2", id="e0")
(8)                    cls.undirected_multigraph_no_multiedge.add_edge("n1", "n2", id="e1")
(8)                    cls.undirected_multigraph_no_multiedge.add_edge("n2", "n3", id="e2")
(8)                    fh =
io.BytesIO(cls.undirected_multigraph_no_multiedge_data.encode("UTF-8"))
(8)                    cls.undirected_multigraph_no_multiedge_fh = fh
(8)                    cls.multigraph_only_ids_for_multiedges_data = """<?xml version="1.0"
encoding="UTF-8"?>
(0)            <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                     xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)              <graph id="G">
(4)                <node id="n0"/>
(4)                <node id="n1"/>
(4)                <node id="n2"/>
(4)                <node id="n10"/>
(4)                <edge source="n0" target="n2"/>
(4)                <edge id="e1" source="n1" target="n2"/>
(4)                <edge id="e2" source="n2" target="n1"/>
(2)              </graph>
(0)            </graphml>"""
(8)                    cls.multigraph_only_ids_for_multiedges = nx.MultiGraph()
(8)                    cls.multigraph_only_ids_for_multiedges.add_node("n10")
(8)                    cls.multigraph_only_ids_for_multiedges.add_edge("n0", "n2")
(8)                    cls.multigraph_only_ids_for_multiedges.add_edge("n1", "n2", id="e1")
(8)                    cls.multigraph_only_ids_for_multiedges.add_edge("n2", "n1", id="e2")
(8)                    fh =
io.BytesIO(cls.multigraph_only_ids_for_multiedges_data.encode("UTF-8"))
(8)                    cls.multigraph_only_ids_for_multiedges_fh = fh
(0)            class TestReadGraphML(BaseGraphML):
(4)                def test_read_simple_directed_graphml(self):
(8)                    G = self.simple_directed_graph
(8)                    H = nx.read_graphml(self.simple_directed_fh)
(8)                    assert sorted(G.nodes()) == sorted(H.nodes())
(8)                    assert sorted(G.edges()) == sorted(H.edges())
(8)                    assert sorted(G.edges(data=True)) == sorted(H.edges(data=True))
(8)                    self.simple_directed_fh.seek(0)
(8)                    PG = nx.parse_graphml(self.simple_directed_data)
(8)                    assert sorted(G.nodes()) == sorted(PG.nodes())
(8)                    assert sorted(G.edges()) == sorted(PG.edges())
(8)                    assert sorted(G.edges(data=True)) == sorted(PG.edges(data=True))
(4)                def test_read_simple_undirected_graphml(self):
(8)                    G = self.simple_undirected_graph
(8)                    H = nx.read_graphml(self.simple_undirected_fh)
(8)                    assert nodes_equal(G.nodes(), H.nodes())
(8)                    assert edges_equal(G.edges(), H.edges())
(8)                    self.simple_undirected_fh.seek(0)
(8)                    PG = nx.parse_graphml(self.simple_undirected_data)
(8)                    assert nodes_equal(G.nodes(), PG.nodes())
(8)                    assert edges_equal(G.edges(), PG.edges())
(4)                def test_read_undirected_multigraph_graphml(self):
(8)                    G = self.undirected_multigraph
(8)                    H = nx.read_graphml(self.undirected_multigraph_fh)
(8)                    assert nodes_equal(G.nodes(), H.nodes())
(8)                    assert edges_equal(G.edges(), H.edges())
(8)                    self.undirected_multigraph_fh.seek(0)
(8)                    PG = nx.parse_graphml(self.undirected_multigraph_data)
(8)                    assert nodes_equal(G.nodes(), PG.nodes())
```

```
(8)                 assert edges_equal(G.edges(), PG.edges())
(4)             def test_read_undirected_multigraph_no_multiedge_graphml(self):
(8)                 G = self.undirected_multigraph_no_multiedge
(8)                 H = nx.read_graphml(self.undirected_multigraph_no_multiedge_fh)
(8)                 assert nodes_equal(G.nodes(), H.nodes())
(8)                 assert edges_equal(G.edges(), H.edges())
(8)                 self.undirected_multigraph_no_multiedge_fh.seek(0)
(8)                 PG = nx.parse_graphml(self.undirected_multigraph_no_multiedge_data)
(8)                 assert nodes_equal(G.nodes(), PG.nodes())
(8)                 assert edges_equal(G.edges(), PG.edges())
(4)             def test_read_undirected_multigraph_only_ids_for_multiedges_graphml(self):
(8)                 G = self.multigraph_only_ids_for_multiedges
(8)                 H = nx.read_graphml(self.multigraph_only_ids_for_multiedges_fh)
(8)                 assert nodes_equal(G.nodes(), H.nodes())
(8)                 assert edges_equal(G.edges(), H.edges())
(8)                 self.multigraph_only_ids_for_multiedges_fh.seek(0)
(8)                 PG = nx.parse_graphml(self.multigraph_only_ids_for_multiedges_data)
(8)                 assert nodes_equal(G.nodes(), PG.nodes())
(8)                 assert edges_equal(G.edges(), PG.edges())
(4)             def test_read_attribute_graphml(self):
(8)                 G = self.attribute_graph
(8)                 H = nx.read_graphml(self.attribute_fh)
(8)                 assert nodes_equal(G.nodes(True), sorted(H.nodes(data=True)))
(8)                 ge = sorted(G.edges(data=True))
(8)                 he = sorted(H.edges(data=True))
(8)                 for a, b in zip(ge, he):
(12)                    assert a == b
(8)                 self.attribute_fh.seek(0)
(8)                 PG = nx.parse_graphml(self.attribute_data)
(8)                 assert sorted(G.nodes(True)) == sorted(PG.nodes(data=True))
(8)                 ge = sorted(G.edges(data=True))
(8)                 he = sorted(PG.edges(data=True))
(8)                 for a, b in zip(ge, he):
(12)                    assert a == b
(4)             def test_node_default_attribute_graphml(self):
(8)                 G = self.node_attribute_default_graph
(8)                 H = nx.read_graphml(self.node_attribute_default_fh)
(8)                 assert G.graph["node_default"] == H.graph["node_default"]
(4)             def test_directed_edge_in_undirected(self):
(8)                 s = """<?xml version="1.0" encoding="UTF-8"?>
(0)         <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                  http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)           <graph id="G">
(4)             <node id="n0"/>
(4)             <node id="n1"/>
(4)             <node id="n2"/>
(4)             <edge source="n0" target="n1"/>
(4)             <edge source="n1" target="n2" directed='true'/>
(2)           </graph>
(0)         </graphml>"""
(8)                 fh = io.BytesIO(s.encode("UTF-8"))
(8)                 pytest.raises(nx.NetworkXError, nx.read_graphml, fh)
(8)                 pytest.raises(nx.NetworkXError, nx.parse_graphml, s)
(4)             def test_undirected_edge_in_directed(self):
(8)                 s = """<?xml version="1.0" encoding="UTF-8"?>
(0)         <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                  http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)           <graph id="G" edgedefault='directed'>
(4)             <node id="n0"/>
(4)             <node id="n1"/>
(4)             <node id="n2"/>
(4)             <edge source="n0" target="n1"/>
(4)             <edge source="n1" target="n2" directed='false'/>
(2)           </graph>
(0)         </graphml>"""
```

```
(8)                 fh = io.BytesIO(s.encode("UTF-8"))
(8)                 pytest.raises(nx.NetworkXError, nx.read_graphml, fh)
(8)                 pytest.raises(nx.NetworkXError, nx.parse_graphml, s)
(4)             def test_key_raise(self):
(8)                 s = """<?xml version="1.0" encoding="UTF-8"?>
(0)         <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                  http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)           <key id="d0" for="node" attr.name="color" attr.type="string">
(4)             <default>yellow</default>
(2)           </key>
(2)           <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
(2)           <graph id="G" edgedefault="directed">
(4)             <node id="n0">
(6)               <data key="d0">green</data>
(4)             </node>
(4)             <node id="n1"/>
(4)             <node id="n2">
(6)               <data key="d0">blue</data>
(4)             </node>
(4)             <edge id="e0" source="n0" target="n2">
(6)               <data key="d2">1.0</data>
(4)             </edge>
(2)           </graph>
(0)         </graphml>
(0)         """
(8)                 fh = io.BytesIO(s.encode("UTF-8"))
(8)                 pytest.raises(nx.NetworkXError, nx.read_graphml, fh)
(8)                 pytest.raises(nx.NetworkXError, nx.parse_graphml, s)
(4)             def test_hyperedge_raise(self):
(8)                 s = """<?xml version="1.0" encoding="UTF-8"?>
(0)         <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                  http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)           <key id="d0" for="node" attr.name="color" attr.type="string">
(4)             <default>yellow</default>
(2)           </key>
(2)           <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
(2)           <graph id="G" edgedefault="directed">
(4)             <node id="n0">
(6)               <data key="d0">green</data>
(4)             </node>
(4)             <node id="n1"/>
(4)             <node id="n2">
(6)               <data key="d0">blue</data>
(4)             </node>
(4)             <hyperedge id="e0" source="n0" target="n2">
(7)                <endpoint node="n0"/>
(7)                <endpoint node="n1"/>
(7)                <endpoint node="n2"/>
(4)             </hyperedge>
(2)           </graph>
(0)         </graphml>
(0)         """
(8)                 fh = io.BytesIO(s.encode("UTF-8"))
(8)                 pytest.raises(nx.NetworkXError, nx.read_graphml, fh)
(8)                 pytest.raises(nx.NetworkXError, nx.parse_graphml, s)
(4)             def test_multigraph_keys(self):
(8)                 s = """<?xml version="1.0" encoding="UTF-8"?>
(0)         <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                  http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)           <graph id="G" edgedefault="directed">
(4)             <node id="n0"/>
(4)             <node id="n1"/>
(4)             <edge id="e0" source="n0" target="n1"/>
```

```
465: (4)                <edge id="e1" source="n0" target="n1"/>
466: (2)              </graph>
467: (0)            </graphml>
468: (0)            """
469: (8)                    fh = io.BytesIO(s.encode("UTF-8"))
470: (8)                    G = nx.read_graphml(fh)
471: (8)                    expected = [("n0", "n1", "e0"), ("n0", "n1", "e1")]
472: (8)                    assert sorted(G.edges(keys=True)) == expected
473: (8)                    fh.seek(0)
474: (8)                    H = nx.parse_graphml(s)
475: (8)                    assert sorted(H.edges(keys=True)) == expected
476: (4)                def test_preserve_multi_edge_data(self):
477: (8)                    """
478: (8)                    Test that data and keys of edges are preserved on consequent
479: (8)                    write and reads
480: (8)                    """
481: (8)                    G = nx.MultiGraph()
482: (8)                    G.add_node(1)
483: (8)                    G.add_node(2)
484: (8)                    G.add_edges_from(
485: (12)                       [
486: (16)                           (1, 2),
487: (16)                           (1, 2, {"key": "data_key1"}),
488: (16)                           (1, 2, {"id": "data_id2"}),
489: (16)                           (1, 2, {"key": "data_key3", "id": "data_id3"}),
490: (16)                           (1, 2, 103, {"key": "data_key4"}),
491: (16)                           (1, 2, 104, {"id": "data_id5"}),
492: (16)                           (1, 2, 105, {"key": "data_key6", "id": "data_id7"}),
493: (12)                       ]
494: (8)                    )
495: (8)                    fh = io.BytesIO()
496: (8)                    nx.write_graphml(G, fh)
497: (8)                    fh.seek(0)
498: (8)                    H = nx.read_graphml(fh, node_type=int)
499: (8)                    assert edges_equal(G.edges(data=True, keys=True), H.edges(data=True,
keys=True))
500: (8)                    assert G._adj == H._adj
501: (8)                    Gadj = {
502: (12)                       str(node): {
503: (16)                           str(nbr): {str(ekey): dd for ekey, dd in key_dict.items()}
504: (16)                           for nbr, key_dict in nbr_dict.items()
505: (12)                       }
506: (12)                       for node, nbr_dict in G._adj.items()
507: (8)                    }
508: (8)                    fh.seek(0)
509: (8)                    HH = nx.read_graphml(fh, node_type=str, edge_key_type=str)
510: (8)                    assert Gadj == HH._adj
511: (8)                    fh.seek(0)
512: (8)                    string_fh = fh.read()
513: (8)                    HH = nx.parse_graphml(string_fh, node_type=str, edge_key_type=str)
514: (8)                    assert Gadj == HH._adj
515: (4)                def test_yfiles_extension(self):
516: (8)                    data = """<?xml version="1.0" encoding="UTF-8" standalone="no"?>
517: (0)            <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
518: (9)                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
519: (9)                     xmlns:y="http://www.yworks.com/xml/graphml"
520: (9)                     xmlns:yed="http://www.yworks.com/xml/yed/3"
521: (9)                     xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
522: (9)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
523: (2)              <!--Created by yFiles for Java 2.7-->
524: (2)              <key for="graphml" id="d0" yfiles.type="resources"/>
525: (2)              <key attr.name="url" attr.type="string" for="node" id="d1"/>
526: (2)              <key attr.name="description" attr.type="string" for="node" id="d2"/>
527: (2)              <key for="node" id="d3" yfiles.type="nodegraphics"/>
528: (2)              <key attr.name="Description" attr.type="string" for="graph" id="d4">
529: (4)                <default/>
530: (2)              </key>
531: (2)              <key attr.name="url" attr.type="string" for="edge" id="d5"/>
532: (2)              <key attr.name="description" attr.type="string" for="edge" id="d6"/>
```

```
(2)  <key for="edge" id="d7" yfiles.type="edgegraphics"/>
(2)  <graph edgedefault="directed" id="G">
(4)    <node id="n0">
(6)      <data key="d3">
(8)        <y:ShapeNode>
(10)         <y:Geometry height="30.0" width="30.0" x="125.0" y="100.0"/>
(10)         <y:Fill color="#FFCC00" transparent="false"/>
(10)         <y:BorderStyle color="#000000" type="line" width="1.0"/>
(10)         <y:NodeLabel alignment="center" autoSizePolicy="content"
(11)          borderDistance="0.0" fontFamily="Dialog" fontSize="13"
(11)          fontStyle="plain" hasBackgroundColor="false" hasLineColor="false"
(11)          height="19.1328125" modelName="internal" modelPosition="c"
(11)          textColor="#000000" visible="true" width="12.27099609375"
(11)          x="8.864501953125" y="5.43359375">1</y:NodeLabel>
(10)         <y:Shape type="rectangle"/>
(8)        </y:ShapeNode>
(6)      </data>
(4)    </node>
(4)    <node id="n1">
(6)      <data key="d3">
(8)        <y:ShapeNode>
(10)         <y:Geometry height="30.0" width="30.0" x="183.0" y="205.0"/>
(10)         <y:Fill color="#FFCC00" transparent="false"/>
(10)         <y:BorderStyle color="#000000" type="line" width="1.0"/>
(10)         <y:NodeLabel alignment="center" autoSizePolicy="content"
(10)         borderDistance="0.0" fontFamily="Dialog" fontSize="13"
(10)         fontStyle="plain" hasBackgroundColor="false" hasLineColor="false"
(10)         height="19.1328125" modelName="internal" modelPosition="c"
(10)         textColor="#000000" visible="true" width="12.27099609375"
(10)         x="8.864501953125" y="5.43359375">2</y:NodeLabel>
(10)         <y:Shape type="rectangle"/>
(8)        </y:ShapeNode>
(6)      </data>
(4)    </node>
(4)    <node id="n2">
(6)      <data key="d6" xml:space="preserve"><![CDATA[description
(0)  line1
(0)  line2]]></data>
(6)      <data key="d3">
(8)        <y:GenericNode configuration="com.yworks.flowchart.terminator">
(10)         <y:Geometry height="40.0" width="80.0" x="950.0" y="286.0"/>
(10)         <y:Fill color="#E8EEF7" color2="#B7C9E3" transparent="false"/>
(10)         <y:BorderStyle color="#000000" type="line" width="1.0"/>
(10)         <y:NodeLabel alignment="center" autoSizePolicy="content"
(10)         fontFamily="Dialog" fontSize="12" fontStyle="plain"
(10)         hasBackgroundColor="false" hasLineColor="false" height="17.96875"
(10)         horizontalTextPosition="center" iconTextGap="4" modelName="custom"
(10)         textColor="#000000" verticalTextPosition="bottom" visible="true"
(10)         width="67.984375" x="6.0078125" xml:space="preserve"
(10)         y="11.015625">3<y:LabelModel>
(10)         <y:SmartNodeLabelModel distance="4.0"/></y:LabelModel>
(10)         <y:ModelParameter><y:SmartNodeLabelModelParameter labelRatioX="0.0"
(10)         labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0"
(10)         offsetY="0.0" upX="0.0" upY="-1.0"/></y:ModelParameter>
</y:NodeLabel>
(8)        </y:GenericNode>
(6)      </data>
(4)    </node>
(4)    <edge id="e0" source="n0" target="n1">
(6)      <data key="d7">
(8)        <y:PolyLineEdge>
(10)         <y:Path sx="0.0" sy="0.0" tx="0.0" ty="0.0"/>
(10)         <y:LineStyle color="#000000" type="line" width="1.0"/>
(10)         <y:Arrows source="none" target="standard"/>
(10)         <y:BendStyle smoothed="false"/>
(8)        </y:PolyLineEdge>
(6)      </data>
(4)    </edge>
(2)  </graph>
```

```
601: (2)              <data key="d0">
602: (4)                <y:Resources/>
603: (2)              </data>
604: (0)            </graphml>
605: (0)            """
606: (8)                    fh = io.BytesIO(data.encode("UTF-8"))
607: (8)                    G = nx.read_graphml(fh, force_multigraph=True)
608: (8)                    assert list(G.edges()) == [("n0", "n1")]
609: (8)                    assert G.has_edge("n0", "n1", key="e0")
610: (8)                    assert G.nodes["n0"]["label"] == "1"
611: (8)                    assert G.nodes["n1"]["label"] == "2"
612: (8)                    assert G.nodes["n2"]["label"] == "3"
613: (8)                    assert G.nodes["n0"]["shape_type"] == "rectangle"
614: (8)                    assert G.nodes["n1"]["shape_type"] == "rectangle"
615: (8)                    assert G.nodes["n2"]["shape_type"] ==
"com.yworks.flowchart.terminator"
616: (8)                    assert G.nodes["n2"]["description"] == "description\nline1\nline2"
617: (8)                    fh.seek(0)
618: (8)                    G = nx.read_graphml(fh)
619: (8)                    assert list(G.edges()) == [("n0", "n1")]
620: (8)                    assert G["n0"]["n1"]["id"] == "e0"
621: (8)                    assert G.nodes["n0"]["label"] == "1"
622: (8)                    assert G.nodes["n1"]["label"] == "2"
623: (8)                    assert G.nodes["n2"]["label"] == "3"
624: (8)                    assert G.nodes["n0"]["shape_type"] == "rectangle"
625: (8)                    assert G.nodes["n1"]["shape_type"] == "rectangle"
626: (8)                    assert G.nodes["n2"]["shape_type"] ==
"com.yworks.flowchart.terminator"
627: (8)                    assert G.nodes["n2"]["description"] == "description\nline1\nline2"
628: (8)                    H = nx.parse_graphml(data, force_multigraph=True)
629: (8)                    assert list(H.edges()) == [("n0", "n1")]
630: (8)                    assert H.has_edge("n0", "n1", key="e0")
631: (8)                    assert H.nodes["n0"]["label"] == "1"
632: (8)                    assert H.nodes["n1"]["label"] == "2"
633: (8)                    assert H.nodes["n2"]["label"] == "3"
634: (8)                    H = nx.parse_graphml(data)
635: (8)                    assert list(H.edges()) == [("n0", "n1")]
636: (8)                    assert H["n0"]["n1"]["id"] == "e0"
637: (8)                    assert H.nodes["n0"]["label"] == "1"
638: (8)                    assert H.nodes["n1"]["label"] == "2"
639: (8)                    assert H.nodes["n2"]["label"] == "3"
640: (4)                def test_bool(self):
641: (8)                    s = """<?xml version="1.0" encoding="UTF-8"?>
642: (0)            <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
643: (9)                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
644: (9)                     xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
645: (9)                     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
646: (2)              <key id="d0" for="node" attr.name="test" attr.type="boolean">
647: (4)                <default>false</default>
648: (2)              </key>
649: (2)              <graph id="G" edgedefault="directed">
650: (4)                <node id="n0">
651: (6)                  <data key="d0">true</data>
652: (4)                </node>
653: (4)                <node id="n1"/>
654: (4)                <node id="n2">
655: (6)                  <data key="d0">false</data>
656: (4)                </node>
657: (4)                <node id="n3">
658: (6)                  <data key="d0">FaLsE</data>
659: (4)                </node>
660: (4)                <node id="n4">
661: (6)                  <data key="d0">True</data>
662: (4)                </node>
663: (4)                <node id="n5">
664: (6)                  <data key="d0">0</data>
665: (4)                </node>
666: (4)                <node id="n6">
667: (6)                  <data key="d0">1</data>
```

```
(4)                 </node>
(2)               </graph>
(0)             </graphml>
(0)             """
(8)                     fh = io.BytesIO(s.encode("UTF-8"))
(8)                     G = nx.read_graphml(fh)
(8)                     H = nx.parse_graphml(s)
(8)                     for graph in [G, H]:
(12)                        assert graph.nodes["n0"]["test"]
(12)                        assert not graph.nodes["n2"]["test"]
(12)                        assert not graph.nodes["n3"]["test"]
(12)                        assert graph.nodes["n4"]["test"]
(12)                        assert not graph.nodes["n5"]["test"]
(12)                        assert graph.nodes["n6"]["test"]
(4)                 def test_graphml_header_line(self):
(8)                     good = """<?xml version="1.0" encoding="UTF-8" standalone="no"?>
(0)             <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                      xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                      http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)               <key id="d0" for="node" attr.name="test" attr.type="boolean">
(4)                 <default>false</default>
(2)               </key>
(2)               <graph id="G">
(4)                 <node id="n0">
(6)                   <data key="d0">true</data>
(4)                 </node>
(2)               </graph>
(0)             </graphml>
(0)             """
(8)                     bad = """<?xml version="1.0" encoding="UTF-8" standalone="no"?>
(0)             <graphml>
(2)               <key id="d0" for="node" attr.name="test" attr.type="boolean">
(4)                 <default>false</default>
(2)               </key>
(2)               <graph id="G">
(4)                 <node id="n0">
(6)                   <data key="d0">true</data>
(4)                 </node>
(2)               </graph>
(0)             </graphml>
(0)             """
(8)                     ugly = """<?xml version="1.0" encoding="UTF-8" standalone="no"?>
(0)             <graphml xmlns="https://ghghgh">
(2)               <key id="d0" for="node" attr.name="test" attr.type="boolean">
(4)                 <default>false</default>
(2)               </key>
(2)               <graph id="G">
(4)                 <node id="n0">
(6)                   <data key="d0">true</data>
(4)                 </node>
(2)               </graph>
(0)             </graphml>
(0)             """
(8)                     for s in (good, bad):
(12)                        fh = io.BytesIO(s.encode("UTF-8"))
(12)                        G = nx.read_graphml(fh)
(12)                        H = nx.parse_graphml(s)
(12)                        for graph in [G, H]:
(16)                            assert graph.nodes["n0"]["test"]
(8)                     fh = io.BytesIO(ugly.encode("UTF-8"))
(8)                     pytest.raises(nx.NetworkXError, nx.read_graphml, fh)
(8)                     pytest.raises(nx.NetworkXError, nx.parse_graphml, ugly)
(4)                 def test_read_attributes_with_groups(self):
(8)                     data = """\
(0)             <?xml version="1.0" encoding="UTF-8" standalone="no"?>
(0)             <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:java="http://www.yworks.com/xml/yfiles-common/1.0/java"
xmlns:sys="http://www.yworks.com/xml/yfiles-common/markup/primitives/2.0"
```

```
xmlns:x="http://www.yworks.com/xml/yfiles-common/markup/2.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:y="http://www.yworks.com/xml/graphml"
xmlns:yed="http://www.yworks.com/xml/yed/3"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd">
735: (2)              <!--Created by yEd 3.17-->
736: (2)              <key attr.name="Description" attr.type="string" for="graph" id="d0"/>
737: (2)              <key for="port" id="d1" yfiles.type="portgraphics"/>
738: (2)              <key for="port" id="d2" yfiles.type="portgeometry"/>
739: (2)              <key for="port" id="d3" yfiles.type="portuserdata"/>
740: (2)              <key attr.name="CustomProperty" attr.type="string" for="node" id="d4">
741: (4)                <default/>
742: (2)              </key>
743: (2)              <key attr.name="url" attr.type="string" for="node" id="d5"/>
744: (2)              <key attr.name="description" attr.type="string" for="node" id="d6"/>
745: (2)              <key for="node" id="d7" yfiles.type="nodegraphics"/>
746: (2)              <key for="graphml" id="d8" yfiles.type="resources"/>
747: (2)              <key attr.name="url" attr.type="string" for="edge" id="d9"/>
748: (2)              <key attr.name="description" attr.type="string" for="edge" id="d10"/>
749: (2)              <key for="edge" id="d11" yfiles.type="edgegraphics"/>
750: (2)              <graph edgedefault="directed" id="G">
751: (4)                <data key="d0"/>
752: (4)                <node id="n0">
753: (6)                  <data key="d4"><![CDATA[CustomPropertyValue]]></data>
754: (6)                  <data key="d6"/>
755: (6)                  <data key="d7">
756: (8)                    <y:ShapeNode>
757: (10)                     <y:Geometry height="30.0" width="30.0" x="125.0"
y="-255.4611111111111"/>
758: (10)                     <y:Fill color="#FFCC00" transparent="false"/>
759: (10)                     <y:BorderStyle color="#000000" raised="false" type="line"
width="1.0"/>
760: (10)                     <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" horizontalTextPosition="center" iconTextGap="4"
modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="true"
width="11.634765625" x="9.1826171875" y="6.015625">2<y:LabelModel>
761: (14)                       <y:SmartNodeLabelModel distance="4.0"/>
762: (12)                     </y:LabelModel>
763: (12)                     <y:ModelParameter>
764: (14)                       <y:SmartNodeLabelModelParameter labelRatioX="0.0"
labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0" offsetY="0.0" upX="0.0"
upY="-1.0"/>
765: (12)                     </y:ModelParameter>
766: (10)                     </y:NodeLabel>
767: (10)                     <y:Shape type="rectangle"/>
768: (8)                    </y:ShapeNode>
769: (6)                  </data>
770: (4)                </node>
771: (4)                <node id="n1" yfiles.foldertype="group">
772: (6)                  <data key="d4"><![CDATA[CustomPropertyValue]]></data>
773: (6)                  <data key="d5"/>
774: (6)                  <data key="d6"/>
775: (6)                  <data key="d7">
776: (8)                    <y:ProxyAutoBoundsNode>
777: (10)                     <y:Realizers active="0">
778: (12)                       <y:GroupNode>
779: (14)                         <y:Geometry height="250.38333333333333" width="140.0" x="-30.0"
y="-330.3833333333333"/>
780: (14)                         <y:Fill color="#F5F5F5" transparent="false"/>
781: (14)                         <y:BorderStyle color="#000000" type="dashed" width="1.0"/>
782: (14)                         <y:NodeLabel alignment="right" autoSizePolicy="node_width"
backgroundColor="#EBEBEB" borderDistance="0.0" fontFamily="Dialog" fontSize="15" fontStyle="plain"
hasLineColor="false" height="21.4609375" horizontalTextPosition="center" iconTextGap="4"
modelName="internal" modelPosition="t" textColor="#000000" verticalTextPosition="bottom"
visible="true" width="140.0" x="0.0" y="0.0">Group 3</y:NodeLabel>
783: (14)                         <y:Shape type="roundrectangle"/>
784: (14)                         <y:State closed="false" closedHeight="50.0" closedWidth="50.0"
innerGraphDisplayEnabled="false"/>
```

```
785: (14)                          <y:Insets bottom="15" bottomF="15.0" left="15" leftF="15.0"
right="15" rightF="15.0" top="15" topF="15.0"/>
786: (14)                          <y:BorderInsets bottom="1" bottomF="1.0" left="0" leftF="0.0"
right="0" rightF="0.0" top="1" topF="1.0001736111111086"/>
787: (12)                        </y:GroupNode>
788: (12)                        <y:GroupNode>
789: (14)                          <y:Geometry height="50.0" width="50.0" x="0.0" y="60.0"/>
790: (14)                          <y:Fill color="#F5F5F5" transparent="false"/>
791: (14)                          <y:BorderStyle color="#000000" type="dashed" width="1.0"/>
792: (14)                          <y:NodeLabel alignment="right" autoSizePolicy="node_width"
backgroundColor="#EBEBEB" borderDistance="0.0" fontFamily="Dialog" fontSize="15" fontStyle="plain"
hasLineColor="false" height="21.4609375" horizontalTextPosition="center" iconTextGap="4"
modelName="internal" modelPosition="t" textColor="#000000" verticalTextPosition="bottom"
visible="true" width="65.201171875" x="-7.6005859375" y="0.0">Folder 3</y:NodeLabel>
793: (14)                          <y:Shape type="roundrectangle"/>
794: (14)                          <y:State closed="true" closedHeight="50.0" closedWidth="50.0"
innerGraphDisplayEnabled="false"/>
795: (14)                          <y:Insets bottom="5" bottomF="5.0" left="5" leftF="5.0"
right="5" rightF="5.0" top="5" topF="5.0"/>
796: (14)                          <y:BorderInsets bottom="0" bottomF="0.0" left="0" leftF="0.0"
right="0" rightF="0.0" top="0" topF="0.0"/>
797: (12)                        </y:GroupNode>
798: (10)                      </y:Realizers>
799: (8)                    </y:ProxyAutoBoundsNode>
800: (6)                  </data>
801: (6)                  <graph edgedefault="directed" id="n1:">
802: (8)                    <node id="n1::n0" yfiles.foldertype="group">
803: (10)                      <data key="d4"><![CDATA[CustomPropertyValue]]></data>
804: (10)                      <data key="d5"/>
805: (10)                      <data key="d6"/>
806: (10)                      <data key="d7">
807: (12)                        <y:ProxyAutoBoundsNode>
808: (14)                          <y:Realizers active="0">
809: (16)                            <y:GroupNode>
810: (18)                              <y:Geometry height="83.46111111111111" width="110.0"
x="-15.0" y="-292.9222222222222"/>
811: (18)                              <y:Fill color="#F5F5F5" transparent="false"/>
812: (18)                              <y:BorderStyle color="#000000" type="dashed" width="1.0"/>
813: (18)                              <y:NodeLabel alignment="right" autoSizePolicy="node_width"
backgroundColor="#EBEBEB" borderDistance="0.0" fontFamily="Dialog" fontSize="15" fontStyle="plain"
hasLineColor="false" height="21.4609375" horizontalTextPosition="center" iconTextGap="4"
modelName="internal" modelPosition="t" textColor="#000000" verticalTextPosition="bottom"
visible="true" width="110.0" x="0.0" y="0.0">Group 1</y:NodeLabel>
814: (18)                              <y:Shape type="roundrectangle"/>
815: (18)                              <y:State closed="false" closedHeight="50.0"
closedWidth="50.0" innerGraphDisplayEnabled="false"/>
816: (18)                              <y:Insets bottom="15" bottomF="15.0" left="15" leftF="15.0"
right="15" rightF="15.0" top="15" topF="15.0"/>
817: (18)                              <y:BorderInsets bottom="1" bottomF="1.0" left="0"
leftF="0.0" right="0" rightF="0.0" top="1" topF="1.0001736111111086"/>
818: (16)                            </y:GroupNode>
819: (16)                            <y:GroupNode>
820: (18)                              <y:Geometry height="50.0" width="50.0" x="0.0" y="60.0"/>
821: (18)                              <y:Fill color="#F5F5F5" transparent="false"/>
822: (18)                              <y:BorderStyle color="#000000" type="dashed" width="1.0"/>
823: (18)                              <y:NodeLabel alignment="right" autoSizePolicy="node_width"
backgroundColor="#EBEBEB" borderDistance="0.0" fontFamily="Dialog" fontSize="15" fontStyle="plain"
hasLineColor="false" height="21.4609375" horizontalTextPosition="center" iconTextGap="4"
modelName="internal" modelPosition="t" textColor="#000000" verticalTextPosition="bottom"
visible="true" width="65.201171875" x="-7.6005859375" y="0.0">Folder 1</y:NodeLabel>
824: (18)                              <y:Shape type="roundrectangle"/>
825: (18)                              <y:State closed="true" closedHeight="50.0"
closedWidth="50.0" innerGraphDisplayEnabled="false"/>
826: (18)                              <y:Insets bottom="5" bottomF="5.0" left="5" leftF="5.0"
right="5" rightF="5.0" top="5" topF="5.0"/>
827: (18)                              <y:BorderInsets bottom="0" bottomF="0.0" left="0"
leftF="0.0" right="0" rightF="0.0" top="0" topF="0.0"/>
828: (16)                            </y:GroupNode>
829: (14)                          </y:Realizers>
```

```
830: (12)                      </y:ProxyAutoBoundsNode>
831: (10)                    </data>
832: (10)                    <graph edgedefault="directed" id="n1::n0:">
833: (12)                      <node id="n1::n0::n0">
834: (14)                        <data key="d4"><![CDATA[CustomPropertyValue]]></data>
835: (14)                        <data key="d6"/>
836: (14)                        <data key="d7">
837: (16)                          <y:ShapeNode>
838: (18)                            <y:Geometry height="30.0" width="30.0" x="50.0"
y="-255.4611111111111"/>
839: (18)                            <y:Fill color="#FFCC00" transparent="false"/>
840: (18)                            <y:BorderStyle color="#000000" raised="false" type="line"
width="1.0"/>
841: (18)                            <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" horizontalTextPosition="center" iconTextGap="4"
modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="true"
width="11.634765625" x="9.1826171875" y="6.015625">1<y:LabelModel>
842: (22)                                <y:SmartNodeLabelModel distance="4.0"/>
843: (20)                              </y:LabelModel>
844: (20)                              <y:ModelParameter>
845: (22)                                <y:SmartNodeLabelModelParameter labelRatioX="0.0"
labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0" offsetY="0.0" upX="0.0"
upY="-1.0"/>
846: (20)                              </y:ModelParameter>
847: (18)                            </y:NodeLabel>
848: (18)                            <y:Shape type="rectangle"/>
849: (16)                          </y:ShapeNode>
850: (14)                        </data>
851: (12)                      </node>
852: (12)                      <node id="n1::n0::n1">
853: (14)                        <data key="d4"><![CDATA[CustomPropertyValue]]></data>
854: (14)                        <data key="d6"/>
855: (14)                        <data key="d7">
856: (16)                          <y:ShapeNode>
857: (18)                            <y:Geometry height="30.0" width="30.0" x="0.0"
y="-255.4611111111111"/>
858: (18)                            <y:Fill color="#FFCC00" transparent="false"/>
859: (18)                            <y:BorderStyle color="#000000" raised="false" type="line"
width="1.0"/>
860: (18)                            <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" horizontalTextPosition="center" iconTextGap="4"
modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="true"
width="11.634765625" x="9.1826171875" y="6.015625">3<y:LabelModel>
861: (22)                                <y:SmartNodeLabelModel distance="4.0"/>
862: (20)                              </y:LabelModel>
863: (20)                              <y:ModelParameter>
864: (22)                                <y:SmartNodeLabelModelParameter labelRatioX="0.0"
labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0" offsetY="0.0" upX="0.0"
upY="-1.0"/>
865: (20)                              </y:ModelParameter>
866: (18)                            </y:NodeLabel>
867: (18)                            <y:Shape type="rectangle"/>
868: (16)                          </y:ShapeNode>
869: (14)                        </data>
870: (12)                      </node>
871: (10)                    </graph>
872: (8)                  </node>
873: (8)                  <node id="n1::n1" yfiles.foldertype="group">
874: (10)                    <data key="d4"><![CDATA[CustomPropertyValue]]></data>
875: (10)                    <data key="d5"/>
876: (10)                    <data key="d6"/>
877: (10)                    <data key="d7">
878: (12)                      <y:ProxyAutoBoundsNode>
879: (14)                        <y:Realizers active="0">
880: (16)                          <y:GroupNode>
881: (18)                            <y:Geometry height="83.46111111111111" width="110.0"
x="-15.0" y="-179.4611111111111"/>
```

```
882: (18)                              <y:Fill color="#F5F5F5" transparent="false"/>
883: (18)                              <y:BorderStyle color="#000000" type="dashed" width="1.0"/>
884: (18)                              <y:NodeLabel alignment="right" autoSizePolicy="node_width"
backgroundColor="#EBEBEB" borderDistance="0.0" fontFamily="Dialog" fontSize="15" fontStyle="plain"
hasLineColor="false" height="21.4609375" horizontalTextPosition="center" iconTextGap="4"
modelName="internal" modelPosition="t" textColor="#000000" verticalTextPosition="bottom"
visible="true" width="110.0" x="0.0" y="0.0">Group 2</y:NodeLabel>
885: (18)                              <y:Shape type="roundrectangle"/>
886: (18)                              <y:State closed="false" closedHeight="50.0"
closedWidth="50.0" innerGraphDisplayEnabled="false"/>
887: (18)                              <y:Insets bottom="15" bottomF="15.0" left="15" leftF="15.0"
right="15" rightF="15.0" top="15" topF="15.0"/>
888: (18)                              <y:BorderInsets bottom="1" bottomF="1.0" left="0"
leftF="0.0" right="0" rightF="0.0" top="1" topF="1.0001736111111086"/>
889: (16)                            </y:GroupNode>
890: (16)                            <y:GroupNode>
891: (18)                              <y:Geometry height="50.0" width="50.0" x="0.0" y="60.0"/>
892: (18)                              <y:Fill color="#F5F5F5" transparent="false"/>
893: (18)                              <y:BorderStyle color="#000000" type="dashed" width="1.0"/>
894: (18)                              <y:NodeLabel alignment="right" autoSizePolicy="node_width"
backgroundColor="#EBEBEB" borderDistance="0.0" fontFamily="Dialog" fontSize="15" fontStyle="plain"
hasLineColor="false" height="21.4609375" horizontalTextPosition="center" iconTextGap="4"
modelName="internal" modelPosition="t" textColor="#000000" verticalTextPosition="bottom"
visible="true" width="65.201171875" x="-7.6005859375" y="0.0">Folder 2</y:NodeLabel>
895: (18)                              <y:Shape type="roundrectangle"/>
896: (18)                              <y:State closed="true" closedHeight="50.0"
closedWidth="50.0" innerGraphDisplayEnabled="false"/>
897: (18)                              <y:Insets bottom="5" bottomF="5.0" left="5" leftF="5.0"
right="5" rightF="5.0" top="5" topF="5.0"/>
898: (18)                              <y:BorderInsets bottom="0" bottomF="0.0" left="0"
leftF="0.0" right="0" rightF="0.0" top="0" topF="0.0"/>
899: (16)                            </y:GroupNode>
900: (14)                          </y:Realizers>
901: (12)                        </y:ProxyAutoBoundsNode>
902: (10)                      </data>
903: (10)                      <graph edgedefault="directed" id="n1::n1:">
904: (12)                        <node id="n1::n1::n0">
905: (14)                          <data key="d4"><![CDATA[CustomPropertyValue]]></data>
906: (14)                          <data key="d6"/>
907: (14)                          <data key="d7">
908: (16)                            <y:ShapeNode>
909: (18)                              <y:Geometry height="30.0" width="30.0" x="0.0" y="-142.0"/>
910: (18)                              <y:Fill color="#FFCC00" transparent="false"/>
911: (18)                              <y:BorderStyle color="#000000" raised="false" type="line"
width="1.0"/>
912: (18)                              <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" horizontalTextPosition="center" iconTextGap="4"
modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="true"
width="11.634765625" x="9.1826171875" y="6.015625">5<y:LabelModel>
913: (22)                                  <y:SmartNodeLabelModel distance="4.0"/>
914: (20)                                </y:LabelModel>
915: (20)                                <y:ModelParameter>
916: (22)                                  <y:SmartNodeLabelModelParameter labelRatioX="0.0"
labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0" offsetY="0.0" upX="0.0"
upY="-1.0"/>
917: (20)                                </y:ModelParameter>
918: (18)                              </y:NodeLabel>
919: (18)                              <y:Shape type="rectangle"/>
920: (16)                            </y:ShapeNode>
921: (14)                          </data>
922: (12)                        </node>
923: (12)                        <node id="n1::n1::n1">
924: (14)                          <data key="d4"><![CDATA[CustomPropertyValue]]></data>
925: (14)                          <data key="d6"/>
926: (14)                          <data key="d7">
927: (16)                            <y:ShapeNode>
928: (18)                              <y:Geometry height="30.0" width="30.0" x="50.0" y="-142.0"/>
929: (18)                              <y:Fill color="#FFCC00" transparent="false"/>
```

```
930: (18)                            <y:BorderStyle color="#000000" raised="false" type="line"
width="1.0"/>
931: (18)                            <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" horizontalTextPosition="center" iconTextGap="4"
modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="true"
width="11.634765625" x="9.1826171875" y="6.015625">6<y:LabelModel>
932: (22)                                <y:SmartNodeLabelModel distance="4.0"/>
933: (20)                              </y:LabelModel>
934: (20)                              <y:ModelParameter>
935: (22)                                <y:SmartNodeLabelModelParameter labelRatioX="0.0"
labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0" offsetY="0.0" upX="0.0"
upY="-1.0"/>
936: (20)                              </y:ModelParameter>
937: (18)                            </y:NodeLabel>
938: (18)                            <y:Shape type="rectangle"/>
939: (16)                          </y:ShapeNode>
940: (14)                        </data>
941: (12)                      </node>
942: (10)                    </graph>
943: (8)                  </node>
944: (6)                </graph>
945: (4)              </node>
946: (4)              <node id="n2">
947: (6)                <data key="d4"><![CDATA[CustomPropertyValue]]></data>
948: (6)                <data key="d6"/>
949: (6)                <data key="d7">
950: (8)                  <y:ShapeNode>
951: (10)                   <y:Geometry height="30.0" width="30.0" x="125.0" y="-142.0"/>
952: (10)                   <y:Fill color="#FFCC00" transparent="false"/>
953: (10)                   <y:BorderStyle color="#000000" raised="false" type="line"
width="1.0"/>
954: (10)                   <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" horizontalTextPosition="center" iconTextGap="4"
modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="true"
width="11.634765625" x="9.1826171875" y="6.015625">9<y:LabelModel>
955: (14)                        <y:SmartNodeLabelModel distance="4.0"/>
956: (12)                      </y:LabelModel>
957: (12)                      <y:ModelParameter>
958: (14)                        <y:SmartNodeLabelModelParameter labelRatioX="0.0"
labelRatioY="0.0" nodeRatioX="0.0" nodeRatioY="0.0" offsetX="0.0" offsetY="0.0" upX="0.0"
upY="-1.0"/>
959: (12)                      </y:ModelParameter>
960: (10)                   </y:NodeLabel>
961: (10)                   <y:Shape type="rectangle"/>
962: (8)                  </y:ShapeNode>
963: (6)                </data>
964: (4)              </node>
965: (4)              <edge id="n1::n1::e0" source="n1::n1::n0" target="n1::n1::n1">
966: (6)                <data key="d10"/>
967: (6)                <data key="d11">
968: (8)                  <y:PolyLineEdge>
969: (10)                   <y:Path sx="15.0" sy="-0.0" tx="-15.0" ty="-0.0"/>
970: (10)                   <y:LineStyle color="#000000" type="line" width="1.0"/>
971: (10)                   <y:Arrows source="none" target="standard"/>
972: (10)                   <y:BendStyle smoothed="false"/>
973: (8)                  </y:PolyLineEdge>
974: (6)                </data>
975: (4)              </edge>
976: (4)              <edge id="n1::n0::e0" source="n1::n0::n1" target="n1::n0::n0">
977: (6)                <data key="d10"/>
978: (6)                <data key="d11">
979: (8)                  <y:PolyLineEdge>
980: (10)                   <y:Path sx="15.0" sy="-0.0" tx="-15.0" ty="-0.0"/>
981: (10)                   <y:LineStyle color="#000000" type="line" width="1.0"/>
982: (10)                   <y:Arrows source="none" target="standard"/>
983: (10)                   <y:BendStyle smoothed="false"/>
984: (8)                  </y:PolyLineEdge>
```

```
(6)                 </data>
(4)               </edge>
(4)               <edge id="e0" source="n1::n0::n0" target="n0">
(6)                 <data key="d10"/>
(6)                 <data key="d11">
(8)                   <y:PolyLineEdge>
(10)                    <y:Path sx="15.0" sy="-0.0" tx="-15.0" ty="-0.0"/>
(10)                    <y:LineStyle color="#000000" type="line" width="1.0"/>
(10)                    <y:Arrows source="none" target="standard"/>
(10)                    <y:BendStyle smoothed="false"/>
(8)                   </y:PolyLineEdge>
(6)                 </data>
(4)               </edge>
(4)               <edge id="e1" source="n1::n1::n1" target="n2">
(6)                 <data key="d10"/>
(6)                 <data key="d11">
(8)                   <y:PolyLineEdge>
(10)                    <y:Path sx="15.0" sy="-0.0" tx="-15.0" ty="-0.0"/>
(10)                    <y:LineStyle color="#000000" type="line" width="1.0"/>
(10)                    <y:Arrows source="none" target="standard"/>
(10)                    <y:BendStyle smoothed="false"/>
(8)                   </y:PolyLineEdge>
(6)                 </data>
(4)               </edge>
(2)             </graph>
(2)             <data key="d8">
(4)               <y:Resources/>
(2)             </data>
(0)           </graphml>
(0)           """
(8)                   fh = io.BytesIO(data.encode("UTF-8"))
(8)                   G = nx.read_graphml(fh)
(8)                   data = [x for _, x in G.nodes(data=True)]
(8)                   assert len(data) == 9
(8)                   for node_data in data:
(12)                      assert node_data["CustomProperty"] != ""
(4)               def test_long_attribute_type(self):
(8)                   s = """<?xml version='1.0' encoding='utf-8'?>
(0)           <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
(9)                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(9)                    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
(9)                    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
(2)             <key attr.name="cudfversion" attr.type="long" for="node" id="d6" />
(2)             <graph edgedefault="directed">
(4)               <node id="n1">
(6)                 <data key="d6">4284</data>
(4)               </node>
(2)             </graph>
(0)           </graphml>"""
(8)                   fh = io.BytesIO(s.encode("UTF-8"))
(8)                   G = nx.read_graphml(fh)
(8)                   expected = [("n1", {"cudfversion": 4284})]
(8)                   assert sorted(G.nodes(data=True)) == expected
(8)                   fh.seek(0)
(8)                   H = nx.parse_graphml(s)
(8)                   assert sorted(H.nodes(data=True)) == expected
(0)           class TestWriteGraphML(BaseGraphML):
(4)               writer = staticmethod(nx.write_graphml_lxml)
(4)               @classmethod
(4)               def setup_class(cls):
(8)                   BaseGraphML.setup_class()
(8)                   _ = pytest.importorskip("lxml.etree")
(4)               def test_write_interface(self):
(8)                   try:
(12)                      import lxml.etree
(12)                      assert nx.write_graphml == nx.write_graphml_lxml
(8)                   except ImportError:
(12)                      assert nx.write_graphml == nx.write_graphml_xml
(4)               def test_write_read_simple_directed_graphml(self):
```

```
(8)                 G = self.simple_directed_graph
(8)                 G.graph["hi"] = "there"
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 assert sorted(G.nodes()) == sorted(H.nodes())
(8)                 assert sorted(G.edges()) == sorted(H.edges())
(8)                 assert sorted(G.edges(data=True)) == sorted(H.edges(data=True))
(8)                 self.simple_directed_fh.seek(0)
(4)             def test_GraphMLWriter_add_graphs(self):
(8)                 gmlw = GraphMLWriter()
(8)                 G = self.simple_directed_graph
(8)                 H = G.copy()
(8)                 gmlw.add_graphs([G, H])
(4)             def test_write_read_simple_no_prettyprint(self):
(8)                 G = self.simple_directed_graph
(8)                 G.graph["hi"] = "there"
(8)                 G.graph["id"] = "1"
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh, prettyprint=False)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 assert sorted(G.nodes()) == sorted(H.nodes())
(8)                 assert sorted(G.edges()) == sorted(H.edges())
(8)                 assert sorted(G.edges(data=True)) == sorted(H.edges(data=True))
(8)                 self.simple_directed_fh.seek(0)
(4)             def test_write_read_attribute_named_key_ids_graphml(self):
(8)                 from xml.etree.ElementTree import parse
(8)                 G = self.attribute_named_key_ids_graph
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh, named_key_ids=True)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 fh.seek(0)
(8)                 assert nodes_equal(G.nodes(), H.nodes())
(8)                 assert edges_equal(G.edges(), H.edges())
(8)                 assert edges_equal(G.edges(data=True), H.edges(data=True))
(8)                 self.attribute_named_key_ids_fh.seek(0)
(8)                 xml = parse(fh)
(8)                 children = list(xml.getroot())
(8)                 assert len(children) == 4
(8)                 keys = [child.items() for child in children[:3]]
(8)                 assert len(keys) == 3
(8)                 assert ("id", "edge_prop") in keys[0]
(8)                 assert ("attr.name", "edge_prop") in keys[0]
(8)                 assert ("id", "prop2") in keys[1]
(8)                 assert ("attr.name", "prop2") in keys[1]
(8)                 assert ("id", "prop1") in keys[2]
(8)                 assert ("attr.name", "prop1") in keys[2]
(8)                 default_behavior_fh = io.BytesIO()
(8)                 nx.write_graphml(G, default_behavior_fh)
(8)                 default_behavior_fh.seek(0)
(8)                 H = nx.read_graphml(default_behavior_fh)
(8)                 named_key_ids_behavior_fh = io.BytesIO()
(8)                 nx.write_graphml(G, named_key_ids_behavior_fh, named_key_ids=True)
(8)                 named_key_ids_behavior_fh.seek(0)
(8)                 J = nx.read_graphml(named_key_ids_behavior_fh)
(8)                 assert all(n1 == n2 for (n1, n2) in zip(H.nodes, J.nodes))
(8)                 assert all(e1 == e2 for (e1, e2) in zip(H.edges, J.edges))
(4)             def test_write_read_attribute_numeric_type_graphml(self):
(8)                 from xml.etree.ElementTree import parse
(8)                 G = self.attribute_numeric_type_graph
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh, infer_numeric_types=True)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 fh.seek(0)
(8)                 assert nodes_equal(G.nodes(), H.nodes())
```

```
(8)                 assert edges_equal(G.edges(), H.edges())
(8)                 assert edges_equal(G.edges(data=True), H.edges(data=True))
(8)                 self.attribute_numeric_type_fh.seek(0)
(8)                 xml = parse(fh)
(8)                 children = list(xml.getroot())
(8)                 assert len(children) == 3
(8)                 keys = [child.items() for child in children[:2]]
(8)                 assert len(keys) == 2
(8)                 assert ("attr.type", "double") in keys[0]
(8)                 assert ("attr.type", "double") in keys[1]
(4)             def test_more_multigraph_keys(self, tmp_path):
(8)                 """Writing keys as edge id attributes means keys become strings.
(8)                 The original keys are stored as data, so read them back in
(8)                 if `str(key) == edge_id`
(8)                 This allows the adjacency to remain the same.
(8)                 """
(8)                 G = nx.MultiGraph()
(8)                 G.add_edges_from([("a", "b", 2), ("a", "b", 3)])
(8)                 fname = tmp_path / "test.graphml"
(8)                 self.writer(G, fname)
(8)                 H = nx.read_graphml(fname)
(8)                 assert H.is_multigraph()
(8)                 assert edges_equal(G.edges(keys=True), H.edges(keys=True))
(8)                 assert G._adj == H._adj
(4)             def test_default_attribute(self):
(8)                 G = nx.Graph(name="Fred")
(8)                 G.add_node(1, label=1, color="green")
(8)                 nx.add_path(G, [0, 1, 2, 3])
(8)                 G.add_edge(1, 2, weight=3)
(8)                 G.graph["node_default"] = {"color": "yellow"}
(8)                 G.graph["edge_default"] = {"weight": 7}
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh, node_type=int)
(8)                 assert nodes_equal(G.nodes(), H.nodes())
(8)                 assert edges_equal(G.edges(), H.edges())
(8)                 assert G.graph == H.graph
(4)             def test_mixed_type_attributes(self):
(8)                 G = nx.MultiGraph()
(8)                 G.add_node("n0", special=False)
(8)                 G.add_node("n1", special=0)
(8)                 G.add_edge("n0", "n1", special=False)
(8)                 G.add_edge("n0", "n1", special=0)
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 assert not H.nodes["n0"]["special"]
(8)                 assert H.nodes["n1"]["special"] == 0
(8)                 assert not H.edges["n0", "n1", 0]["special"]
(8)                 assert H.edges["n0", "n1", 1]["special"] == 0
(4)             def test_str_number_mixed_type_attributes(self):
(8)                 G = nx.MultiGraph()
(8)                 G.add_node("n0", special="hello")
(8)                 G.add_node("n1", special=0)
(8)                 G.add_edge("n0", "n1", special="hello")
(8)                 G.add_edge("n0", "n1", special=0)
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 assert H.nodes["n0"]["special"] == "hello"
(8)                 assert H.nodes["n1"]["special"] == 0
(8)                 assert H.edges["n0", "n1", 0]["special"] == "hello"
(8)                 assert H.edges["n0", "n1", 1]["special"] == 0
(4)             def test_mixed_int_type_number_attributes(self):
(8)                 np = pytest.importorskip("numpy")
(8)                 G = nx.MultiGraph()
```

```
(8)                 G.add_node("n0", special=np.int64(0))
(8)                 G.add_node("n1", special=1)
(8)                 G.add_edge("n0", "n1", special=np.int64(2))
(8)                 G.add_edge("n0", "n1", special=3)
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 assert H.nodes["n0"]["special"] == 0
(8)                 assert H.nodes["n1"]["special"] == 1
(8)                 assert H.edges["n0", "n1", 0]["special"] == 2
(8)                 assert H.edges["n0", "n1", 1]["special"] == 3
(4)             def test_multigraph_to_graph(self, tmp_path):
(8)                 G = nx.MultiGraph()
(8)                 G.add_edges_from([("a", "b", 2), ("b", "c", 3)])  # no multiedges
(8)                 fname = tmp_path / "test.graphml"
(8)                 self.writer(G, fname)
(8)                 H = nx.read_graphml(fname)
(8)                 assert not H.is_multigraph()
(8)                 H = nx.read_graphml(fname, force_multigraph=True)
(8)                 assert H.is_multigraph()
(8)                 G.add_edge("a", "b", "e-id")
(8)                 fname = tmp_path / "test.graphml"
(8)                 self.writer(G, fname)
(8)                 H = nx.read_graphml(fname)
(8)                 assert H.is_multigraph()
(8)                 H = nx.read_graphml(fname, force_multigraph=True)
(8)                 assert H.is_multigraph()
(4)             def test_write_generate_edge_id_from_attribute(self, tmp_path):
(8)                 from xml.etree.ElementTree import parse
(8)                 G = nx.Graph()
(8)                 G.add_edges_from([("a", "b"), ("b", "c"), ("a", "c")])
(8)                 edge_attributes = {e: str(e) for e in G.edges}
(8)                 nx.set_edge_attributes(G, edge_attributes, "eid")
(8)                 fname = tmp_path / "test.graphml"
(8)                 self.writer(G, fname, edge_id_from_attribute="eid")
(8)                 generator = nx.generate_graphml(G, edge_id_from_attribute="eid")
(8)                 H = nx.read_graphml(fname)
(8)                 assert nodes_equal(G.nodes(), H.nodes())
(8)                 assert edges_equal(G.edges(), H.edges())
(8)                 nx.set_edge_attributes(G, edge_attributes, "id")
(8)                 assert edges_equal(G.edges(data=True), H.edges(data=True))
(8)                 tree = parse(fname)
(8)                 children = list(tree.getroot())
(8)                 assert len(children) == 2
(8)                 edge_ids = [
(12)                    edge.attrib["id"]
(12)                    for edge in tree.getroot().findall(
(16)                        ".//{http://graphml.graphdrawing.org/xmlns}edge"
(12)                    )
(8)                 ]
(8)                 assert sorted(edge_ids) == sorted(edge_attributes.values())
(8)                 data = "".join(generator)
(8)                 J = nx.parse_graphml(data)
(8)                 assert sorted(G.nodes()) == sorted(J.nodes())
(8)                 assert sorted(G.edges()) == sorted(J.edges())
(8)                 nx.set_edge_attributes(G, edge_attributes, "id")
(8)                 assert edges_equal(G.edges(data=True), J.edges(data=True))
(4)             def test_multigraph_write_generate_edge_id_from_attribute(self, tmp_path):
(8)                 from xml.etree.ElementTree import parse
(8)                 G = nx.MultiGraph()
(8)                 G.add_edges_from([("a", "b"), ("b", "c"), ("a", "c"), ("a", "b")])
(8)                 edge_attributes = {e: str(e) for e in G.edges}
(8)                 nx.set_edge_attributes(G, edge_attributes, "eid")
(8)                 fname = tmp_path / "test.graphml"
(8)                 self.writer(G, fname, edge_id_from_attribute="eid")
(8)                 generator = nx.generate_graphml(G, edge_id_from_attribute="eid")
(8)                 H = nx.read_graphml(fname)
(8)                 assert H.is_multigraph()
```

```
1261: (8)                    H = nx.read_graphml(fname, force_multigraph=True)
1262: (8)                    assert H.is_multigraph()
1263: (8)                    assert nodes_equal(G.nodes(), H.nodes())
1264: (8)                    assert edges_equal(G.edges(), H.edges())
1265: (8)                    assert sorted(data.get("eid") for u, v, data in H.edges(data=True)) ==
sorted(
1266: (12)                       edge_attributes.values()
1267: (8)                    )
1268: (8)                    assert sorted(key for u, v, key in H.edges(keys=True)) == sorted(
1269: (12)                       edge_attributes.values()
1270: (8)                    )
1271: (8)                    tree = parse(fname)
1272: (8)                    children = list(tree.getroot())
1273: (8)                    assert len(children) == 2
1274: (8)                    edge_ids = [
1275: (12)                       edge.attrib["id"]
1276: (12)                       for edge in tree.getroot().findall(
1277: (16)                           ".//{http://graphml.graphdrawing.org/xmlns}edge"
1278: (12)                       )
1279: (8)                    ]
1280: (8)                    assert sorted(edge_ids) == sorted(edge_attributes.values())
1281: (8)                    graphml_data = "".join(generator)
1282: (8)                    J = nx.parse_graphml(graphml_data)
1283: (8)                    assert J.is_multigraph()
1284: (8)                    assert nodes_equal(G.nodes(), J.nodes())
1285: (8)                    assert edges_equal(G.edges(), J.edges())
1286: (8)                    assert sorted(data.get("eid") for u, v, data in J.edges(data=True)) ==
sorted(
1287: (12)                       edge_attributes.values()
1288: (8)                    )
1289: (8)                    assert sorted(key for u, v, key in J.edges(keys=True)) == sorted(
1290: (12)                       edge_attributes.values()
1291: (8)                    )
1292: (4)                def test_numpy_float64(self, tmp_path):
1293: (8)                    np = pytest.importorskip("numpy")
1294: (8)                    wt = np.float64(3.4)
1295: (8)                    G = nx.Graph([(1, 2, {"weight": wt})])
1296: (8)                    fname = tmp_path / "test.graphml"
1297: (8)                    self.writer(G, fname)
1298: (8)                    H = nx.read_graphml(fname, node_type=int)
1299: (8)                    assert G.edges == H.edges
1300: (8)                    wtG = G[1][2]["weight"]
1301: (8)                    wtH = H[1][2]["weight"]
1302: (8)                    assert wtG == pytest.approx(wtH, abs=1e-6)
1303: (8)                    assert type(wtG) == np.float64
1304: (8)                    assert type(wtH) == float
1305: (4)                def test_numpy_float32(self, tmp_path):
1306: (8)                    np = pytest.importorskip("numpy")
1307: (8)                    wt = np.float32(3.4)
1308: (8)                    G = nx.Graph([(1, 2, {"weight": wt})])
1309: (8)                    fname = tmp_path / "test.graphml"
1310: (8)                    self.writer(G, fname)
1311: (8)                    H = nx.read_graphml(fname, node_type=int)
1312: (8)                    assert G.edges == H.edges
1313: (8)                    wtG = G[1][2]["weight"]
1314: (8)                    wtH = H[1][2]["weight"]
1315: (8)                    assert wtG == pytest.approx(wtH, abs=1e-6)
1316: (8)                    assert type(wtG) == np.float32
1317: (8)                    assert type(wtH) == float
1318: (4)                def test_numpy_float64_inference(self, tmp_path):
1319: (8)                    np = pytest.importorskip("numpy")
1320: (8)                    G = self.attribute_numeric_type_graph
1321: (8)                    G.edges[("n1", "n1")]["weight"] = np.float64(1.1)
1322: (8)                    fname = tmp_path / "test.graphml"
1323: (8)                    self.writer(G, fname, infer_numeric_types=True)
1324: (8)                    H = nx.read_graphml(fname)
1325: (8)                    assert G._adj == H._adj
1326: (4)                def test_unicode_attributes(self, tmp_path):
1327: (8)                    G = nx.Graph()
```

```
(8)                 name1 = chr(2344) + chr(123) + chr(6543)
(8)                 name2 = chr(5543) + chr(1543) + chr(324)
(8)                 node_type = str
(8)                 G.add_edge(name1, "Radiohead", foo=name2)
(8)                 fname = tmp_path / "test.graphml"
(8)                 self.writer(G, fname)
(8)                 H = nx.read_graphml(fname, node_type=node_type)
(8)                 assert G._adj == H._adj
(4)             def test_unicode_escape(self):
(8)                 import json
(8)                 a = {"a": '{"a": "123"}'}  # an object with many chars to escape
(8)                 sa = json.dumps(a)
(8)                 G = nx.Graph()
(8)                 G.graph["test"] = sa
(8)                 fh = io.BytesIO()
(8)                 self.writer(G, fh)
(8)                 fh.seek(0)
(8)                 H = nx.read_graphml(fh)
(8)                 assert G.graph["test"] == H.graph["test"]
(0)         class TestXMLGraphML(TestWriteGraphML):
(4)             writer = staticmethod(nx.write_graphml_xml)
(4)             @classmethod
(4)             def setup_class(cls):
(8)                 TestWriteGraphML.setup_class()
(0)         def test_exception_for_unsupported_datatype_node_attr():
(4)             """Test that a detailed exception is raised when an attribute is of a type
(4)             not supported by GraphML, e.g. a list"""
(4)             pytest.importorskip("lxml.etree")
(4)             G = nx.Graph()
(4)             G.add_node(0, my_list_attribute=[0, 1, 2])
(4)             fh = io.BytesIO()
(4)             with pytest.raises(TypeError, match="GraphML does not support"):
(8)                 nx.write_graphml(G, fh)
(0)         def test_exception_for_unsupported_datatype_edge_attr():
(4)             """Test that a detailed exception is raised when an attribute is of a type
(4)             not supported by GraphML, e.g. a list"""
(4)             pytest.importorskip("lxml.etree")
(4)             G = nx.Graph()
(4)             G.add_edge(0, 1, my_list_attribute=[0, 1, 2])
(4)             fh = io.BytesIO()
(4)             with pytest.raises(TypeError, match="GraphML does not support"):
(8)                 nx.write_graphml(G, fh)
(0)         def test_exception_for_unsupported_datatype_graph_attr():
(4)             """Test that a detailed exception is raised when an attribute is of a type
(4)             not supported by GraphML, e.g. a list"""
(4)             pytest.importorskip("lxml.etree")
(4)             G = nx.Graph()
(4)             G.graph["my_list_attribute"] = [0, 1, 2]
(4)             fh = io.BytesIO()
(4)             with pytest.raises(TypeError, match="GraphML does not support"):
(8)                 nx.write_graphml(G, fh)
(0)         def test_empty_attribute():
(4)             """Tests that a GraphML string with an empty attribute can be parsed
(4)             correctly."""
(4)             s = """<?xml version='1.0' encoding='utf-8'?>
(4)             <graphml>
(6)               <key id="d1" for="node" attr.name="foo" attr.type="string"/>
(6)               <key id="d2" for="node" attr.name="bar" attr.type="string"/>
(6)               <graph>
(8)                 <node id="0">
(10)                  <data key="d1">aaa</data>
(10)                  <data key="d2">bbb</data>
(8)                 </node>
(8)                 <node id="1">
(10)                  <data key="d1">ccc</data>
(10)                  <data key="d2"></data>
(8)                 </node>
(6)               </graph>
(4)             </graphml>"""
```

```
1397: (4)               fh = io.BytesIO(s.encode("UTF-8"))
1398: (4)               G = nx.read_graphml(fh)
1399: (4)               assert G.nodes["0"] == {"foo": "aaa", "bar": "bbb"}
1400: (4)               assert G.nodes["1"] == {"foo": "ccc", "bar": ""}

----------------------------------------

File 531 - test_p2g.py:

1: (0)              import io
2: (0)              import networkx as nx
3: (0)              from networkx.readwrite.p2g import read_p2g, write_p2g
4: (0)              from networkx.utils import edges_equal
5: (0)              class TestP2G:
6: (4)                  @classmethod
7: (4)                  def setup_class(cls):
8: (8)                      cls.G = nx.Graph(name="test")
9: (8)                      e = [("a", "b"), ("b", "c"), ("c", "d"), ("d", "e"), ("e", "f"), ("a",
"f")]
10: (8)                     cls.G.add_edges_from(e)
11: (8)                     cls.G.add_node("g")
12: (8)                     cls.DG = nx.DiGraph(cls.G)
13: (4)                 def test_read_p2g(self):
14: (8)                     s = b"""\
15: (0)             name
16: (0)             3 4
17: (0)             a
18: (0)             1 2
19: (0)             b
20: (0)             c
21: (0)             0 2
22: (0)             """
23: (8)                     bytesIO = io.BytesIO(s)
24: (8)                     G = read_p2g(bytesIO)
25: (8)                     assert G.name == "name"
26: (8)                     assert sorted(G) == ["a", "b", "c"]
27: (8)                     edges = [(str(u), str(v)) for u, v in G.edges()]
28: (8)                     assert edges_equal(G.edges(), [("a", "c"), ("a", "b"), ("c", "a"),
("c", "c")])
29: (4)                 def test_write_p2g(self):
30: (8)                     s = b"""foo
31: (0)             3 2
32: (0)             1
33: (0)             1
34: (0)             2
35: (0)             2
36: (0)             3
37: (0)             """
38: (8)                     fh = io.BytesIO()
39: (8)                     G = nx.DiGraph()
40: (8)                     G.name = "foo"
41: (8)                     G.add_edges_from([(1, 2), (2, 3)])
42: (8)                     write_p2g(G, fh)
43: (8)                     fh.seek(0)
44: (8)                     r = fh.read()
45: (8)                     assert r == s
46: (4)                 def test_write_read_p2g(self):
47: (8)                     fh = io.BytesIO()
48: (8)                     G = nx.DiGraph()
49: (8)                     G.name = "foo"
50: (8)                     G.add_edges_from([("a", "b"), ("b", "c")])
51: (8)                     write_p2g(G, fh)
52: (8)                     fh.seek(0)
53: (8)                     H = read_p2g(fh)
54: (8)                     assert edges_equal(G.edges(), H.edges())

----------------------------------------

File 532 - test_leda.py:
```

```
1: (0)              import io
2: (0)              import networkx as nx
3: (0)              class TestLEDA:
4: (4)                  def test_parse_leda(self):
5: (8)                      data = """#header section         \nLEDA.GRAPH \nstring\nint\n-
1\n#nodes section\n5 \n|{v1}| \n|{v2}| \n|{v3}| \n|{v4}| \n|{v5}| \n\n#edges section\n7 \n1 2 0 |
{4}| \n1 3 0 |{3}| \n2 3 0 |{2}| \n3 4 0 |{3}| \n3 5 0 |{7}| \n4 5 0 |{6}| \n5 1 0 |{foo}|"""
6: (8)                      G = nx.parse_leda(data)
7: (8)                      G = nx.parse_leda(data.split("\n"))
8: (8)                      assert sorted(G.nodes()) == ["v1", "v2", "v3", "v4", "v5"]
9: (8)                      assert sorted(G.edges(data=True)) == [
10: (12)                        ("v1", "v2", {"label": "4"}),
11: (12)                        ("v1", "v3", {"label": "3"}),
12: (12)                        ("v2", "v3", {"label": "2"}),
13: (12)                        ("v3", "v4", {"label": "3"}),
14: (12)                        ("v3", "v5", {"label": "7"}),
15: (12)                        ("v4", "v5", {"label": "6"}),
16: (12)                        ("v5", "v1", {"label": "foo"}),
17: (8)                     ]
18: (4)                 def test_read_LEDA(self):
19: (8)                     fh = io.BytesIO()
20: (8)                     data = """#header section         \nLEDA.GRAPH \nstring\nint\n-
1\n#nodes section\n5 \n|{v1}| \n|{v2}| \n|{v3}| \n|{v4}| \n|{v5}| \n\n#edges section\n7 \n1 2 0 |
{4}| \n1 3 0 |{3}| \n2 3 0 |{2}| \n3 4 0 |{3}| \n3 5 0 |{7}| \n4 5 0 |{6}| \n5 1 0 |{foo}|"""
21: (8)                     G = nx.parse_leda(data)
22: (8)                     fh.write(data.encode("UTF-8"))
23: (8)                     fh.seek(0)
24: (8)                     Gin = nx.read_leda(fh)
25: (8)                     assert sorted(G.nodes()) == sorted(Gin.nodes())
26: (8)                     assert sorted(G.edges()) == sorted(Gin.edges())
```

File 533 - test_pajek.py:

```
1: (0)              """
2: (0)              Pajek tests
3: (0)              """
4: (0)              import networkx as nx
5: (0)              from networkx.utils import edges_equal, nodes_equal
6: (0)              class TestPajek:
7: (4)                  @classmethod
8: (4)                  def setup_class(cls):
9: (8)                      cls.data = """*network Tralala\n*vertices 4\n   1 "A1"         0.0938
0.0896   ellipse x_fact 1 y_fact 1\n   2 "Bb"         0.8188 0.2458   ellipse x_fact 1 y_fact 1\n
3 "C"          0.3688 0.7792   ellipse x_fact 1\n   4 "D2"         0.9583 0.8563   ellipse x_fact
1\n*arcs\n1 1 1  h2 0 w 3 c Blue s 3 a1 -130 k1 0.6 a2 -130 k2 0.6 ap 0.5 l "Bezier loop" lc
BlueViolet fos 20 lr 58 lp 0.3 la 360\n2 1 1  h2 0 a1 120 k1 1.3 a2 -120 k2 0.3 ap 25 l "Bezier
arc" lphi 270 la 180 lr 19 lp 0.5\n1 2 1  h2 0 a1 40 k1 2.8 a2 30 k2 0.8 ap 25 l "Bezier arc" lphi
90 la 0 lp 0.65\n4 2 -1  h2 0 w 1 k1 -2 k2 250 ap 25 l "Circular arc" c Red lc OrangeRed\n3 4 1  p
Dashed h2 0 w 2 c OliveGreen ap 25 l "Straight arc" lc PineGreen\n1 3 1  p Dashed h2 0 w 5 k1 -1
k2 -20 ap 25 l "Oval arc" c Brown lc Black\n3 3 -1  h1 6 w 1 h2 12 k1 -2 k2 -15 ap 0.5 l "Circular
loop" c Red lc OrangeRed lphi 270 la 180"""
10: (8)                     cls.G = nx.MultiDiGraph()
11: (8)                     cls.G.add_nodes_from(["A1", "Bb", "C", "D2"])
12: (8)                     cls.G.add_edges_from(
13: (12)                        [
14: (16)                            ("A1", "A1"),
15: (16)                            ("A1", "Bb"),
16: (16)                            ("A1", "C"),
17: (16)                            ("Bb", "A1"),
18: (16)                            ("C", "C"),
19: (16)                            ("C", "D2"),
20: (16)                            ("D2", "Bb"),
21: (12)                        ]
22: (8)                     )
23: (8)                     cls.G.graph["name"] = "Tralala"
24: (4)                 def test_parse_pajek_simple(self):
```

```
 25: (8)                    data = """*Vertices 2\n1 "1"\n2 "2"\n*Edges\n1 2\n2 1"""
 26: (8)                    G = nx.parse_pajek(data)
 27: (8)                    assert sorted(G.nodes()) == ["1", "2"]
 28: (8)                    assert edges_equal(G.edges(), [("1", "2"), ("1", "2")])
 29: (4)                def test_parse_pajek(self):
 30: (8)                    G = nx.parse_pajek(self.data)
 31: (8)                    assert sorted(G.nodes()) == ["A1", "Bb", "C", "D2"]
 32: (8)                    assert edges_equal(
 33: (12)                       G.edges(),
 34: (12)                       [
 35: (16)                           ("A1", "A1"),
 36: (16)                           ("A1", "Bb"),
 37: (16)                           ("A1", "C"),
 38: (16)                           ("Bb", "A1"),
 39: (16)                           ("C", "C"),
 40: (16)                           ("C", "D2"),
 41: (16)                           ("D2", "Bb"),
 42: (12)                       ],
 43: (8)                    )
 44: (4)                def test_parse_pajet_mat(self):
 45: (8)                    data = """*Vertices 3\n1 "one"\n2 "two"\n3 "three"\n*Matrix\n1 1 0\n0
1 0\n0 1 0\n"""
 46: (8)                    G = nx.parse_pajek(data)
 47: (8)                    assert set(G.nodes()) == {"one", "two", "three"}
 48: (8)                    assert G.nodes["two"] == {"id": "2"}
 49: (8)                    assert edges_equal(
 50: (12)                       set(G.edges()),
 51: (12)                       {("one", "one"), ("two", "one"), ("two", "two"), ("two",
"three")},
 52: (8)                    )
 53: (4)                def test_read_pajek(self, tmp_path):
 54: (8)                    G = nx.parse_pajek(self.data)
 55: (8)                    fname = tmp_path / "test.pjk"
 56: (8)                    with open(fname, "wb") as fh:
 57: (12)                       fh.write(self.data.encode("UTF-8"))
 58: (8)                    Gin = nx.read_pajek(fname)
 59: (8)                    assert sorted(G.nodes()) == sorted(Gin.nodes())
 60: (8)                    assert edges_equal(G.edges(), Gin.edges())
 61: (8)                    assert self.G.graph == Gin.graph
 62: (8)                    for n in G:
 63: (12)                       assert G.nodes[n] == Gin.nodes[n]
 64: (4)                def test_write_pajek(self):
 65: (8)                    import io
 66: (8)                    G = nx.parse_pajek(self.data)
 67: (8)                    fh = io.BytesIO()
 68: (8)                    nx.write_pajek(G, fh)
 69: (8)                    fh.seek(0)
 70: (8)                    H = nx.read_pajek(fh)
 71: (8)                    assert nodes_equal(list(G), list(H))
 72: (8)                    assert edges_equal(list(G.edges()), list(H.edges()))
 73: (4)                def test_ignored_attribute(self):
 74: (8)                    import io
 75: (8)                    G = nx.Graph()
 76: (8)                    fh = io.BytesIO()
 77: (8)                    G.add_node(1, int_attr=1)
 78: (8)                    G.add_node(2, empty_attr="  ")
 79: (8)                    G.add_edge(1, 2, int_attr=2)
 80: (8)                    G.add_edge(2, 3, empty_attr="  ")
 81: (8)                    import warnings
 82: (8)                    with warnings.catch_warnings(record=True) as w:
 83: (12)                       nx.write_pajek(G, fh)
 84: (12)                       assert len(w) == 4
 85: (4)                def test_noname(self):
 86: (8)                    line = "*network\n"
 87: (8)                    other_lines = self.data.split("\n")[1:]
 88: (8)                    data = line + "\n".join(other_lines)
 89: (8)                    G = nx.parse_pajek(data)
 90: (4)                def test_unicode(self):
 91: (8)                    import io
```

```
92: (8)                     G = nx.Graph()
93: (8)                     name1 = chr(2344) + chr(123) + chr(6543)
94: (8)                     name2 = chr(5543) + chr(1543) + chr(324)
95: (8)                     G.add_edge(name1, "Radiohead", foo=name2)
96: (8)                     fh = io.BytesIO()
97: (8)                     nx.write_pajek(G, fh)
98: (8)                     fh.seek(0)
99: (8)                     H = nx.read_pajek(fh)
100: (8)                    assert nodes_equal(list(G), list(H))
101: (8)                    assert edges_equal(list(G.edges()), list(H.edges()))
102: (8)                    assert G.graph == H.graph

----------------------------------------

File 534 - test_sparse6.py:

1: (0)              from io import BytesIO
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              from networkx.utils import edges_equal, nodes_equal
5: (0)              class TestSparseGraph6:
6: (4)                  def test_from_sparse6_bytes(self):
7: (8)                      data = b":Q___eDcdFcDeFcE`GaJ`IaHbKNbLM"
8: (8)                      G = nx.from_sparse6_bytes(data)
9: (8)                      assert nodes_equal(
10: (12)                        sorted(G.nodes()),
11: (12)                        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17],
12: (8)                     )
13: (8)                     assert edges_equal(
14: (12)                        G.edges(),
15: (12)                        [
16: (16)                            (0, 1),
17: (16)                            (0, 2),
18: (16)                            (0, 3),
19: (16)                            (1, 12),
20: (16)                            (1, 14),
21: (16)                            (2, 13),
22: (16)                            (2, 15),
23: (16)                            (3, 16),
24: (16)                            (3, 17),
25: (16)                            (4, 7),
26: (16)                            (4, 9),
27: (16)                            (4, 11),
28: (16)                            (5, 6),
29: (16)                            (5, 8),
30: (16)                            (5, 9),
31: (16)                            (6, 10),
32: (16)                            (6, 11),
33: (16)                            (7, 8),
34: (16)                            (7, 10),
35: (16)                            (8, 12),
36: (16)                            (9, 15),
37: (16)                            (10, 14),
38: (16)                            (11, 13),
39: (16)                            (12, 16),
40: (16)                            (13, 17),
41: (16)                            (14, 17),
42: (16)                            (15, 16),
43: (12)                        ],
44: (8)                     )
45: (4)                 def test_from_bytes_multigraph_graph(self):
46: (8)                     graph_data = b":An"
47: (8)                     G = nx.from_sparse6_bytes(graph_data)
48: (8)                     assert type(G) == nx.Graph
49: (8)                     multigraph_data = b":Ab"
50: (8)                     M = nx.from_sparse6_bytes(multigraph_data)
51: (8)                     assert type(M) == nx.MultiGraph
52: (4)                 def test_read_sparse6(self):
53: (8)                     data = b":Q___eDcdFcDeFcE`GaJ`IaHbKNbLM"
```

```
54: (8)                     G = nx.from_sparse6_bytes(data)
55: (8)                     fh = BytesIO(data)
56: (8)                     Gin = nx.read_sparse6(fh)
57: (8)                     assert nodes_equal(G.nodes(), Gin.nodes())
58: (8)                     assert edges_equal(G.edges(), Gin.edges())
59: (4)                 def test_read_many_graph6(self):
60: (8)                     data = b":Q___eDcdFcDeFcE`GaJ`IaHbKNbLM\n"
b":Q___dCfDEdcEgcbEGbFIaJ`JaHN`IM"
61: (8)                     fh = BytesIO(data)
62: (8)                     glist = nx.read_sparse6(fh)
63: (8)                     assert len(glist) == 2
64: (8)                     for G in glist:
65: (12)                        assert nodes_equal(
66: (16)                            G.nodes(),
67: (16)                            [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17],
68: (12)                        )
69: (0)             class TestWriteSparse6:
70: (4)                 """Unit tests for writing graphs in the sparse6 format.
71: (4)                 Most of the test cases were checked against the sparse6 encoder in Sage.
72: (4)                 """
73: (4)                 def test_null_graph(self):
74: (8)                     G = nx.null_graph()
75: (8)                     result = BytesIO()
76: (8)                     nx.write_sparse6(G, result)
77: (8)                     assert result.getvalue() == b">>sparse6<<:?\n"
78: (4)                 def test_trivial_graph(self):
79: (8)                     G = nx.trivial_graph()
80: (8)                     result = BytesIO()
81: (8)                     nx.write_sparse6(G, result)
82: (8)                     assert result.getvalue() == b">>sparse6<<:@\n"
83: (4)                 def test_empty_graph(self):
84: (8)                     G = nx.empty_graph(5)
85: (8)                     result = BytesIO()
86: (8)                     nx.write_sparse6(G, result)
87: (8)                     assert result.getvalue() == b">>sparse6<<:D\n"
88: (4)                 def test_large_empty_graph(self):
89: (8)                     G = nx.empty_graph(68)
90: (8)                     result = BytesIO()
91: (8)                     nx.write_sparse6(G, result)
92: (8)                     assert result.getvalue() == b">>sparse6<<:~?@C\n"
93: (4)                 def test_very_large_empty_graph(self):
94: (8)                     G = nx.empty_graph(258049)
95: (8)                     result = BytesIO()
96: (8)                     nx.write_sparse6(G, result)
97: (8)                     assert result.getvalue() == b">>sparse6<<:~~???~?@\n"
98: (4)                 def test_complete_graph(self):
99: (8)                     G = nx.complete_graph(4)
100: (8)                    result = BytesIO()
101: (8)                    nx.write_sparse6(G, result)
102: (8)                    assert result.getvalue() == b">>sparse6<<:CcKI\n"
103: (4)                def test_no_header(self):
104: (8)                    G = nx.complete_graph(4)
105: (8)                    result = BytesIO()
106: (8)                    nx.write_sparse6(G, result, header=False)
107: (8)                    assert result.getvalue() == b":CcKI\n"
108: (4)                def test_padding(self):
109: (8)                    codes = (b":Cdv", b":DaYn", b":EaYnN", b":FaYnL", b":GaYnLz")
110: (8)                    for n, code in enumerate(codes, start=4):
111: (12)                       G = nx.path_graph(n)
112: (12)                       result = BytesIO()
113: (12)                       nx.write_sparse6(G, result, header=False)
114: (12)                       assert result.getvalue() == code + b"\n"
115: (4)                def test_complete_bipartite(self):
116: (8)                    G = nx.complete_bipartite_graph(6, 9)
117: (8)                    result = BytesIO()
118: (8)                    nx.write_sparse6(G, result)
119: (8)                    expected = b">>sparse6<<:Nk" + b"?G`cJ" * 9 + b"\n"
120: (8)                    assert result.getvalue() == expected
```

```
121: (4)                def test_read_write_inverse(self):
122: (8)                    for i in list(range(13)) + [31, 47, 62, 63, 64, 72]:
123: (12)                       m = min(2 * i, i * i // 2)
124: (12)                       g = nx.random_graphs.gnm_random_graph(i, m, seed=i)
125: (12)                       gstr = BytesIO()
126: (12)                       nx.write_sparse6(g, gstr, header=False)
127: (12)                       gstr = gstr.getvalue().rstrip()
128: (12)                       g2 = nx.from_sparse6_bytes(gstr)
129: (12)                       assert g2.order() == g.order()
130: (12)                       assert edges_equal(g2.edges(), g.edges())
131: (4)                def test_no_directed_graphs(self):
132: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
133: (12)                       nx.write_sparse6(nx.DiGraph(), BytesIO())
134: (4)                def test_write_path(self, tmp_path):
135: (8)                    fullfilename = str(tmp_path / "test.s6")
136: (8)                    nx.write_sparse6(nx.null_graph(), fullfilename)
137: (8)                    with open(fullfilename, mode="rb") as fh:
138: (12)                       assert fh.read() == b">>sparse6<<:?\n"

----------------------------------------

File 535 - __init__.py:

1: (0)

----------------------------------------

File 536 - test_text.py:

1: (0)              import random
2: (0)              from itertools import product
3: (0)              from textwrap import dedent
4: (0)              import pytest
5: (0)              import networkx as nx
6: (0)              def test_forest_str_directed():
7: (4)                  graph = nx.balanced_tree(r=2, h=2, create_using=nx.DiGraph)
8: (4)                  for node in graph.nodes:
9: (8)                      graph.nodes[node]["label"] = "node_" + chr(ord("a") + node)
10: (4)                 node_target = dedent(
11: (8)                     """
12: (8)                     ╙── 0
13: (12)                        ├─╼ 1
14: (12)                        │   ├─╼ 3
15: (12)                        │   └─╼ 4
16: (12)                        └─╼ 2
17: (16)                            ├─╼ 5
18: (16)                            └─╼ 6
19: (8)                     """
20: (4)                 ).strip()
21: (4)                 label_target = dedent(
22: (8)                     """
23: (8)                     ╙── node_a
24: (12)                        ├─╼ node_b
25: (12)                        │   ├─╼ node_d
26: (12)                        │   └─╼ node_e
27: (12)                        └─╼ node_c
28: (16)                            ├─╼ node_f
29: (16)                            └─╼ node_g
30: (8)                     """
31: (4)                 ).strip()
32: (4)                 ret = nx.forest_str(graph, with_labels=False)
33: (4)                 print(ret)
34: (4)                 assert ret == node_target
35: (4)                 ret = nx.forest_str(graph, with_labels=True)
36: (4)                 print(ret)
37: (4)                 assert ret == label_target
38: (4)                 lines = []
39: (4)                 ret = nx.forest_str(graph, write=lines.append, with_labels=False)
40: (4)                 assert ret is None
```

```
(4)                 assert lines == node_target.split("\n")
(4)                 ret = nx.forest_str(graph, write=print)
(4)                 assert ret is None
(0)             def test_write_network_text_empty_graph():
(4)                 def _graph_str(g, **kw):
(8)                     printbuf = []
(8)                     nx.write_network_text(g, printbuf.append, end="", **kw)
(8)                     return "\n".join(printbuf)
(4)                 assert _graph_str(nx.DiGraph()) == "╙"
(4)                 assert _graph_str(nx.Graph()) == "╙"
(4)                 assert _graph_str(nx.DiGraph(), ascii_only=True) == "+"
(4)                 assert _graph_str(nx.Graph(), ascii_only=True) == "+"
(0)             def test_write_network_text_within_forest_glyph():
(4)                 g = nx.DiGraph()
(4)                 g.add_nodes_from([1, 2, 3, 4])
(4)                 g.add_edge(2, 4)
(4)                 lines = []
(4)                 write = lines.append
(4)                 nx.write_network_text(g, path=write, end="")
(4)                 nx.write_network_text(g, path=write, ascii_only=True, end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     ╟── 1
(8)                     ╟── 2
(8)                     ╎   ╙── 4
(8)                     ╙── 3
(8)                     +-- 1
(8)                     +-- 2
(8)                     :   L-> 4
(8)                     +-- 3
(8)                     """
(4)                 ).strip()
(4)                 assert text == target
(0)             def test_forest_str_directed_multi_tree():
(4)                 tree1 = nx.balanced_tree(r=2, h=2, create_using=nx.DiGraph)
(4)                 tree2 = nx.balanced_tree(r=2, h=2, create_using=nx.DiGraph)
(4)                 forest = nx.disjoint_union_all([tree1, tree2])
(4)                 ret = nx.forest_str(forest)
(4)                 print(ret)
(4)                 target = dedent(
(8)                     """
(8)                     ╟── 0
(8)                     ╎   ├── 1
(8)                     ╎   ╎   ├── 3
(8)                     ╎   ╎   ╙── 4
(8)                     ╎   ╙── 2
(8)                     ╎       ├── 5
(8)                     ╎       ╙── 6
(8)                     ╙── 7
(12)                        ├── 8
(12)                        ╎   ├── 10
(12)                        ╎   ╙── 11
(12)                        ╙── 9
(16)                            ├── 12
(16)                            ╙── 13
(8)                     """
(4)                 ).strip()
(4)                 assert ret == target
(4)                 tree3 = nx.balanced_tree(r=2, h=2, create_using=nx.DiGraph)
(4)                 forest = nx.disjoint_union_all([tree1, tree2, tree3])
(4)                 ret = nx.forest_str(forest, sources=[0, 14, 7])
(4)                 print(ret)
(4)                 target = dedent(
(8)                     """
(8)                     ╟── 0
(8)                     ╎   ├── 1
(8)                     ╎   ╎   ├── 3
```

```
(8)                     │   │   └─╼ 4
(8)                     │   └─╼ 2
(8)                     │       ├─╼ 5
(8)                     │       └─╼ 6
(8)                     ╠═══ 14
(8)                     │   ├─╼ 15
(8)                     │   │   ├─╼ 17
(8)                     │   │   └─╼ 18
(8)                     │   └─╼ 16
(8)                     │       ├─╼ 19
(8)                     │       └─╼ 20
(8)                     ╚═══ 7
(12)                        ├─╼ 8
(12)                        │   ├─╼ 10
(12)                        │   └─╼ 11
(12)                        └─╼ 9
(16)                            ├─╼ 12
(16)                            └─╼ 13
(8)                     """
(4)                 ).strip()
(4)                 assert ret == target
(4)                 ret = nx.forest_str(forest, sources=[0, 14, 7], ascii_only=True)
(4)                 print(ret)
(4)                 target = dedent(
(8)                     """
(8)                     +-- 0
(8)                     :   |-> 1
(8)                     :   |   |-> 3
(8)                     :   |   L-> 4
(8)                     :   L-> 2
(8)                     :       |-> 5
(8)                     :       L-> 6
(8)                     +-- 14
(8)                     :   |-> 15
(8)                     :   |   |-> 17
(8)                     :   |   L-> 18
(8)                     :   L-> 16
(8)                     :       |-> 19
(8)                     :       L-> 20
(8)                     +-- 7
(12)                        |-> 8
(12)                        |   |-> 10
(12)                        |   L-> 11
(12)                        L-> 9
(16)                            |-> 12
(16)                            L-> 13
(8)                     """
(4)                 ).strip()
(4)                 assert ret == target
(0)             def test_forest_str_undirected_multi_tree():
(4)                 tree1 = nx.balanced_tree(r=2, h=2, create_using=nx.Graph)
(4)                 tree2 = nx.balanced_tree(r=2, h=2, create_using=nx.Graph)
(4)                 tree2 = nx.relabel_nodes(tree2, {n: n + len(tree1) for n in tree2.nodes})
(4)                 forest = nx.union(tree1, tree2)
(4)                 ret = nx.forest_str(forest, sources=[0, 7])
(4)                 print(ret)
(4)                 target = dedent(
(8)                     """
(8)                     ╠═══ 0
(8)                     │   ├── 1
(8)                     │   │   ├── 3
(8)                     │   │   └── 4
(8)                     │   └── 2
(8)                     │       ├── 5
(8)                     │       └── 6
(8)                     ╚═══ 7
(12)                        ├── 8
(12)                        │   ├── 10
(12)                        │   └── 11
```

```
(12)                        └── 9
(16)                            ├── 12
(16)                            └── 13
(8)                     """
(4)                 ).strip()
(4)                 assert ret == target
(4)                 ret = nx.forest_str(forest, sources=[0, 7], ascii_only=True)
(4)                 print(ret)
(4)                 target = dedent(
(8)                     """
(8)                     +-- 0
(8)                     :   |-- 1
(8)                     :   |   |-- 3
(8)                     :   |   L-- 4
(8)                     :   L-- 2
(8)                     :       |-- 5
(8)                     :       L-- 6
(8)                     +-- 7
(12)                        |-- 8
(12)                        |   |-- 10
(12)                        |   L-- 11
(12)                        L-- 9
(16)                            |-- 12
(16)                            L-- 13
(8)                     """
(4)                 ).strip()
(4)                 assert ret == target
(0)             def test_forest_str_undirected():
(4)                 graph = nx.balanced_tree(r=2, h=2, create_using=nx.Graph)
(4)                 nx.forest_str(graph)
(4)                 node_target0 = dedent(
(8)                     """
(8)                     ╙── 0
(12)                        ├── 1
(12)                        │   ├── 3
(12)                        │   └── 4
(12)                        └── 2
(16)                            ├── 5
(16)                            └── 6
(8)                     """
(4)                 ).strip()
(4)                 ret = nx.forest_str(graph, sources=[0])
(4)                 print(ret)
(4)                 assert ret == node_target0
(4)                 node_target2 = dedent(
(8)                     """
(8)                     ╙── 2
(12)                        ├── 0
(12)                        │   └── 1
(12)                        │       ├── 3
(12)                        │       └── 4
(12)                        ├── 5
(12)                        └── 6
(8)                     """
(4)                 ).strip()
(4)                 ret = nx.forest_str(graph, sources=[2])
(4)                 print(ret)
(4)                 assert ret == node_target2
(0)             def test_forest_str_errors():
(4)                 ugraph = nx.complete_graph(3, create_using=nx.Graph)
(4)                 with pytest.raises(nx.NetworkXNotImplemented):
(8)                     nx.forest_str(ugraph)
(4)                 dgraph = nx.complete_graph(3, create_using=nx.DiGraph)
(4)                 with pytest.raises(nx.NetworkXNotImplemented):
(8)                     nx.forest_str(dgraph)
(0)             def test_forest_str_overspecified_sources():
(4)                 """
(4)                 When sources are directly specified, we won't be able to determine when we
(4)                 are in the last component, so there will always be a trailing, leftmost
```

```
(4)                 pipe.
(4)                 """
(4)                 graph = nx.disjoint_union_all(
(8)                     [
(12)                        nx.balanced_tree(r=2, h=1, create_using=nx.DiGraph),
(12)                        nx.balanced_tree(r=1, h=2, create_using=nx.DiGraph),
(12)                        nx.balanced_tree(r=2, h=1, create_using=nx.DiGraph),
(8)                     ]
(4)                 )
(4)                 target1 = dedent(
(8)                     """
(8)                     ╟─╼ 0
(8)                     ╎   ├─╼ 1
(8)                     ╎   └─╼ 2
(8)                     ╟─╼ 3
(8)                     ╎   └─╼ 4
(8)                     ╎       └─╼ 5
(8)                     ╟─╼ 6
(8)                     ╎   ├─╼ 7
(8)                     ╎   └─╼ 8
(8)                     """
(4)                 ).strip()
(4)                 target2 = dedent(
(8)                     """
(8)                     ╟─╼ 0
(8)                     ╎   ├─╼ 1
(8)                     ╎   └─╼ 2
(8)                     ╟─╼ 3
(8)                     ╎   └─╼ 4
(8)                     ╎       └─╼ 5
(8)                     ╙─╼ 6
(12)                        ├─╼ 7
(12)                        └─╼ 8
(8)                     """
(4)                 ).strip()
(4)                 lines = []
(4)                 nx.forest_str(graph, write=lines.append, sources=graph.nodes)
(4)                 got1 = "\n".join(lines)
(4)                 print("got1: ")
(4)                 print(got1)
(4)                 lines = []
(4)                 nx.forest_str(graph, write=lines.append)
(4)                 got2 = "\n".join(lines)
(4)                 print("got2: ")
(4)                 print(got2)
(4)                 assert got1 == target1
(4)                 assert got2 == target2
(0)             def test_write_network_text_iterative_add_directed_edges():
(4)                 """
(4)                 Walk through the cases going from a disconnected to fully connected graph
(4)                 """
(4)                 graph = nx.DiGraph()
(4)                 graph.add_nodes_from([1, 2, 3, 4])
(4)                 lines = []
(4)                 write = lines.append
(4)                 write("--- initial state ---")
(4)                 nx.write_network_text(graph, path=write, end="")
(4)                 for i, j in product(graph.nodes, graph.nodes):
(8)                     write(f"--- add_edge({i}, {j}) ---")
(8)                     graph.add_edge(i, j)
(8)                     nx.write_network_text(graph, path=write, end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     --- initial state ---
(8)                     ╟─╼ 1
(8)                     ╟─╼ 2
(8)                     ╟─╼ 3
```

```
(8)                     ╙── 4
(8)                     --- add_edge(1, 1) ---
(8)                     ╟── 1 - 1
(8)                     ╎   └─ ...
(8)                     ╟── 2
(8)                     ╟── 3
(8)                     ╙── 4
(8)                     --- add_edge(1, 2) ---
(8)                     ╟── 1 - 1
(8)                     ╎   ├─ 2
(8)                     ╎   └─ ...
(8)                     ╟── 3
(8)                     ╙── 4
(8)                     --- add_edge(1, 3) ---
(8)                     ╟── 1 - 1
(8)                     ╎   ├─ 2
(8)                     ╎   ├─ 3
(8)                     ╎   └─ ...
(8)                     ╙── 4
(8)                     --- add_edge(1, 4) ---
(8)                     ╙── 1 - 1
(12)                        ├─ 2
(12)                        ├─ 3
(12)                        ├─ 4
(12)                        └─ ...
(8)                     --- add_edge(2, 1) ---
(8)                     ╙── 2 - 1
(12)                        └─ 1 - 1
(16)                           ├─ 3
(16)                           ├─ 4
(16)                           └─ ...
(8)                     --- add_edge(2, 2) ---
(8)                     ╙── 1 - 1, 2
(12)                        ├─ 2 - 2
(12)                        │  └─ ...
(12)                        ├─ 3
(12)                        ├─ 4
(12)                        └─ ...
(8)                     --- add_edge(2, 3) ---
(8)                     ╙── 1 - 1, 2
(12)                        ├─ 2 - 2
(12)                        │  ├─ 3 - 1
(12)                        │  └─ ...
(12)                        ├─ 4
(12)                        └─ ...
(8)                     --- add_edge(2, 4) ---
(8)                     ╙── 1 - 1, 2
(12)                        ├─ 2 - 2
(12)                        │  ├─ 3 - 1
(12)                        │  ├─ 4 - 1
(12)                        │  └─ ...
(12)                        └─ ...
(8)                     --- add_edge(3, 1) ---
(8)                     ╙── 2 - 1, 2
(12)                        ├─ 1 - 1, 3
(12)                        │  ├─ 3 - 2
(12)                        │  │  └─ ...
(12)                        │  ├─ 4 - 2
(12)                        │  └─ ...
(12)                        └─ ...
(8)                     --- add_edge(3, 2) ---
(8)                     ╙── 3 - 1, 2
(12)                        ├─ 1 - 1, 2
(12)                        │  ├─ 2 - 2, 3
(12)                        │  │  ├─ 4 - 1
(12)                        │  │  └─ ...
(12)                        │  └─ ...
(12)                        └─ ...
(8)                     --- add_edge(3, 3) ---
```

```
(8)                     ╙── 1 ╾ 1, 2, 3
(12)                        ├─╼ 2 ╾ 2, 3
(12)                        │   ├─╼ 3 ╾ 1, 3
(12)                        │   │   └─╼  ...
(12)                        │   ├─╼ 4 ╾ 1
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     --- add_edge(3, 4) ---
(8)                     ╙── 1 ╾ 1, 2, 3
(12)                        ├─╼ 2 ╾ 2, 3
(12)                        │   ├─╼ 3 ╾ 1, 3
(12)                        │   │   ├─╼ 4 ╾ 1, 2
(12)                        │   │   └─╼  ...
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     --- add_edge(4, 1) ---
(8)                     ╙── 2 ╾ 1, 2, 3
(12)                        ├─╼ 1 ╾ 1, 3, 4
(12)                        │   ├─╼ 3 ╾ 2, 3
(12)                        │   │   ├─╼ 4 ╾ 1, 2
(12)                        │   │   │   └─╼  ...
(12)                        │   │   └─╼  ...
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     --- add_edge(4, 2) ---
(8)                     ╙── 3 ╾ 1, 2, 3
(12)                        ├─╼ 1 ╾ 1, 2, 4
(12)                        │   ├─╼ 2 ╾ 2, 3, 4
(12)                        │   │   ├─╼ 4 ╾ 1, 3
(12)                        │   │   │   └─╼  ...
(12)                        │   │   └─╼  ...
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     --- add_edge(4, 3) ---
(8)                     ╙── 4 ╾ 1, 2, 3
(12)                        ├─╼ 1 ╾ 1, 2, 3
(12)                        │   ├─╼ 2 ╾ 2, 3, 4
(12)                        │   │   ├─╼ 3 ╾ 1, 3, 4
(12)                        │   │   │   └─╼  ...
(12)                        │   │   └─╼  ...
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     --- add_edge(4, 4) ---
(8)                     ╙── 1 ╾ 1, 2, 3, 4
(12)                        ├─╼ 2 ╾ 2, 3, 4
(12)                        │   ├─╼ 3 ╾ 1, 3, 4
(12)                        │   │   ├─╼ 4 ╾ 1, 2, 4
(12)                        │   │   │   └─╼  ...
(12)                        │   │   └─╼  ...
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_iterative_add_undirected_edges():
(4)                 """
(4)                 Walk through the cases going from a disconnected to fully connected graph
(4)                 """
(4)                 graph = nx.Graph()
(4)                 graph.add_nodes_from([1, 2, 3, 4])
(4)                 lines = []
(4)                 write = lines.append
(4)                 write("--- initial state ---")
(4)                 nx.write_network_text(graph, path=write, end="")
(4)                 for i, j in product(graph.nodes, graph.nodes):
(8)                     if i == j:
(12)                        continue
(8)                     write(f"--- add_edge({i}, {j}) ---")
(8)                     graph.add_edge(i, j)
```

```
        nx.write_network_text(graph, path=write, end="")
    text = "\n".join(lines)
    print(text)
    target = dedent(
        """
        --- initial state ---
        ╟── 1
        ╟── 2
        ╟── 3
        ╙── 4
        --- add_edge(1, 2) ---
        ╟── 3
        ╟── 4
        ╙── 1
            ╙── 2
        --- add_edge(1, 3) ---
        ╟── 4
        ╙── 2
            ╙── 1
                ╙── 3
        --- add_edge(1, 4) ---
        ╙── 2
            ╙── 1
                ├── 3
                ╙── 4
        --- add_edge(2, 1) ---
        ╙── 2
            ╙── 1
                ├── 3
                ╙── 4
        --- add_edge(2, 3) ---
        ╙── 4
            ╙── 1
                ├── 2
                │   ╙── 3 — 1
                ╙──  ...
        --- add_edge(2, 4) ---
        ╙── 3
            ├── 1
            │   ├── 2 — 3
            │   ╙── 4 — 1
            ╙──  ...
            ╙──  ...
        --- add_edge(3, 1) ---
        ╙── 3
            ├── 1
            │   ├── 2 — 3
            │   ╙── 4 — 1
            ╙──  ...
            ╙──  ...
        --- add_edge(3, 2) ---
        ╙── 3
            ├── 1
            │   ├── 2 — 3
            │   ╙── 4 — 1
            ╙──  ...
            ╙──  ...
        --- add_edge(3, 4) ---
        ╙── 1
            ├── 2
            │   ├── 3 — 1
            │   ╙── 4 — 1, 2
            ╙──  ...
            ╙──  ...
        --- add_edge(4, 1) ---
        ╙── 1
            ├── 2
            │   ├── 3 — 1
            │   ╙── 4 — 1, 2
```

```
                    │   └──  ...
                    └──  ...
                --- add_edge(4, 2) ---
                ╙── 1
                    ├── 2
                    │   ├── 3 — 1
                    │   │   └── 4 — 1, 2
                    │   └──  ...
                    └──  ...
                --- add_edge(4, 3) ---
                ╙── 1
                    ├── 2
                    │   ├── 3 — 1
                    │   │   └── 4 — 1, 2
                    │   └──  ...
                    └──  ...
                """
            ).strip()
            assert target == text
        def test_write_network_text_iterative_add_random_directed_edges():
            """
            Walk through the cases going from a disconnected to fully connected graph
            """
            rng = random.Random(724466096)
            graph = nx.DiGraph()
            graph.add_nodes_from([1, 2, 3, 4, 5])
            possible_edges = list(product(graph.nodes, graph.nodes))
            rng.shuffle(possible_edges)
            graph.add_edges_from(possible_edges[0:8])
            lines = []
            write = lines.append
            write("--- initial state ---")
            nx.write_network_text(graph, path=write, end="")
            for i, j in possible_edges[8:12]:
                write(f"--- add_edge({i}, {j}) ---")
                graph.add_edge(i, j)
                nx.write_network_text(graph, path=write, end="")
            text = "\n".join(lines)
            print(text)
            target = dedent(
                """
                --- initial state ---
                ╙── 3 ╾ 5
                    └─╼ 2 ╾ 2
                        ├─╼ 4 ╾ 4
                        │   ├─╼ 5
                        │   │   ├─╼ 1 ╾ 1
                        │   │   └─╼  ...
                        │   └─╼  ...
                        └─╼  ...
                --- add_edge(4, 1) ---
                ╙── 3 ╾ 5
                    └─╼ 2 ╾ 2
                        ├─╼ 4 ╾ 4
                        │   ├─╼ 5
                        │   │   ├─╼ 1 ╾ 1, 4
                        │   │   └─╼  ...
                        │   └─╼  ...
                        └─╼  ...
                --- add_edge(2, 1) ---
                ╙── 3 ╾ 5
                    └─╼ 2 ╾ 2
                        ├─╼ 4 ╾ 4
                        │   ├─╼ 5
                        │   │   ├─╼ 1 ╾ 1, 4, 2
                        │   │   └─╼  ...
                        │   └─╼  ...
```

```
(16)                        │   └─╸ ...
(16)                        └─╸ ...
(8)                 --- add_edge(5, 2) ---
(8)                 ╙── 3 ╾ 5
(12)                    └─╸ 2 ╾ 2, 5
(16)                        ├─╸ 4 ╾ 4
(16)                        │   ├─╸ 5
(16)                        │   │   ├─╸ 1 ╾ 1, 4, 2
(16)                        │   │   └─╸ ...
(16)                        │   └─╸ ...
(16)                        └─╸ ...
(16)                        └─╸ ...
(8)                 --- add_edge(1, 5) ---
(8)                 ╙── 3 ╾ 5
(12)                    └─╸ 2 ╾ 2, 5
(16)                        ├─╸ 4 ╾ 4
(16)                        │   ├─╸ 5 ╾ 1
(16)                        │   │   ├─╸ 1 ╾ 1, 4, 2
(16)                        │   │   └─╸ ...
(16)                        │   └─╸ ...
(16)                        └─╸ ...
(16)                        └─╸ ...
(8)                 """
(4)             ).strip()
(4)             assert target == text
(0)         def test_write_network_text_nearly_forest():
(4)             g = nx.DiGraph()
(4)             g.add_edge(1, 2)
(4)             g.add_edge(1, 5)
(4)             g.add_edge(2, 3)
(4)             g.add_edge(3, 4)
(4)             g.add_edge(5, 6)
(4)             g.add_edge(6, 7)
(4)             g.add_edge(6, 8)
(4)             orig = g.copy()
(4)             g.add_edge(1, 8)  # forward edge
(4)             g.add_edge(4, 2)  # back edge
(4)             g.add_edge(6, 3)  # cross edge
(4)             lines = []
(4)             write = lines.append
(4)             write("--- directed case ---")
(4)             nx.write_network_text(orig, path=write, end="")
(4)             write("--- add (1, 8), (4, 2), (6, 3) ---")
(4)             nx.write_network_text(g, path=write, end="")
(4)             write("--- undirected case ---")
(4)             nx.write_network_text(orig.to_undirected(), path=write, sources=[1],
end="")
(4)             write("--- add (1, 8), (4, 2), (6, 3) ---")
(4)             nx.write_network_text(g.to_undirected(), path=write, sources=[1], end="")
(4)             text = "\n".join(lines)
(4)             print(text)
(4)             target = dedent(
(8)                 """
(8)                 --- directed case ---
(8)                 ╙── 1
(12)                    ├─╸ 2
(12)                    │   └─╸ 3
(12)                    │       └─╸ 4
(12)                    └─╸ 5
(16)                        └─╸ 6
(20)                            ├─╸ 7
(20)                            └─╸ 8
(8)                 --- add (1, 8), (4, 2), (6, 3) ---
(8)                 ╙── 1
(12)                    ├─╸ 2 ╾ 4
(12)                    │   └─╸ 3 ╾ 6
(12)                    │       └─╸ 4
(12)                    │           └─╸ ...
(12)                    ├─╸ 5
```

```
(12)                        │   └─╸ 6
(12)                        │       ├─╸ 7
(12)                        │       ├─╸ 8 ━ 1
(12)                        │       └─╸  ...
(12)                        └─╸  ...
(8)                     --- undirected case ---
(8)                     ╙── 1
(12)                        ├── 2
(12)                        │   └── 3
(12)                        │       └── 4
(12)                        └── 5
(16)                            └── 6
(20)                                ├── 7
(20)                                └── 8
(8)                     --- add (1, 8), (4, 2), (6, 3) ---
(8)                     ╙── 1
(12)                        ├── 2
(12)                        │   ├── 3
(12)                        │   │   ├── 4 — 2
(12)                        │   │   └── 6
(12)                        │   │       ├── 5 — 1
(12)                        │   │       ├── 7
(12)                        │   │       └── 8 — 1
(12)                        │   └──  ...
(12)                        └──  ...
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_complete_graph_ascii_only():
(4)                 graph = nx.generators.complete_graph(5, create_using=nx.DiGraph)
(4)                 lines = []
(4)                 write = lines.append
(4)                 write("--- directed case ---")
(4)                 nx.write_network_text(graph, path=write, ascii_only=True, end="")
(4)                 write("--- undirected case ---")
(4)                 nx.write_network_text(graph.to_undirected(), path=write, ascii_only=True,
end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     --- directed case ---
(8)                     +-- 0 <- 1, 2, 3, 4
(12)                        |-> 1 <- 2, 3, 4
(12)                        |   |-> 2 <- 0, 3, 4
(12)                        |   |   |-> 3 <- 0, 1, 4
(12)                        |   |   |   |-> 4 <- 0, 1, 2
(12)                        |   |   |   |   L->  ...
(12)                        |   |   |   L->  ...
(12)                        |   |   L->  ...
(12)                        |   L->  ...
(12)                        L->  ...
(8)                     --- undirected case ---
(8)                     +-- 0
(12)                        |-- 1
(12)                        |   |-- 2 - 0
(12)                        |   |   |-- 3 - 0, 1
(12)                        |   |   |   L-- 4 - 0, 1, 2
(12)                        |   |   L--  ...
(12)                        |   L--  ...
(12)                        L--  ...
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_with_labels():
(4)                 graph = nx.generators.complete_graph(5, create_using=nx.DiGraph)
(4)                 for n in graph.nodes:
(8)                     graph.nodes[n]["label"] = f"Node(n={n})"
(4)                 lines = []
```

```
(4)                 write = lines.append
(4)                 nx.write_network_text(graph, path=write, with_labels=True,
ascii_only=False, end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     ╙── Node(n=0) ╾ Node(n=1), Node(n=2), Node(n=3), Node(n=4)
(12)                        ├─╼ Node(n=1) ╾ Node(n=2), Node(n=3), Node(n=4)
(12)                        │   ├─╼ Node(n=2) ╾ Node(n=0), Node(n=3), Node(n=4)
(12)                        │   │   ├─╼ Node(n=3) ╾ Node(n=0), Node(n=1), Node(n=4)
(12)                        │   │   │   ├─╼ Node(n=4) ╾ Node(n=0), Node(n=1), Node(n=2)
(12)                        │   │   │   │   └─╼  ...
(12)                        │   │   │   └─╼  ...
(12)                        │   │   └─╼  ...
(12)                        │   └─╼  ...
(12)                        └─╼  ...
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_complete_graphs():
(4)                 lines = []
(4)                 write = lines.append
(4)                 for k in [0, 1, 2, 3, 4, 5]:
(8)                     g = nx.generators.complete_graph(k)
(8)                     write(f"--- undirected k={k} ---")
(8)                     nx.write_network_text(g, path=write, end="")
(4)                 for k in [0, 1, 2, 3, 4, 5]:
(8)                     g = nx.generators.complete_graph(k, nx.DiGraph)
(8)                     write(f"--- directed k={k} ---")
(8)                     nx.write_network_text(g, path=write, end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     --- undirected k=0 ---
(8)                     ╙
(8)                     --- undirected k=1 ---
(8)                     ╙── 0
(8)                     --- undirected k=2 ---
(8)                     ╙── 0
(12)                        └── 1
(8)                     --- undirected k=3 ---
(8)                     ╙── 0
(12)                        ├── 1
(12)                        │   └── 2 — 0
(12)                        └──  ...
(8)                     --- undirected k=4 ---
(8)                     ╙── 0
(12)                        ├── 1
(12)                        │   ├── 2 — 0
(12)                        │   │   └── 3 — 0, 1
(12)                        │   └──  ...
(12)                        └──  ...
(8)                     --- undirected k=5 ---
(8)                     ╙── 0
(12)                        ├── 1
(12)                        │   ├── 2 — 0
(12)                        │   │   ├── 3 — 0, 1
(12)                        │   │   │   └── 4 — 0, 1, 2
(12)                        │   │   └──  ...
(12)                        │   └──  ...
(12)                        └──  ...
(8)                     --- directed k=0 ---
(8)                     ╙
(8)                     --- directed k=1 ---
(8)                     ╙── 0
(8)                     --- directed k=2 ---
(8)                     ╙── 0 ╾ 1
```

```
(12)                        └─╸ 1
(16)                            └─╸  ...
(8)                     --- directed k=3 ---
(8)                     ╙── 0 ╾ 1, 2
(12)                        ├─╸ 1 ╾ 2
(12)                        │   ├─╸ 2 ╾ 0
(12)                        │   │   └─╸  ...
(12)                        │   └─╸  ...
(12)                        └─╸  ...
(8)                     --- directed k=4 ---
(8)                     ╙── 0 ╾ 1, 2, 3
(12)                        ├─╸ 1 ╾ 2, 3
(12)                        │   ├─╸ 2 ╾ 0, 3
(12)                        │   │   ├─╸ 3 ╾ 0, 1
(12)                        │   │   │   └─╸  ...
(12)                        │   │   └─╸  ...
(12)                        │   └─╸  ...
(12)                        └─╸  ...
(8)                     --- directed k=5 ---
(8)                     ╙── 0 ╾ 1, 2, 3, 4
(12)                        ├─╸ 1 ╾ 2, 3, 4
(12)                        │   ├─╸ 2 ╾ 0, 3, 4
(12)                        │   │   ├─╸ 3 ╾ 0, 1, 4
(12)                        │   │   │   ├─╸ 4 ╾ 0, 1, 2
(12)                        │   │   │   │   └─╸  ...
(12)                        │   │   │   └─╸  ...
(12)                        │   │   └─╸  ...
(12)                        │   └─╸  ...
(12)                        └─╸  ...
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_multiple_sources():
(4)                 g = nx.DiGraph()
(4)                 g.add_edge(1, 2)
(4)                 g.add_edge(1, 3)
(4)                 g.add_edge(2, 4)
(4)                 g.add_edge(3, 5)
(4)                 g.add_edge(3, 6)
(4)                 g.add_edge(5, 4)
(4)                 g.add_edge(4, 1)
(4)                 g.add_edge(1, 5)
(4)                 lines = []
(4)                 write = lines.append
(4)                 nodes = sorted(g.nodes())
(4)                 for n in nodes:
(8)                     write(f"--- source node: {n} ---")
(8)                     nx.write_network_text(g, path=write, sources=[n], end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     --- source node: 1 ---
(8)                     ╙── 1 ╾ 4
(12)                        ├─╸ 2
(12)                        │   └─╸ 4 ╾ 5
(12)                        │       └─╸  ...
(12)                        ├─╸ 3
(12)                        │   ├─╸ 5 ╾ 1
(12)                        │   └─╸  ...
(12)                        │   └─╸ 6
(12)                        └─╸  ...
(8)                     --- source node: 2 ---
(8)                     ╙── 2 ╾ 1
(12)                        └─╸ 4 ╾ 5
(16)                            └─╸ 1
(20)                                ├─╸ 3
(20)                                │   ├─╸ 5 ╾ 1
(20)                                │   │   └─╸  ...
```

```
(20)                                │   └─ 6
(20)                                └─  ...
(8)                     --- source node: 3 ---
(8)                     ╙─ 3 ─ 1
(12)                        ├─ 5 ─ 1
(12)                        │   └─ 4 ─ 2
(12)                        │       └─ 1
(12)                        │           ├─ 2
(12)                        │           │   └─  ...
(12)                        │           └─  ...
(12)                        └─ 6
(8)                     --- source node: 4 ---
(8)                     ╙─ 4 ─ 2, 5
(12)                        └─ 1
(16)                            ├─ 2
(16)                            │   └─  ...
(16)                            ├─ 3
(16)                            │   ├─ 5 ─ 1
(16)                            │   │   └─  ...
(16)                            │   └─ 6
(16)                            └─  ...
(8)                     --- source node: 5 ---
(8)                     ╙─ 5 ─ 3, 1
(12)                        └─ 4 ─ 2
(16)                            └─ 1
(20)                                ├─ 2
(20)                                │   └─  ...
(20)                                ├─ 3
(20)                                │   ├─ 6
(20)                                │   └─  ...
(20)                                └─  ...
(8)                     --- source node: 6 ---
(8)                     ╙─ 6 ─ 3
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_star_graph():
(4)                 graph = nx.star_graph(5, create_using=nx.Graph)
(4)                 lines = []
(4)                 write = lines.append
(4)                 nx.write_network_text(graph, path=write, end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     ╙─ 1
(12)                        └─ 0
(16)                            ├─ 2
(16)                            ├─ 3
(16)                            ├─ 4
(16)                            └─ 5
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_path_graph():
(4)                 graph = nx.path_graph(3, create_using=nx.Graph)
(4)                 lines = []
(4)                 write = lines.append
(4)                 nx.write_network_text(graph, path=write, end="")
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     ╙─ 0
(12)                        └─ 1
(16)                            └─ 2
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
```

```
(0)  def test_write_network_text_lollipop_graph():
(4)      graph = nx.lollipop_graph(4, 2, create_using=nx.Graph)
(4)      lines = []
(4)      write = lines.append
(4)      nx.write_network_text(graph, path=write, end="")
(4)      text = "\n".join(lines)
(4)      print(text)
(4)      target = dedent(
(8)          """
(8)          ╙── 5
(12)             └── 4
(16)                 └── 3
(20)                     ├── 0
(20)                     │   ├── 1 — 3
(20)                     │   │   └── 2 — 0, 3
(20)                     │   └── ...
(20)                     └── ...
(8)          """
(4)      ).strip()
(4)      assert target == text
(0)  def test_write_network_text_wheel_graph():
(4)      graph = nx.wheel_graph(7, create_using=nx.Graph)
(4)      lines = []
(4)      write = lines.append
(4)      nx.write_network_text(graph, path=write, end="")
(4)      text = "\n".join(lines)
(4)      print(text)
(4)      target = dedent(
(8)          """
(8)          ╙── 1
(12)             ├── 0
(12)             │   ├── 2 — 1
(12)             │   │   └── 3 — 0
(12)             │   │       └── 4 — 0
(12)             │   │           └── 5 — 0
(12)             │   │               └── 6 — 0, 1
(12)             │   └── ...
(12)             └── ...
(8)          """
(4)      ).strip()
(4)      assert target == text
(0)  def test_write_network_text_circular_ladder_graph():
(4)      graph = nx.circular_ladder_graph(4, create_using=nx.Graph)
(4)      lines = []
(4)      write = lines.append
(4)      nx.write_network_text(graph, path=write, end="")
(4)      text = "\n".join(lines)
(4)      print(text)
(4)      target = dedent(
(8)          """
(8)          ╙── 0
(12)             ├── 1
(12)             │   ├── 2
(12)             │   │   ├── 3 — 0
(12)             │   │   └── 7
(12)             │   │       ├── 6 — 2
(12)             │   │       │   └── 5 — 1
(12)             │   │       │       └── 4 — 0, 7
(12)             │   │       └── ...
(12)             │   └── ...
(12)             └── ...
(8)          """
(4)      ).strip()
(4)      assert target == text
(0)  def test_write_network_text_dorogovtsev_goltsev_mendes_graph():
(4)      graph = nx.dorogovtsev_goltsev_mendes_graph(4, create_using=nx.Graph)
(4)      lines = []
(4)      write = lines.append
```

```
(4)             nx.write_network_text(graph, path=write, end="")
(4)             text = "\n".join(lines)
(4)             print(text)
(4)             target = dedent(
(8)                 """
(8)                 ╙── 15
(12)                    ├── 0
(12)                    │   ├── 1 - 15
(12)                    │   │   ├── 2 - 0
(12)                    │   │   │   ├── 4 - 0
(12)                    │   │   │   │   ├── 9 - 0
(12)                    │   │   │   │   │   ├── 22 - 0
(12)                    │   │   │   │   │   └── 38 - 4
(12)                    │   │   │   │   ├── 13 - 2
(12)                    │   │   │   │   │   ├── 34 - 2
(12)                    │   │   │   │   │   └── 39 - 4
(12)                    │   │   │   │   ├── 18 - 0
(12)                    │   │   │   │   ├── 30 - 2
(12)                    │   │   │   │   └── ...
(12)                    │   │   │   ├── 5 - 1
(12)                    │   │   │   │   ├── 12 - 1
(12)                    │   │   │   │   │   ├── 29 - 1
(12)                    │   │   │   │   │   └── 40 - 5
(12)                    │   │   │   │   ├── 14 - 2
(12)                    │   │   │   │   │   ├── 35 - 2
(12)                    │   │   │   │   │   └── 41 - 5
(12)                    │   │   │   │   ├── 25 - 1
(12)                    │   │   │   │   ├── 31 - 2
(12)                    │   │   │   │   └── ...
(12)                    │   │   │   ├── 7 - 0
(12)                    │   │   │   │   ├── 20 - 0
(12)                    │   │   │   │   └── 32 - 2
(12)                    │   │   │   ├── 10 - 1
(12)                    │   │   │   │   ├── 27 - 1
(12)                    │   │   │   │   └── 33 - 2
(12)                    │   │   │   ├── 16 - 0
(12)                    │   │   │   ├── 23 - 1
(12)                    │   │   │   └── ...
(12)                    │   │   ├── 3 - 0
(12)                    │   │   │   ├── 8 - 0
(12)                    │   │   │   │   ├── 21 - 0
(12)                    │   │   │   │   └── 36 - 3
(12)                    │   │   │   ├── 11 - 1
(12)                    │   │   │   │   ├── 28 - 1
(12)                    │   │   │   │   └── 37 - 3
(12)                    │   │   │   ├── 17 - 0
(12)                    │   │   │   ├── 24 - 1
(12)                    │   │   │   └── ...
(12)                    │   │   ├── 6 - 0
(12)                    │   │   │   ├── 19 - 0
(12)                    │   │   │   └── 26 - 1
(12)                    │   │   └── ...
(12)                    │   └── ...
(12)                    └── ...
(8)                 """
(4)             ).strip()
(4)             assert target == text
(0)         def test_write_network_text_tree_max_depth():
(4)             orig = nx.balanced_tree(r=1, h=3, create_using=nx.DiGraph)
(4)             lines = []
(4)             write = lines.append
(4)             write("--- directed case, max_depth=0 ---")
(4)             nx.write_network_text(orig, path=write, end="", max_depth=0)
(4)             write("--- directed case, max_depth=1 ---")
(4)             nx.write_network_text(orig, path=write, end="", max_depth=1)
(4)             write("--- directed case, max_depth=2 ---")
(4)             nx.write_network_text(orig, path=write, end="", max_depth=2)
(4)             write("--- directed case, max_depth=3 ---")
(4)             nx.write_network_text(orig, path=write, end="", max_depth=3)
```

```
(4)             write("--- directed case, max_depth=4 ---")
(4)             nx.write_network_text(orig, path=write, end="", max_depth=4)
(4)             write("--- undirected case, max_depth=0 ---")
(4)             nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=0)
(4)             write("--- undirected case, max_depth=1 ---")
(4)             nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=1)
(4)             write("--- undirected case, max_depth=2 ---")
(4)             nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=2)
(4)             write("--- undirected case, max_depth=3 ---")
(4)             nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=3)
(4)             write("--- undirected case, max_depth=4 ---")
(4)             nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=4)
(4)             text = "\n".join(lines)
(4)             print(text)
(4)             target = dedent(
(8)                 """
(8)                 --- directed case, max_depth=0 ---
(8)                 ╙ ...
(8)                 --- directed case, max_depth=1 ---
(8)                 ╙── 0
(12)                    └─ ...
(8)                 --- directed case, max_depth=2 ---
(8)                 ╙── 0
(12)                    └─ 1
(16)                       └─ ...
(8)                 --- directed case, max_depth=3 ---
(8)                 ╙── 0
(12)                    └─ 1
(16)                       └─ 2
(20)                          └─ ...
(8)                 --- directed case, max_depth=4 ---
(8)                 ╙── 0
(12)                    └─ 1
(16)                       └─ 2
(20)                          └─ 3
(8)                 --- undirected case, max_depth=0 ---
(8)                 ╙ ...
(8)                 --- undirected case, max_depth=1 ---
(8)                 ╙── 0 — 1
(12)                    └── ...
(8)                 --- undirected case, max_depth=2 ---
(8)                 ╙── 0
(12)                    └── 1 — 2
(16)                       └── ...
(8)                 --- undirected case, max_depth=3 ---
(8)                 ╙── 0
(12)                    └── 1
(16)                       └── 2 — 3
(20)                          └── ...
(8)                 --- undirected case, max_depth=4 ---
(8)                 ╙── 0
(12)                    └── 1
(16)                       └── 2
(20)                          └── 3
(8)                 """
(4)             ).strip()
(4)             assert target == text
(0)         def test_write_network_text_graph_max_depth():
(4)             orig = nx.erdos_renyi_graph(10, 0.15, directed=True, seed=40392)
(4)             lines = []
(4)             write = lines.append
(4)             write("--- directed case, max_depth=None ---")
(4)             nx.write_network_text(orig, path=write, end="", max_depth=None)
(4)             write("--- directed case, max_depth=0 ---")
```

```
(4)                nx.write_network_text(orig, path=write, end="", max_depth=0)
(4)                write("--- directed case, max_depth=1 ---")
(4)                nx.write_network_text(orig, path=write, end="", max_depth=1)
(4)                write("--- directed case, max_depth=2 ---")
(4)                nx.write_network_text(orig, path=write, end="", max_depth=2)
(4)                write("--- directed case, max_depth=3 ---")
(4)                nx.write_network_text(orig, path=write, end="", max_depth=3)
(4)                write("--- undirected case, max_depth=None ---")
(4)                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=None)
(4)                write("--- undirected case, max_depth=0 ---")
(4)                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=0)
(4)                write("--- undirected case, max_depth=1 ---")
(4)                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=1)
(4)                write("--- undirected case, max_depth=2 ---")
(4)                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=2)
(4)                write("--- undirected case, max_depth=3 ---")
(4)                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=3)
(4)                text = "\n".join(lines)
(4)                print(text)
(4)                target = dedent(
(8)                    """
(8)                    --- directed case, max_depth=None ---
(8)                    ╟── 4
(8)                    ╎   ├─╼ 0 ━ 3
(8)                    ╎   ├─╼ 5 ━ 7
(8)                    ╎   ╎   └─╼ 3
(8)                    ╎   ╎       ├─╼ 1 ━ 9
(8)                    ╎   ╎       ╎   └─╼ 9 ━ 6
(8)                    ╎   ╎       ╎       ├─╼ 6
(8)                    ╎   ╎       ╎       ╎   └─╼  ...
(8)                    ╎   ╎       ╎       ├─╼ 7 ━ 4
(8)                    ╎   ╎       ╎       ╎   ├─╼ 2
(8)                    ╎   ╎       ╎       ╎   └─╼  ...
(8)                    ╎   ╎       ╎       └─╼  ...
(8)                    ╎   ╎       └─╼  ...
(8)                    ╎   └─╼  ...
(8)                    ╙── 8
(8)                    --- directed case, max_depth=0 ---
(8)                    ╙ ...
(8)                    --- directed case, max_depth=1 ---
(8)                    ╟── 4
(8)                    ╎   └─╼  ...
(8)                    ╙── 8
(8)                    --- directed case, max_depth=2 ---
(8)                    ╟── 4
(8)                    ╎   ├─╼ 0 ━ 3
(8)                    ╎   ├─╼ 5 ━ 7
(8)                    ╎   ╎   └─╼  ...
(8)                    ╎   └─╼ 7 ━ 9
(8)                    ╎       └─╼  ...
(8)                    ╙── 8
(8)                    --- directed case, max_depth=3 ---
(8)                    ╟── 4
(8)                    ╎   ├─╼ 0 ━ 3
(8)                    ╎   ├─╼ 5 ━ 7
(8)                    ╎   ╎   └─╼ 3
(8)                    ╎   ╎       └─╼  ...
(8)                    ╎   └─╼ 7 ━ 9
(8)                    ╎       ├─╼ 2
(8)                    ╎       └─╼  ...
(8)                    ╙── 8
(8)                    --- undirected case, max_depth=None ---
(8)                    ╟── 8
(8)                    ╙── 2
```

```
                        └── 7
                            ├── 4
                            │   ├── 0
                            │   │   └── 3
                            │   │       ├── 1
                            │   │       │   └── 9 — 7
                            │   │       │       └── 6
                            │   │       └── 5 — 4, 7
                            │   └──  ...
                            └──  ...
                    --- undirected case, max_depth=0 ---
                    ╙ ...
                    --- undirected case, max_depth=1 ---
                    ╟── 8
                    ╙── 2 — 7
                        └──  ...
                    --- undirected case, max_depth=2 ---
                    ╟── 8
                    ╙── 2
                        └── 7 — 4, 5, 9
                            └──  ...
                    --- undirected case, max_depth=3 ---
                    ╟── 8
                    ╙── 2
                        └── 7
                            ├── 4 — 0, 5
                            │   └──  ...
                            ├── 5 — 4, 3
                            │   └──  ...
                            └── 9 — 1, 6
                                └──  ...
                    """
                ).strip()
                assert target == text
            def test_write_network_text_clique_max_depth():
                orig = nx.complete_graph(5, nx.DiGraph)
                lines = []
                write = lines.append
                write("--- directed case, max_depth=None ---")
                nx.write_network_text(orig, path=write, end="", max_depth=None)
                write("--- directed case, max_depth=0 ---")
                nx.write_network_text(orig, path=write, end="", max_depth=0)
                write("--- directed case, max_depth=1 ---")
                nx.write_network_text(orig, path=write, end="", max_depth=1)
                write("--- directed case, max_depth=2 ---")
                nx.write_network_text(orig, path=write, end="", max_depth=2)
                write("--- directed case, max_depth=3 ---")
                nx.write_network_text(orig, path=write, end="", max_depth=3)
                write("--- undirected case, max_depth=None ---")
                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=None)
                write("--- undirected case, max_depth=0 ---")
                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=0)
                write("--- undirected case, max_depth=1 ---")
                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=1)
                write("--- undirected case, max_depth=2 ---")
                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=2)
                write("--- undirected case, max_depth=3 ---")
                nx.write_network_text(orig.to_undirected(), path=write, end="",
max_depth=3)
                text = "\n".join(lines)
                print(text)
                target = dedent(
                    """
                    --- directed case, max_depth=None ---
                    ╙── 0 ━ 1, 2, 3, 4
```

```
                    ├─╸ 1 ━ 2, 3, 4
                    │   ├─╸ 2 ━ 0, 3, 4
                    │   │   ├─╸ 3 ━ 0, 1, 4
                    │   │   │   ├─╸ 4 ━ 0, 1, 2
                    │   │   │   │   └─╸ ...
                    │   │   │   └─╸ ...
                    │   │   └─╸ ...
                    │   └─╸ ...
                    └─╸ ...
                --- directed case, max_depth=0 ---
                ╙ ...
                --- directed case, max_depth=1 ---
                ╙─╸ 0 ━ 1, 2, 3, 4
                    └─╸ ...
                --- directed case, max_depth=2 ---
                ╙─╸ 0 ━ 1, 2, 3, 4
                    ├─╸ 1 ━ 2, 3, 4
                    │   └─╸ ...
                    ├─╸ 2 ━ 1, 3, 4
                    │   └─╸ ...
                    ├─╸ 3 ━ 1, 2, 4
                    │   └─╸ ...
                    └─╸ 4 ━ 1, 2, 3
                        └─╸ ...
                --- directed case, max_depth=3 ---
                ╙─╸ 0 ━ 1, 2, 3, 4
                    ├─╸ 1 ━ 2, 3, 4
                    │   ├─╸ 2 ━ 0, 3, 4
                    │   │   └─╸ ...
                    │   ├─╸ 3 ━ 0, 2, 4
                    │   │   └─╸ ...
                    │   ├─╸ 4 ━ 0, 2, 3
                    │   │   └─╸ ...
                    │   └─╸ ...
                    └─╸ ...
                --- undirected case, max_depth=None ---
                ╙─╸ 0
                    ├─╸ 1
                    │   ├─╸ 2 ━ 0
                    │   │   ├─╸ 3 ━ 0, 1
                    │   │   │   └─╸ 4 ━ 0, 1, 2
                    │   │   └─╸ ...
                    │   └─╸ ...
                    └─╸ ...
                --- undirected case, max_depth=0 ---
                ╙ ...
                --- undirected case, max_depth=1 ---
                ╙─╸ 0 ━ 1, 2, 3, 4
                    └─╸ ...
                --- undirected case, max_depth=2 ---
                ╙─╸ 0
                    ├─╸ 1 ━ 2, 3, 4
                    │   └─╸ ...
                    ├─╸ 2 ━ 1, 3, 4
                    │   └─╸ ...
                    ├─╸ 3 ━ 1, 2, 4
                    │   └─╸ ...
                    └─╸ 4 ━ 1, 2, 3
                --- undirected case, max_depth=3 ---
                ╙─╸ 0
                    ├─╸ 1
                    │   ├─╸ 2 ━ 0, 3, 4
                    │   │   └─╸ ...
                    │   ├─╸ 3 ━ 0, 2, 4
                    │   │   └─╸ ...
                    │   ├─╸ 4 ━ 0, 2, 3
                    │   └─╸ ...
                    └─╸ ...
                """
            ).strip()
```

```
(4)                 assert target == text
(0)             def test_write_network_text_custom_label():
(4)                 graph = nx.erdos_renyi_graph(5, 0.4, directed=True, seed=359222358)
(4)                 for node in graph.nodes:
(8)                     graph.nodes[node]["label"] = f"Node({node})"
(8)                     graph.nodes[node]["chr"] = chr(node + ord("a") - 1)
(8)                     if node % 2 == 0:
(12)                        graph.nodes[node]["part"] = chr(node + ord("a"))
(4)                 lines = []
(4)                 write = lines.append
(4)                 write("--- when with_labels=True, uses the 'label' attr ---")
(4)                 nx.write_network_text(graph, path=write, with_labels=True, end="",
max_depth=None)
(4)                 write("--- when with_labels=False, uses str(node) value ---")
(4)                 nx.write_network_text(graph, path=write, with_labels=False, end="",
max_depth=None)
(4)                 write("--- when with_labels is a string, use that attr ---")
(4)                 nx.write_network_text(graph, path=write, with_labels="chr", end="",
max_depth=None)
(4)                 write("--- fallback to str(node) when the attr does not exist ---")
(4)                 nx.write_network_text(graph, path=write, with_labels="part", end="",
max_depth=None)
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     --- when with_labels=True, uses the 'label' attr ---
(8)                     ╙── Node(1)
(12)                        └─╼ Node(3) ━ Node(2)
(16)                            ├─╼ Node(0)
(16)                            │   ├─╼ Node(2) ━ Node(3), Node(4)
(16)                            │   └─   ...
(16)                            │   └─╼ Node(4)
(16)                            │       └─   ...
(16)                            └─   ...
(8)                     --- when with_labels=False, uses str(node) value ---
(8)                     ╙── 1
(12)                        └─╼ 3 ━ 2
(16)                            ├─╼ 0
(16)                            │   ├─╼ 2 ━ 3, 4
(16)                            │   └─   ...
(16)                            │   └─╼ 4
(16)                            │       └─   ...
(16)                            └─   ...
(8)                     --- when with_labels is a string, use that attr ---
(8)                     ╙── a
(12)                        └─╼ c ━ b
(16)                            ├─╼ `
(16)                            │   ├─╼ b ━ c, d
(16)                            │   └─   ...
(16)                            │   └─╼ d
(16)                            │       └─   ...
(16)                            └─   ...
(8)                     --- fallback to str(node) when the attr does not exist ---
(8)                     ╙── 1
(12)                        └─╼ 3 ━ c
(16)                            ├─╼ a
(16)                            │   ├─╼ c ━ 3, e
(16)                            │   └─   ...
(16)                            │   └─╼ e
(16)                            │       └─   ...
(16)                            └─   ...
(8)                     """
(4)                 ).strip()
(4)                 assert target == text
(0)             def test_write_network_text_vertical_chains():
(4)                 graph1 = nx.lollipop_graph(4, 2, create_using=nx.Graph)
(4)                 graph1.add_edge(0, -1)
(4)                 graph1.add_edge(-1, -2)
```

```
(4)                 graph1.add_edge(-2, -3)
(4)                 graph2 = graph1.to_directed()
(4)                 graph2.remove_edges_from([(u, v) for u, v in graph2.edges if v > u])
(4)                 lines = []
(4)                 write = lines.append
(4)                 write("--- Undirected UTF ---")
(4)                 nx.write_network_text(graph1, path=write, end="", vertical_chains=True)
(4)                 write("--- Undirected ASCI ---")
(4)                 nx.write_network_text(
(8)                     graph1, path=write, end="", vertical_chains=True, ascii_only=True
(4)                 )
(4)                 write("--- Directed UTF ---")
(4)                 nx.write_network_text(graph2, path=write, end="", vertical_chains=True)
(4)                 write("--- Directed ASCI ---")
(4)                 nx.write_network_text(
(8)                     graph2, path=write, end="", vertical_chains=True, ascii_only=True
(4)                 )
(4)                 text = "\n".join(lines)
(4)                 print(text)
(4)                 target = dedent(
(8)                     """
(8)                     --- Undirected UTF ---
(8)                     ╙── 5
(12)                        │
(12)                        4
(12)                        │
(12)                        3
(12)                        ├── 0
(12)                        │   ├── 1 — 3
(12)                        │   │   │
(12)                        │   │   2 — 0, 3
(12)                        │   ├── -1
(12)                        │   │   │
(12)                        │   │   -2
(12)                        │   │   │
(12)                        │   │   -3
(12)                        │   └──  ...
(12)                        └──  ...
(8)                     --- Undirected ASCI ---
(8)                     +-- 5
(12)                        |
(12)                        4
(12)                        |
(12)                        3
(12)                        |-- 0
(12)                        |   |-- 1 - 3
(12)                        |   |   |
(12)                        |   |   2 - 0, 3
(12)                        |   |-- -1
(12)                        |   |   |
(12)                        |   |   -2
(12)                        |   |   |
(12)                        |   |   -3
(12)                        |   L--  ...
(12)                        L--  ...
(8)                     --- Directed UTF ---
(8)                     ╙── 5
(12)                        ╽
(12)                        4
(12)                        ╽
(12)                        3
(12)                        ├─► 0 ─► 1, 2
(12)                        │   ╽
(12)                        │   -1
(12)                        │   ╽
(12)                        │   -2
(12)                        │   ╽
(12)                        │   -3
(12)                        ├─► 1 ─► 2
```

```
                        │   └─╼  ...
                        └─╼ 2
                            └─╼  ...
                    --- Directed ASCI ---
                    +-- 5
                        !
                        4
                        !
                        3
                        |-> 0 <- 1, 2
                        |   !
                        |   -1
                        |   !
                        |   -2
                        |   !
                        |   -3
                        |-> 1 <- 2
                        |   L->  ...
                        L-> 2
                            L->  ...
                    """
                ).strip()
                assert target == text
            def test_collapse_directed():
                graph = nx.balanced_tree(r=2, h=3, create_using=nx.DiGraph)
                lines = []
                write = lines.append
                write("--- Original ---")
                nx.write_network_text(graph, path=write, end="")
                graph.nodes[1]["collapse"] = True
                write("--- Collapse Node 1 ---")
                nx.write_network_text(graph, path=write, end="")
                write("--- Add alternate path (5, 3) to collapsed zone")
                graph.add_edge(5, 3)
                nx.write_network_text(graph, path=write, end="")
                write("--- Collapse Node 0 ---")
                graph.nodes[0]["collapse"] = True
                nx.write_network_text(graph, path=write, end="")
                text = "\n".join(lines)
                print(text)
                target = dedent(
                    """
                    --- Original ---
                    ╙── 0
                        ├─╼ 1
                        │   ├─╼ 3
                        │   │   ├─╼ 7
                        │   │   └─╼ 8
                        │   └─╼ 4
                        │       ├─╼ 9
                        │       └─╼ 10
                        └─╼ 2
                            ├─╼ 5
                            │   ├─╼ 11
                            │   └─╼ 12
                            └─╼ 6
                                ├─╼ 13
                                └─╼ 14
                    --- Collapse Node 1 ---
                    ╙── 0
                        ├─╼ 1
                        │   └─╼  ...
                        └─╼ 2
                            ├─╼ 5
                            │   ├─╼ 11
                            │   └─╼ 12
                            └─╼ 6
                                ├─╼ 13
                                └─╼ 14
```

```
(8)                 --- Add alternate path (5, 3) to collapsed zone
(8)                 ╙── 0
(12)                    ├─╼ 1
(12)                    │   └─╼  ...
(12)                    └─╼ 2
(16)                        ├─╼ 5
(16)                        │   ├─╼ 11
(16)                        │   ├─╼ 12
(16)                        │   └─╼ 3 ━ 1
(16)                        │       ├─╼ 7
(16)                        │       └─╼ 8
(16)                        └─╼ 6
(20)                            ├─╼ 13
(20)                            └─╼ 14
(8)                 --- Collapse Node 0 ---
(8)                 ╙── 0
(12)                    └─╼  ...
(8)                 """
(4)             ).strip()
(4)             assert target == text
(0)         def test_collapse_undirected():
(4)             graph = nx.balanced_tree(r=2, h=3, create_using=nx.Graph)
(4)             lines = []
(4)             write = lines.append
(4)             write("--- Original ---")
(4)             nx.write_network_text(graph, path=write, end="", sources=[0])
(4)             graph.nodes[1]["collapse"] = True
(4)             write("--- Collapse Node 1 ---")
(4)             nx.write_network_text(graph, path=write, end="", sources=[0])
(4)             write("--- Add alternate path (5, 3) to collapsed zone")
(4)             graph.add_edge(5, 3)
(4)             nx.write_network_text(graph, path=write, end="", sources=[0])
(4)             write("--- Collapse Node 0 ---")
(4)             graph.nodes[0]["collapse"] = True
(4)             nx.write_network_text(graph, path=write, end="", sources=[0])
(4)             text = "\n".join(lines)
(4)             print(text)
(4)             target = dedent(
(8)                 """
(8)                 --- Original ---
(8)                 ╙── 0
(12)                    ├── 1
(12)                    │   ├── 3
(12)                    │   │   ├── 7
(12)                    │   │   └── 8
(12)                    │   └── 4
(12)                    │       ├── 9
(12)                    │       └── 10
(12)                    └── 2
(16)                        ├── 5
(16)                        │   ├── 11
(16)                        │   └── 12
(16)                        └── 6
(20)                            ├── 13
(20)                            └── 14
(8)                 --- Collapse Node 1 ---
(8)                 ╙── 0
(12)                    ├── 1 ━ 3, 4
(12)                    └──  ...
(12)                    └── 2
(16)                        ├── 5
(16)                        │   ├── 11
(16)                        │   └── 12
(16)                        └── 6
(20)                            ├── 13
(20)                            └── 14
(8)                 --- Add alternate path (5, 3) to collapsed zone
(8)                 ╙── 0
(12)                    ├── 1 ━ 3, 4
```

```
                        │   └──  ...
                        └── 2
                            ├── 5
                            │   ├── 11
                            │   ├── 12
                            │   └── 3 — 1
                            │       ├── 7
                            │       └── 8
                            └── 6
                                ├── 13
                                └── 14
                    --- Collapse Node 0 ---
                    ╙── 0 — 1, 2
                        └──  ...
                    """
                ).strip()
                assert target == text
            def generate_test_graphs():
                """
                Generate a gauntlet of different test graphs with different properties
                """
                import random
                rng = random.Random(976689776)
                num_randomized = 3
                for directed in [0, 1]:
                    cls = nx.DiGraph if directed else nx.Graph
                    for num_nodes in range(17):
                        graph = cls()
                        graph.add_nodes_from(range(num_nodes))
                        yield graph
                        if num_nodes > 0:
                            for p in [0.1, 0.3, 0.5, 0.7, 0.9]:
                                for seed in range(num_randomized):
                                    graph = nx.erdos_renyi_graph(
                                        num_nodes, p, directed=directed, seed=rng
                                    )
                                    yield graph
                            yield nx.complete_graph(num_nodes, cls)
                    yield nx.path_graph(3, create_using=cls)
                    yield nx.balanced_tree(r=1, h=3, create_using=cls)
                    if not directed:
                        yield nx.circular_ladder_graph(4, create_using=cls)
                        yield nx.star_graph(5, create_using=cls)
                        yield nx.lollipop_graph(4, 2, create_using=cls)
                        yield nx.wheel_graph(7, create_using=cls)
                        yield nx.dorogovtsev_goltsev_mendes_graph(4, create_using=cls)
            @pytest.mark.parametrize(
                ("vertical_chains", "ascii_only"),
                tuple(
                    [
                        (vertical_chains, ascii_only)
                        for vertical_chains in [0, 1]
                        for ascii_only in [0, 1]
                    ]
                ),
            )
            def test_network_text_round_trip(vertical_chains, ascii_only):
                """
                Write the graph to network text format, then parse it back in, assert it
is
                the same as the original graph. Passing this test is strong validation of
                both the format generator and parser.
                """
                from networkx.readwrite.text import _parse_network_text
                for graph in generate_test_graphs():
                    graph = nx.relabel_nodes(graph, {n: str(n) for n in graph.nodes})
                    lines = list(
                        nx.generate_network_text(
                            graph, vertical_chains=vertical_chains, ascii_only=ascii_only
```

```
1674: (12)                      )
1675: (8)                   )
1676: (8)                   new = _parse_network_text(lines)
1677: (8)                   try:
1678: (12)                      assert new.nodes == graph.nodes
1679: (12)                      assert new.edges == graph.edges
1680: (8)                   except Exception:
1681: (12)                      print("ERROR in round trip with graph")
1682: (12)                      nx.write_network_text(graph)
1683: (12)                      raise

----------------------------------------

File 537 - test_all_random_functions.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              import random
4: (0)              import networkx as nx
5: (0)              from networkx.algorithms import approximation as approx
6: (0)              from networkx.algorithms import threshold
7: (0)              progress = 0
8: (0)              np.random.seed(42)
9: (0)              np_rv = np.random.rand()
10: (0)             random.seed(42)
11: (0)             py_rv = random.random()
12: (0)             def t(f, *args, **kwds):
13: (4)                 """call one function and check if global RNG changed"""
14: (4)                 global progress
15: (4)                 progress += 1
16: (4)                 print(progress, ",", end="")
17: (4)                 f(*args, **kwds)
18: (4)                 after_np_rv = np.random.rand()
19: (4)                 assert np_rv == after_np_rv
20: (4)                 np.random.seed(42)
21: (4)                 after_py_rv = random.random()
22: (4)                 assert py_rv == after_py_rv
23: (4)                 random.seed(42)
24: (0)             def run_all_random_functions(seed):
25: (4)                 n = 20
26: (4)                 m = 10
27: (4)                 k = l = 2
28: (4)                 s = v = 10
29: (4)                 p = q = p1 = p2 = p_in = p_out = 0.4
30: (4)                 alpha = radius = theta = 0.75
31: (4)                 sizes = (20, 20, 10)
32: (4)                 colors = [1, 2, 3]
33: (4)                 G = nx.barbell_graph(12, 20)
34: (4)                 H = nx.cycle_graph(3)
35: (4)                 H.add_weighted_edges_from((u, v, 0.2) for u, v in H.edges)
36: (4)                 deg_sequence = [3, 2, 1, 3, 2, 1, 3, 2, 1, 2, 1, 2, 1]
37: (4)                 in_degree_sequence = w = sequence = aseq = bseq = deg_sequence
38: (4)                 t(nx.maximal_independent_set, G, seed=seed)
39: (4)                 t(nx.rich_club_coefficient, G, seed=seed, normalized=False)
40: (4)                 t(nx.random_reference, G, seed=seed)
41: (4)                 t(nx.lattice_reference, G, seed=seed)
42: (4)                 t(nx.sigma, G, 1, 2, seed=seed)
43: (4)                 t(nx.omega, G, 1, 2, seed=seed)
44: (4)                 t(nx.double_edge_swap, G, seed=seed)
45: (4)                 t(nx.connected_double_edge_swap, nx.complete_graph(9), seed=seed)
46: (4)                 t(nx.random_layout, G, seed=seed)
47: (4)                 t(nx.fruchterman_reingold_layout, G, seed=seed)
48: (4)                 t(nx.algebraic_connectivity, G, seed=seed)
49: (4)                 t(nx.fiedler_vector, G, seed=seed)
50: (4)                 t(nx.spectral_ordering, G, seed=seed)
51: (4)                 t(approx.average_clustering, G, seed=seed)
52: (4)                 t(approx.simulated_annealing_tsp, H, "greedy", source=1, seed=seed)
53: (4)                 t(approx.threshold_accepting_tsp, H, "greedy", source=1, seed=seed)
54: (4)                 t(
```

```
55: (8)                     approx.traveling_salesman_problem,
56: (8)                     H,
57: (8)                     method=lambda G, weight: approx.simulated_annealing_tsp(
58: (12)                        G, "greedy", weight, seed=seed
59: (8)                     ),
60: (4)                 )
61: (4)                 t(
62: (8)                     approx.traveling_salesman_problem,
63: (8)                     H,
64: (8)                     method=lambda G, weight: approx.threshold_accepting_tsp(
65: (12)                        G, "greedy", weight, seed=seed
66: (8)                     ),
67: (4)                 )
68: (4)                 t(nx.betweenness_centrality, G, seed=seed)
69: (4)                 t(nx.edge_betweenness_centrality, G, seed=seed)
70: (4)                 t(nx.approximate_current_flow_betweenness_centrality, G, seed=seed)
71: (4)                 t(nx.algorithms.community.kernighan_lin_bisection, G, seed=seed)
72: (4)                 t(nx.algorithms.tree.greedy_branching, G, seed=seed)
73: (4)                 t(nx.algorithms.tree.Edmonds, G, seed=seed)
74: (4)                 t(nx.spectral_graph_forge, G, alpha, seed=seed)
75: (4)                 t(nx.algorithms.community.asyn_fluidc, G, k, max_iter=1, seed=seed)
76: (4)                 t(
77: (8)
nx.algorithms.connectivity.edge_augmentation.greedy_k_edge_augmentation,
78: (8)                     G,
79: (8)                     k,
80: (8)                     seed=seed,
81: (4)                 )
82: (4)                 t(nx.algorithms.coloring.strategy_random_sequential, G, colors, seed=seed)
83: (4)                 cs = ["d", "i", "i", "d", "d", "i"]
84: (4)                 t(threshold.swap_d, cs, seed=seed)
85: (4)                 t(nx.configuration_model, deg_sequence, seed=seed)
86: (4)                 t(
87: (8)                     nx.directed_configuration_model,
88: (8)                     in_degree_sequence,
89: (8)                     in_degree_sequence,
90: (8)                     seed=seed,
91: (4)                 )
92: (4)                 t(nx.expected_degree_graph, w, seed=seed)
93: (4)                 t(nx.random_degree_sequence_graph, sequence, seed=seed)
94: (4)                 joint_degrees = {
95: (8)                     1: {4: 1},
96: (8)                     2: {2: 2, 3: 2, 4: 2},
97: (8)                     3: {2: 2, 4: 1},
98: (8)                     4: {1: 1, 2: 2, 3: 1},
99: (4)                 }
100: (4)                t(nx.joint_degree_graph, joint_degrees, seed=seed)
101: (4)                joint_degree_sequence = [
102: (8)                    (1, 0),
103: (8)                    (1, 0),
104: (8)                    (1, 0),
105: (8)                    (2, 0),
106: (8)                    (1, 0),
107: (8)                    (2, 1),
108: (8)                    (0, 1),
109: (8)                    (0, 1),
110: (4)                ]
111: (4)                t(nx.random_clustered_graph, joint_degree_sequence, seed=seed)
112: (4)                constructor = [(3, 3, 0.5), (10, 10, 0.7)]
113: (4)                t(nx.random_shell_graph, constructor, seed=seed)
114: (4)                t(nx.random_triad, G.to_directed(), seed=seed)
115: (4)                mapping = {1: 0.4, 2: 0.3, 3: 0.3}
116: (4)                t(nx.utils.random_weighted_sample, mapping, k, seed=seed)
117: (4)                t(nx.utils.weighted_choice, mapping, seed=seed)
118: (4)                t(nx.algorithms.bipartite.configuration_model, aseq, bseq, seed=seed)
119: (4)                t(nx.algorithms.bipartite.preferential_attachment_graph, aseq, p,
seed=seed)
120: (4)                def kernel_integral(u, w, z):
121: (8)                    return z - w
```

```
(4)                 t(nx.random_kernel_graph, n, kernel_integral, seed=seed)
(4)                 sizes = [75, 75, 300]
(4)                 probs = [[0.25, 0.05, 0.02], [0.05, 0.35, 0.07], [0.02, 0.07, 0.40]]
(4)                 t(nx.stochastic_block_model, sizes, probs, seed=seed)
(4)                 t(nx.random_partition_graph, sizes, p_in, p_out, seed=seed)
(4)                 t(threshold.random_threshold_sequence, n, p, seed=seed)
(4)                 t(nx.tournament.random_tournament, n, seed=seed)
(4)                 t(nx.relaxed_caveman_graph, l, k, p, seed=seed)
(4)                 t(nx.planted_partition_graph, l, k, p_in, p_out, seed=seed)
(4)                 t(nx.gaussian_random_partition_graph, n, s, v, p_in, p_out, seed=seed)
(4)                 t(nx.gn_graph, n, seed=seed)
(4)                 t(nx.gnr_graph, n, p, seed=seed)
(4)                 t(nx.gnc_graph, n, seed=seed)
(4)                 t(nx.scale_free_graph, n, seed=seed)
(4)                 t(nx.directed.random_uniform_k_out_graph, n, k, seed=seed)
(4)                 t(nx.random_k_out_graph, n, k, alpha, seed=seed)
(4)                 N = 1000
(4)                 t(nx.partial_duplication_graph, N, n, p, q, seed=seed)
(4)                 t(nx.duplication_divergence_graph, n, p, seed=seed)
(4)                 t(nx.random_geometric_graph, n, radius, seed=seed)
(4)                 t(nx.soft_random_geometric_graph, n, radius, seed=seed)
(4)                 t(nx.geographical_threshold_graph, n, theta, seed=seed)
(4)                 t(nx.waxman_graph, n, seed=seed)
(4)                 t(nx.navigable_small_world_graph, n, seed=seed)
(4)                 t(nx.thresholded_random_geometric_graph, n, radius, theta, seed=seed)
(4)                 t(nx.uniform_random_intersection_graph, n, m, p, seed=seed)
(4)                 t(nx.k_random_intersection_graph, n, m, k, seed=seed)
(4)                 t(nx.general_random_intersection_graph, n, 2, [0.1, 0.5], seed=seed)
(4)                 t(nx.fast_gnp_random_graph, n, p, seed=seed)
(4)                 t(nx.gnp_random_graph, n, p, seed=seed)
(4)                 t(nx.dense_gnm_random_graph, n, m, seed=seed)
(4)                 t(nx.gnm_random_graph, n, m, seed=seed)
(4)                 t(nx.newman_watts_strogatz_graph, n, k, p, seed=seed)
(4)                 t(nx.watts_strogatz_graph, n, k, p, seed=seed)
(4)                 t(nx.connected_watts_strogatz_graph, n, k, p, seed=seed)
(4)                 t(nx.random_regular_graph, 3, n, seed=seed)
(4)                 t(nx.barabasi_albert_graph, n, m, seed=seed)
(4)                 t(nx.extended_barabasi_albert_graph, n, m, p, q, seed=seed)
(4)                 t(nx.powerlaw_cluster_graph, n, m, p, seed=seed)
(4)                 t(nx.random_lobster, n, p1, p2, seed=seed)
(4)                 t(nx.random_powerlaw_tree, n, seed=seed, tries=5000)
(4)                 t(nx.random_powerlaw_tree_sequence, 10, seed=seed, tries=5000)
(4)                 t(nx.random_tree, n, seed=seed)
(4)                 t(nx.utils.powerlaw_sequence, n, seed=seed)
(4)                 t(nx.utils.zipf_rv, 2.3, seed=seed)
(4)                 cdist = [0.2, 0.4, 0.5, 0.7, 0.9, 1.0]
(4)                 t(nx.utils.discrete_sequence, n, cdistribution=cdist, seed=seed)
(4)                 t(nx.algorithms.bipartite.random_graph, n, m, p, seed=seed)
(4)                 t(nx.algorithms.bipartite.gnmk_random_graph, n, m, k, seed=seed)
(4)                 LFR = nx.generators.LFR_benchmark_graph
(4)                 t(
(8)                     LFR,
(8)                     25,
(8)                     3,
(8)                     1.5,
(8)                     0.1,
(8)                     average_degree=3,
(8)                     min_community=10,
(8)                     seed=seed,
(8)                     max_community=20,
(4)                 )
(4)                 t(nx.random_internet_as_graph, n, seed=seed)
(0)             @pytest.mark.slow
(0)             def test_rng_interface():
(4)                 global progress
(4)                 for seed in [14, np.random.RandomState(14)]:
(8)                     np.random.seed(42)
(8)                     random.seed(42)
(8)                     run_all_random_functions(seed)
```

```
191: (8)                    progress = 0
192: (8)                    after_np_rv = np.random.rand()
193: (8)                    assert np_rv == after_np_rv
194: (8)                    after_py_rv = random.random()
195: (8)                    assert py_rv == after_py_rv
```

File 538 - test_convert.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.convert import (
4: (4)                  from_dict_of_dicts,
5: (4)                  from_dict_of_lists,
6: (4)                  to_dict_of_dicts,
7: (4)                  to_dict_of_lists,
8: (4)                  to_networkx_graph,
9: (0)              )
10: (0)             from networkx.generators.classic import barbell_graph, cycle_graph
11: (0)             from networkx.utils import edges_equal, graphs_equal, nodes_equal
12: (0)             class TestConvert:
13: (4)                 def edgelists_equal(self, e1, e2):
14: (8)                     return sorted(sorted(e) for e in e1) == sorted(sorted(e) for e in e2)
15: (4)                 def test_simple_graphs(self):
16: (8)                     for dest, source in [
17: (12)                        (to_dict_of_dicts, from_dict_of_dicts),
18: (12)                        (to_dict_of_lists, from_dict_of_lists),
19: (8)                     ]:
20: (12)                        G = barbell_graph(10, 3)
21: (12)                        G.graph = {}
22: (12)                        dod = dest(G)
23: (12)                        GG = source(dod)
24: (12)                        assert graphs_equal(G, GG)
25: (12)                        GW = to_networkx_graph(dod)
26: (12)                        assert graphs_equal(G, GW)
27: (12)                        GI = nx.Graph(dod)
28: (12)                        assert graphs_equal(G, GI)
29: (12)                        P4 = nx.path_graph(4)
30: (12)                        P3 = nx.path_graph(3)
31: (12)                        P4.graph = {}
32: (12)                        P3.graph = {}
33: (12)                        dod = dest(P4, nodelist=[0, 1, 2])
34: (12)                        Gdod = nx.Graph(dod)
35: (12)                        assert graphs_equal(Gdod, P3)
36: (4)                 def test_exceptions(self):
37: (8)                     class G:
38: (12)                        adj = None
39: (8)                     pytest.raises(nx.NetworkXError, to_networkx_graph, G)
40: (8)                     class G:
41: (12)                        is_strict = None
42: (8)                     pytest.raises(nx.NetworkXError, to_networkx_graph, G)
43: (8)                     G = {"a": 0}
44: (8)                     pytest.raises(TypeError, to_networkx_graph, G)
45: (8)                     class G:
46: (12)                        next = None
47: (8)                     pytest.raises(nx.NetworkXError, to_networkx_graph, G)
48: (8)                     pytest.raises(nx.NetworkXError, to_networkx_graph, "a")
49: (4)                 def test_digraphs(self):
50: (8)                     for dest, source in [
51: (12)                        (to_dict_of_dicts, from_dict_of_dicts),
52: (12)                        (to_dict_of_lists, from_dict_of_lists),
53: (8)                     ]:
54: (12)                        G = cycle_graph(10)
55: (12)                        dod = dest(G)
56: (12)                        GG = source(dod)
57: (12)                        assert nodes_equal(sorted(G.nodes()), sorted(GG.nodes()))
58: (12)                        assert edges_equal(sorted(G.edges()), sorted(GG.edges()))
59: (12)                        GW = to_networkx_graph(dod)
```

```
60: (12)                        assert nodes_equal(sorted(G.nodes()), sorted(GW.nodes()))
61: (12)                        assert edges_equal(sorted(G.edges()), sorted(GW.edges()))
62: (12)                        GI = nx.Graph(dod)
63: (12)                        assert nodes_equal(sorted(G.nodes()), sorted(GI.nodes()))
64: (12)                        assert edges_equal(sorted(G.edges()), sorted(GI.edges()))
65: (12)                        G = cycle_graph(10, create_using=nx.DiGraph)
66: (12)                        dod = dest(G)
67: (12)                        GG = source(dod, create_using=nx.DiGraph)
68: (12)                        assert sorted(G.nodes()) == sorted(GG.nodes())
69: (12)                        assert sorted(G.edges()) == sorted(GG.edges())
70: (12)                        GW = to_networkx_graph(dod, create_using=nx.DiGraph)
71: (12)                        assert sorted(G.nodes()) == sorted(GW.nodes())
72: (12)                        assert sorted(G.edges()) == sorted(GW.edges())
73: (12)                        GI = nx.DiGraph(dod)
74: (12)                        assert sorted(G.nodes()) == sorted(GI.nodes())
75: (12)                        assert sorted(G.edges()) == sorted(GI.edges())
76: (4)                 def test_graph(self):
77: (8)                     g = nx.cycle_graph(10)
78: (8)                     G = nx.Graph()
79: (8)                     G.add_nodes_from(g)
80: (8)                     G.add_weighted_edges_from((u, v, u) for u, v in g.edges())
81: (8)                     dod = to_dict_of_dicts(G)
82: (8)                     GG = from_dict_of_dicts(dod, create_using=nx.Graph)
83: (8)                     assert nodes_equal(sorted(G.nodes()), sorted(GG.nodes()))
84: (8)                     assert edges_equal(sorted(G.edges()), sorted(GG.edges()))
85: (8)                     GW = to_networkx_graph(dod, create_using=nx.Graph)
86: (8)                     assert nodes_equal(sorted(G.nodes()), sorted(GW.nodes()))
87: (8)                     assert edges_equal(sorted(G.edges()), sorted(GW.edges()))
88: (8)                     GI = nx.Graph(dod)
89: (8)                     assert sorted(G.nodes()) == sorted(GI.nodes())
90: (8)                     assert sorted(G.edges()) == sorted(GI.edges())
91: (8)                     dol = to_dict_of_lists(G)
92: (8)                     GG = from_dict_of_lists(dol, create_using=nx.Graph)
93: (8)                     enone = [(u, v, {}) for (u, v, d) in G.edges(data=True)]
94: (8)                     assert nodes_equal(sorted(G.nodes()), sorted(GG.nodes()))
95: (8)                     assert edges_equal(enone, sorted(GG.edges(data=True)))
96: (8)                     GW = to_networkx_graph(dol, create_using=nx.Graph)
97: (8)                     assert nodes_equal(sorted(G.nodes()), sorted(GW.nodes()))
98: (8)                     assert edges_equal(enone, sorted(GW.edges(data=True)))
99: (8)                     GI = nx.Graph(dol)
100: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(GI.nodes()))
101: (8)                    assert edges_equal(enone, sorted(GI.edges(data=True)))
102: (4)                def test_with_multiedges_self_loops(self):
103: (8)                    G = cycle_graph(10)
104: (8)                    XG = nx.Graph()
105: (8)                    XG.add_nodes_from(G)
106: (8)                    XG.add_weighted_edges_from((u, v, u) for u, v in G.edges())
107: (8)                    XGM = nx.MultiGraph()
108: (8)                    XGM.add_nodes_from(G)
109: (8)                    XGM.add_weighted_edges_from((u, v, u) for u, v in G.edges())
110: (8)                    XGM.add_edge(0, 1, weight=2)  # multiedge
111: (8)                    XGS = nx.Graph()
112: (8)                    XGS.add_nodes_from(G)
113: (8)                    XGS.add_weighted_edges_from((u, v, u) for u, v in G.edges())
114: (8)                    XGS.add_edge(0, 0, weight=100)  # self loop
115: (8)                    dod = to_dict_of_dicts(XGS)
116: (8)                    GG = from_dict_of_dicts(dod, create_using=nx.Graph)
117: (8)                    assert nodes_equal(XGS.nodes(), GG.nodes())
118: (8)                    assert edges_equal(XGS.edges(), GG.edges())
119: (8)                    GW = to_networkx_graph(dod, create_using=nx.Graph)
120: (8)                    assert nodes_equal(XGS.nodes(), GW.nodes())
121: (8)                    assert edges_equal(XGS.edges(), GW.edges())
122: (8)                    GI = nx.Graph(dod)
123: (8)                    assert nodes_equal(XGS.nodes(), GI.nodes())
124: (8)                    assert edges_equal(XGS.edges(), GI.edges())
125: (8)                    dol = to_dict_of_lists(XGS)
126: (8)                    GG = from_dict_of_lists(dol, create_using=nx.Graph)
127: (8)                    enone = [(u, v, {}) for (u, v, d) in XGS.edges(data=True)]
128: (8)                    assert nodes_equal(sorted(XGS.nodes()), sorted(GG.nodes()))
```

```
129: (8)                    assert edges_equal(enone, sorted(GG.edges(data=True)))
130: (8)                    GW = to_networkx_graph(dol, create_using=nx.Graph)
131: (8)                    assert nodes_equal(sorted(XGS.nodes()), sorted(GW.nodes()))
132: (8)                    assert edges_equal(enone, sorted(GW.edges(data=True)))
133: (8)                    GI = nx.Graph(dol)
134: (8)                    assert nodes_equal(sorted(XGS.nodes()), sorted(GI.nodes()))
135: (8)                    assert edges_equal(enone, sorted(GI.edges(data=True)))
136: (8)                    dod = to_dict_of_dicts(XGM)
137: (8)                    GG = from_dict_of_dicts(dod, create_using=nx.MultiGraph,
multigraph_input=True)
138: (8)                    assert nodes_equal(sorted(XGM.nodes()), sorted(GG.nodes()))
139: (8)                    assert edges_equal(sorted(XGM.edges()), sorted(GG.edges()))
140: (8)                    GW = to_networkx_graph(dod, create_using=nx.MultiGraph,
multigraph_input=True)
141: (8)                    assert nodes_equal(sorted(XGM.nodes()), sorted(GW.nodes()))
142: (8)                    assert edges_equal(sorted(XGM.edges()), sorted(GW.edges()))
143: (8)                    GI = nx.MultiGraph(dod)
144: (8)                    assert nodes_equal(sorted(XGM.nodes()), sorted(GI.nodes()))
145: (8)                    assert sorted(XGM.edges()) == sorted(GI.edges())
146: (8)                    GE = from_dict_of_dicts(dod, create_using=nx.MultiGraph,
multigraph_input=False)
147: (8)                    assert nodes_equal(sorted(XGM.nodes()), sorted(GE.nodes()))
148: (8)                    assert sorted(XGM.edges()) != sorted(GE.edges())
149: (8)                    GI = nx.MultiGraph(XGM)
150: (8)                    assert nodes_equal(sorted(XGM.nodes()), sorted(GI.nodes()))
151: (8)                    assert edges_equal(sorted(XGM.edges()), sorted(GI.edges()))
152: (8)                    GM = nx.MultiGraph(G)
153: (8)                    assert nodes_equal(sorted(GM.nodes()), sorted(G.nodes()))
154: (8)                    assert edges_equal(sorted(GM.edges()), sorted(G.edges()))
155: (8)                    dol = to_dict_of_lists(G)
156: (8)                    GG = from_dict_of_lists(dol, create_using=nx.MultiGraph)
157: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(GG.nodes()))
158: (8)                    assert edges_equal(sorted(G.edges()), sorted(GG.edges()))
159: (8)                    GW = to_networkx_graph(dol, create_using=nx.MultiGraph)
160: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(GW.nodes()))
161: (8)                    assert edges_equal(sorted(G.edges()), sorted(GW.edges()))
162: (8)                    GI = nx.MultiGraph(dol)
163: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(GI.nodes()))
164: (8)                    assert edges_equal(sorted(G.edges()), sorted(GI.edges()))
165: (4)                def test_edgelists(self):
166: (8)                    P = nx.path_graph(4)
167: (8)                    e = [(0, 1), (1, 2), (2, 3)]
168: (8)                    G = nx.Graph(e)
169: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(P.nodes()))
170: (8)                    assert edges_equal(sorted(G.edges()), sorted(P.edges()))
171: (8)                    assert edges_equal(sorted(G.edges(data=True)),
sorted(P.edges(data=True)))
172: (8)                    e = [(0, 1, {}), (1, 2, {}), (2, 3, {})]
173: (8)                    G = nx.Graph(e)
174: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(P.nodes()))
175: (8)                    assert edges_equal(sorted(G.edges()), sorted(P.edges()))
176: (8)                    assert edges_equal(sorted(G.edges(data=True)),
sorted(P.edges(data=True)))
177: (8)                    e = ((n, n + 1) for n in range(3))
178: (8)                    G = nx.Graph(e)
179: (8)                    assert nodes_equal(sorted(G.nodes()), sorted(P.nodes()))
180: (8)                    assert edges_equal(sorted(G.edges()), sorted(P.edges()))
181: (8)                    assert edges_equal(sorted(G.edges(data=True)),
sorted(P.edges(data=True)))
182: (4)                def test_directed_to_undirected(self):
183: (8)                    edges1 = [(0, 1), (1, 2), (2, 0)]
184: (8)                    edges2 = [(0, 1), (1, 2), (0, 2)]
185: (8)                    assert self.edgelists_equal(nx.Graph(nx.DiGraph(edges1)).edges(),
edges1)
186: (8)                    assert self.edgelists_equal(nx.Graph(nx.DiGraph(edges2)).edges(),
edges1)
187: (8)                    assert self.edgelists_equal(nx.MultiGraph(nx.DiGraph(edges1)).edges(),
edges1)
188: (8)                    assert self.edgelists_equal(nx.MultiGraph(nx.DiGraph(edges2)).edges(),
```

```
edges1)
189: (8)                    assert self.edgelists_equal(
190: (12)                       nx.MultiGraph(nx.MultiDiGraph(edges1)).edges(), edges1
191: (8)                    )
192: (8)                    assert self.edgelists_equal(
193: (12)                       nx.MultiGraph(nx.MultiDiGraph(edges2)).edges(), edges1
194: (8)                    )
195: (8)                    assert self.edgelists_equal(nx.Graph(nx.MultiDiGraph(edges1)).edges(),
edges1)
196: (8)                    assert self.edgelists_equal(nx.Graph(nx.MultiDiGraph(edges2)).edges(),
edges1)
197: (4)                def test_attribute_dict_integrity(self):
198: (8)                    G = nx.Graph()
199: (8)                    G.add_nodes_from("abc")
200: (8)                    H = to_networkx_graph(G, create_using=nx.Graph)
201: (8)                    assert list(H.nodes) == list(G.nodes)
202: (8)                    H = nx.DiGraph(G)
203: (8)                    assert list(H.nodes) == list(G.nodes)
204: (4)                def test_to_edgelist(self):
205: (8)                    G = nx.Graph([(1, 1)])
206: (8)                    elist = nx.to_edgelist(G, nodelist=list(G))
207: (8)                    assert edges_equal(G.edges(data=True), elist)
208: (4)                def test_custom_node_attr_dict_safekeeping(self):
209: (8)                    class custom_dict(dict):
210: (12)                       pass
211: (8)                    class Custom(nx.Graph):
212: (12)                       node_attr_dict_factory = custom_dict
213: (8)                    g = nx.Graph()
214: (8)                    g.add_node(1, weight=1)
215: (8)                    h = Custom(g)
216: (8)                    assert isinstance(g._node[1], dict)
217: (8)                    assert isinstance(h._node[1], custom_dict)
218: (0)            @pytest.mark.parametrize(
219: (4)                "edgelist",
220: (4)                (
221: (8)                    [(0, 1), (1, 2)],
222: (8)                    [(0, 1, {"weight": 1.0}), (1, 2, {"weight": 2.0})],
223: (4)                ),
224: (0)            )
225: (0)            def test_to_dict_of_dicts_with_edgedata_param(edgelist):
226: (4)                G = nx.Graph()
227: (4)                G.add_edges_from(edgelist)
228: (4)                expected = {0: {1: 10}, 1: {0: 10, 2: 10}, 2: {1: 10}}
229: (4)                assert nx.to_dict_of_dicts(G, edge_data=10) == expected
230: (0)            def test_to_dict_of_dicts_with_edgedata_and_nodelist():
231: (4)                G = nx.path_graph(5)
232: (4)                nodelist = [2, 3, 4]
233: (4)                expected = {2: {3: 10}, 3: {2: 10, 4: 10}, 4: {3: 10}}
234: (4)                assert nx.to_dict_of_dicts(G, nodelist=nodelist, edge_data=10) == expected
235: (0)            def test_to_dict_of_dicts_with_edgedata_multigraph():
236: (4)                """Multi edge data overwritten when edge_data != None"""
237: (4)                G = nx.MultiGraph()
238: (4)                G.add_edge(0, 1, key="a")
239: (4)                G.add_edge(0, 1, key="b")
240: (4)                expected = {0: {1: 10}, 1: {0: 10}}
241: (4)                assert nx.to_dict_of_dicts(G, edge_data=10) == expected
242: (0)            def test_to_networkx_graph_non_edgelist():
243: (4)                invalid_edgelist = [1, 2, 3]
244: (4)                with pytest.raises(nx.NetworkXError, match="Input is not a valid edge
list"):
245: (8)                    nx.to_networkx_graph(invalid_edgelist)

----------------------------------------

File 539 - test_convert_numpy.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              npt = pytest.importorskip("numpy.testing")
```

```
4: (0)              import networkx as nx
5: (0)              from networkx.generators.classic import barbell_graph, cycle_graph, path_graph
6: (0)              from networkx.utils import graphs_equal
7: (0)              class TestConvertNumpyArray:
8: (4)                  def setup_method(self):
9: (8)                      self.G1 = barbell_graph(10, 3)
10: (8)                     self.G2 = cycle_graph(10, create_using=nx.DiGraph)
11: (8)                     self.G3 = self.create_weighted(nx.Graph())
12: (8)                     self.G4 = self.create_weighted(nx.DiGraph())
13: (4)                 def create_weighted(self, G):
14: (8)                     g = cycle_graph(4)
15: (8)                     G.add_nodes_from(g)
16: (8)                     G.add_weighted_edges_from((u, v, 10 + u) for u, v in g.edges())
17: (8)                     return G
18: (4)                 def assert_equal(self, G1, G2):
19: (8)                     assert sorted(G1.nodes()) == sorted(G2.nodes())
20: (8)                     assert sorted(G1.edges()) == sorted(G2.edges())
21: (4)                 def identity_conversion(self, G, A, create_using):
22: (8)                     assert A.sum() > 0
23: (8)                     GG = nx.from_numpy_array(A, create_using=create_using)
24: (8)                     self.assert_equal(G, GG)
25: (8)                     GW = nx.to_networkx_graph(A, create_using=create_using)
26: (8)                     self.assert_equal(G, GW)
27: (8)                     GI = nx.empty_graph(0, create_using).__class__(A)
28: (8)                     self.assert_equal(G, GI)
29: (4)                 def test_shape(self):
30: (8)                     "Conversion from non-square array."
31: (8)                     A = np.array([[1, 2, 3], [4, 5, 6]])
32: (8)                     pytest.raises(nx.NetworkXError, nx.from_numpy_array, A)
33: (4)                 def test_identity_graph_array(self):
34: (8)                     "Conversion from graph to array to graph."
35: (8)                     A = nx.to_numpy_array(self.G1)
36: (8)                     self.identity_conversion(self.G1, A, nx.Graph())
37: (4)                 def test_identity_digraph_array(self):
38: (8)                     """Conversion from digraph to array to digraph."""
39: (8)                     A = nx.to_numpy_array(self.G2)
40: (8)                     self.identity_conversion(self.G2, A, nx.DiGraph())
41: (4)                 def test_identity_weighted_graph_array(self):
42: (8)                     """Conversion from weighted graph to array to weighted graph."""
43: (8)                     A = nx.to_numpy_array(self.G3)
44: (8)                     self.identity_conversion(self.G3, A, nx.Graph())
45: (4)                 def test_identity_weighted_digraph_array(self):
46: (8)                     """Conversion from weighted digraph to array to weighted digraph."""
47: (8)                     A = nx.to_numpy_array(self.G4)
48: (8)                     self.identity_conversion(self.G4, A, nx.DiGraph())
49: (4)                 def test_nodelist(self):
50: (8)                     """Conversion from graph to array to graph with nodelist."""
51: (8)                     P4 = path_graph(4)
52: (8)                     P3 = path_graph(3)
53: (8)                     nodelist = list(P3)
54: (8)                     A = nx.to_numpy_array(P4, nodelist=nodelist)
55: (8)                     GA = nx.Graph(A)
56: (8)                     self.assert_equal(GA, P3)
57: (8)                     nodelist += [nodelist[0]]
58: (8)                     pytest.raises(nx.NetworkXError, nx.to_numpy_array, P3,
nodelist=nodelist)
59: (8)                     nodelist = [-1, 0, 1]
60: (8)                     with pytest.raises(
61: (12)                        nx.NetworkXError,
62: (12)                        match=f"Nodes {nodelist - P3.nodes} in nodelist is not in G",
63: (8)                     ):
64: (12)                        nx.to_numpy_array(P3, nodelist=nodelist)
65: (4)                 def test_weight_keyword(self):
66: (8)                     WP4 = nx.Graph()
67: (8)                     WP4.add_edges_from((n, n + 1, {"weight": 0.5, "other": 0.3}) for n in
range(3))
68: (8)                     P4 = path_graph(4)
69: (8)                     A = nx.to_numpy_array(P4)
70: (8)                     np.testing.assert_equal(A, nx.to_numpy_array(WP4, weight=None))
```

```
71: (8)                     np.testing.assert_equal(0.5 * A, nx.to_numpy_array(WP4))
72: (8)                     np.testing.assert_equal(0.3 * A, nx.to_numpy_array(WP4,
weight="other"))
73: (4)                 def test_from_numpy_array_type(self):
74: (8)                     A = np.array([[1]])
75: (8)                     G = nx.from_numpy_array(A)
76: (8)                     assert type(G[0][0]["weight"]) == int
77: (8)                     A = np.array([[1]]).astype(float)
78: (8)                     G = nx.from_numpy_array(A)
79: (8)                     assert type(G[0][0]["weight"]) == float
80: (8)                     A = np.array([[1]]).astype(str)
81: (8)                     G = nx.from_numpy_array(A)
82: (8)                     assert type(G[0][0]["weight"]) == str
83: (8)                     A = np.array([[1]]).astype(bool)
84: (8)                     G = nx.from_numpy_array(A)
85: (8)                     assert type(G[0][0]["weight"]) == bool
86: (8)                     A = np.array([[1]]).astype(complex)
87: (8)                     G = nx.from_numpy_array(A)
88: (8)                     assert type(G[0][0]["weight"]) == complex
89: (8)                     A = np.array([[1]]).astype(object)
90: (8)                     pytest.raises(TypeError, nx.from_numpy_array, A)
91: (8)                     A = np.array([[[1, 1, 1], [1, 1, 1]], [[1, 1, 1], [1, 1, 1]]])
92: (8)                     with pytest.raises(
93: (12)                        nx.NetworkXError, match=f"Input array must be 2D, not {A.ndim}"
94: (8)                     ):
95: (12)                        g = nx.from_numpy_array(A)
96: (4)                 def test_from_numpy_array_dtype(self):
97: (8)                     dt = [("weight", float), ("cost", int)]
98: (8)                     A = np.array([[(1.0, 2)]], dtype=dt)
99: (8)                     G = nx.from_numpy_array(A)
100: (8)                    assert type(G[0][0]["weight"]) == float
101: (8)                    assert type(G[0][0]["cost"]) == int
102: (8)                    assert G[0][0]["cost"] == 2
103: (8)                    assert G[0][0]["weight"] == 1.0
104: (4)                def test_from_numpy_array_parallel_edges(self):
105: (8)                    """Tests that the :func:`networkx.from_numpy_array` function
106: (8)                    interprets integer weights as the number of parallel edges when
107: (8)                    creating a multigraph.
108: (8)                    """
109: (8)                    A = np.array([[1, 1], [1, 2]])
110: (8)                    expected = nx.DiGraph()
111: (8)                    edges = [(0, 0), (0, 1), (1, 0)]
112: (8)                    expected.add_weighted_edges_from([(u, v, 1) for (u, v) in edges])
113: (8)                    expected.add_edge(1, 1, weight=2)
114: (8)                    actual = nx.from_numpy_array(A, parallel_edges=True,
create_using=nx.DiGraph)
115: (8)                    assert graphs_equal(actual, expected)
116: (8)                    actual = nx.from_numpy_array(A, parallel_edges=False,
create_using=nx.DiGraph)
117: (8)                    assert graphs_equal(actual, expected)
118: (8)                    edges = [(0, 0), (0, 1), (1, 0), (1, 1), (1, 1)]
119: (8)                    expected = nx.MultiDiGraph()
120: (8)                    expected.add_weighted_edges_from([(u, v, 1) for (u, v) in edges])
121: (8)                    actual = nx.from_numpy_array(
122: (12)                       A, parallel_edges=True, create_using=nx.MultiDiGraph
123: (8)                    )
124: (8)                    assert graphs_equal(actual, expected)
125: (8)                    expected = nx.MultiDiGraph()
126: (8)                    expected.add_edges_from(set(edges), weight=1)
127: (8)                    expected[1][1][0]["weight"] = 2
128: (8)                    actual = nx.from_numpy_array(
129: (12)                       A, parallel_edges=False, create_using=nx.MultiDiGraph
130: (8)                    )
131: (8)                    assert graphs_equal(actual, expected)
132: (4)                @pytest.mark.parametrize(
133: (8)                    "dt",
134: (8)                    (
135: (12)                       None,  # default
136: (12)                       int,  # integer dtype
```

```
(12)                        np.dtype(
(16)                            [("weight", "f8"), ("color", "i1")]
(12)                        ),  # Structured dtype with named fields
(8)                     ),
(4)                 )
(4)                 def test_from_numpy_array_no_edge_attr(self, dt):
(8)                     A = np.array([[0, 1], [1, 0]], dtype=dt)
(8)                     G = nx.from_numpy_array(A, edge_attr=None)
(8)                     assert "weight" not in G.edges[0, 1]
(8)                     assert len(G.edges[0, 1]) == 0
(4)                 def test_from_numpy_array_multiedge_no_edge_attr(self):
(8)                     A = np.array([[0, 2], [2, 0]])
(8)                     G = nx.from_numpy_array(A, create_using=nx.MultiDiGraph,
edge_attr=None)
(8)                     assert all("weight" not in e for _, e in G[0][1].items())
(8)                     assert len(G[0][1][0]) == 0
(4)                 def test_from_numpy_array_custom_edge_attr(self):
(8)                     A = np.array([[0, 2], [3, 0]])
(8)                     G = nx.from_numpy_array(A, edge_attr="cost")
(8)                     assert "weight" not in G.edges[0, 1]
(8)                     assert G.edges[0, 1]["cost"] == 3
(4)                 def test_symmetric(self):
(8)                     """Tests that a symmetric array has edges added only once to an
(8)                     undirected multigraph when using :func:`networkx.from_numpy_array`.
(8)                     """
(8)                     A = np.array([[0, 1], [1, 0]])
(8)                     G = nx.from_numpy_array(A, create_using=nx.MultiGraph)
(8)                     expected = nx.MultiGraph()
(8)                     expected.add_edge(0, 1, weight=1)
(8)                     assert graphs_equal(G, expected)
(4)                 def test_dtype_int_graph(self):
(8)                     """Test that setting dtype int actually gives an integer array.
(8)                     For more information, see GitHub pull request #1363.
(8)                     """
(8)                     G = nx.complete_graph(3)
(8)                     A = nx.to_numpy_array(G, dtype=int)
(8)                     assert A.dtype == int
(4)                 def test_dtype_int_multigraph(self):
(8)                     """Test that setting dtype int actually gives an integer array.
(8)                     For more information, see GitHub pull request #1363.
(8)                     """
(8)                     G = nx.MultiGraph(nx.complete_graph(3))
(8)                     A = nx.to_numpy_array(G, dtype=int)
(8)                     assert A.dtype == int
(0)             @pytest.fixture
(0)             def multigraph_test_graph():
(4)                 G = nx.MultiGraph()
(4)                 G.add_edge(1, 2, weight=7)
(4)                 G.add_edge(1, 2, weight=70)
(4)                 return G
(0)             @pytest.mark.parametrize(("operator", "expected"), ((sum, 77), (min, 7), (max,
70)))
(0)             def test_numpy_multigraph(multigraph_test_graph, operator, expected):
(4)                 A = nx.to_numpy_array(multigraph_test_graph, multigraph_weight=operator)
(4)                 assert A[1, 0] == expected
(0)             def test_to_numpy_array_multigraph_nodelist(multigraph_test_graph):
(4)                 G = multigraph_test_graph
(4)                 G.add_edge(0, 1, weight=3)
(4)                 A = nx.to_numpy_array(G, nodelist=[1, 2])
(4)                 assert A.shape == (2, 2)
(4)                 assert A[1, 0] == 77
(0)             @pytest.mark.parametrize(
(4)                 "G, expected",
(4)                 [
(8)                     (nx.Graph(), np.array([[0, 1 + 2j], [1 + 2j, 0]], dtype=complex)),
(8)                     (nx.DiGraph(), np.array([[0, 1 + 2j], [0, 0]], dtype=complex)),
(4)                 ],
(0)             )
(0)             def test_to_numpy_array_complex_weights(G, expected):
```

```
204: (4)                G.add_edge(0, 1, weight=1 + 2j)
205: (4)                A = nx.to_numpy_array(G, dtype=complex)
206: (4)                npt.assert_array_equal(A, expected)
207: (0)            def test_to_numpy_array_arbitrary_weights():
208: (4)                G = nx.DiGraph()
209: (4)                w = 922337203685477580102  # Out of range for int64
210: (4)                G.add_edge(0, 1, weight=922337203685477580102)  # val not representable by
int64
211: (4)                A = nx.to_numpy_array(G, dtype=object)
212: (4)                expected = np.array([[0, w], [0, 0]], dtype=object)
213: (4)                npt.assert_array_equal(A, expected)
214: (4)                A = nx.to_numpy_array(G.to_undirected(), dtype=object)
215: (4)                expected = np.array([[0, w], [w, 0]], dtype=object)
216: (4)                npt.assert_array_equal(A, expected)
217: (0)            @pytest.mark.parametrize(
218: (4)                "func, expected",
219: (4)                ((min, -1), (max, 10), (sum, 11), (np.mean, 11 / 3), (np.median, 2)),
220: (0)            )
221: (0)            def test_to_numpy_array_multiweight_reduction(func, expected):
222: (4)                """Test various functions for reducing multiedge weights."""
223: (4)                G = nx.MultiDiGraph()
224: (4)                weights = [-1, 2, 10.0]
225: (4)                for w in weights:
226: (8)                    G.add_edge(0, 1, weight=w)
227: (4)                A = nx.to_numpy_array(G, multigraph_weight=func, dtype=float)
228: (4)                assert np.allclose(A, [[0, expected], [0, 0]])
229: (4)                A = nx.to_numpy_array(G.to_undirected(), multigraph_weight=func,
dtype=float)
230: (4)                assert np.allclose(A, [[0, expected], [expected, 0]])
231: (0)            @pytest.mark.parametrize(
232: (4)                ("G, expected"),
233: (4)                [
234: (8)                    (nx.Graph(), [[(0, 0), (10, 5)], [(10, 5), (0, 0)]]),
235: (8)                    (nx.DiGraph(), [[(0, 0), (10, 5)], [(0, 0), (0, 0)]]),
236: (4)                ],
237: (0)            )
238: (0)            def test_to_numpy_array_structured_dtype_attrs_from_fields(G, expected):
239: (4)                """When `dtype` is structured (i.e. has names) and `weight` is None, use
240: (4)                the named fields of the dtype to look up edge attributes."""
241: (4)                G.add_edge(0, 1, weight=10, cost=5.0)
242: (4)                dtype = np.dtype([("weight", int), ("cost", int)])
243: (4)                A = nx.to_numpy_array(G, dtype=dtype, weight=None)
244: (4)                expected = np.asarray(expected, dtype=dtype)
245: (4)                npt.assert_array_equal(A, expected)
246: (0)            def test_to_numpy_array_structured_dtype_single_attr_default():
247: (4)                G = nx.path_graph(3)
248: (4)                dtype = np.dtype([("weight", float)])  # A single named field
249: (4)                A = nx.to_numpy_array(G, dtype=dtype, weight=None)
250: (4)                expected = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]], dtype=float)
251: (4)                npt.assert_array_equal(A["weight"], expected)
252: (0)            @pytest.mark.parametrize(
253: (4)                ("field_name", "expected_attr_val"),
254: (4)                [
255: (8)                    ("weight", 1),
256: (8)                    ("cost", 3),
257: (4)                ],
258: (0)            )
259: (0)            def test_to_numpy_array_structured_dtype_single_attr(field_name,
expected_attr_val):
260: (4)                G = nx.Graph()
261: (4)                G.add_edge(0, 1, cost=3)
262: (4)                dtype = np.dtype([(field_name, float)])
263: (4)                A = nx.to_numpy_array(G, dtype=dtype, weight=None)
264: (4)                expected = np.array([[0, expected_attr_val], [expected_attr_val, 0]],
dtype=float)
265: (4)                npt.assert_array_equal(A[field_name], expected)
266: (0)            @pytest.mark.parametrize("graph_type", (nx.Graph, nx.DiGraph))
267: (0)            @pytest.mark.parametrize(
268: (4)                "edge",
```

```
269: (4)                [
270: (8)                    (0, 1),  # No edge attributes
271: (8)                    (0, 1, {"weight": 10}),  # One edge attr
272: (8)                    (0, 1, {"weight": 5, "flow": -4}),  # Multiple but not all edge attrs
273: (8)                    (0, 1, {"weight": 2.0, "cost": 10, "flow": -45}),  # All attrs
274: (4)                ],
275: (0)            )
276: (0)            def test_to_numpy_array_structured_dtype_multiple_fields(graph_type, edge):
277: (4)                G = graph_type([edge])
278: (4)                dtype = np.dtype([("weight", float), ("cost", float), ("flow", float)])
279: (4)                A = nx.to_numpy_array(G, dtype=dtype, weight=None)
280: (4)                for attr in dtype.names:
281: (8)                    expected = nx.to_numpy_array(G, dtype=float, weight=attr)
282: (8)                    npt.assert_array_equal(A[attr], expected)
283: (0)            @pytest.mark.parametrize("G", (nx.Graph(), nx.DiGraph()))
284: (0)            def test_to_numpy_array_structured_dtype_scalar_nonedge(G):
285: (4)                G.add_edge(0, 1, weight=10)
286: (4)                dtype = np.dtype([("weight", float), ("cost", float)])
287: (4)                A = nx.to_numpy_array(G, dtype=dtype, weight=None, nonedge=np.nan)
288: (4)                for attr in dtype.names:
289: (8)                    expected = nx.to_numpy_array(G, dtype=float, weight=attr,
nonedge=np.nan)
290: (8)                    npt.assert_array_equal(A[attr], expected)
291: (0)            @pytest.mark.parametrize("G", (nx.Graph(), nx.DiGraph()))
292: (0)            def test_to_numpy_array_structured_dtype_nonedge_ary(G):
293: (4)                """Similar to the scalar case, except has a different non-edge value for
294: (4)                each named field."""
295: (4)                G.add_edge(0, 1, weight=10)
296: (4)                dtype = np.dtype([("weight", float), ("cost", float)])
297: (4)                nonedges = np.array([(0, np.inf)], dtype=dtype)
298: (4)                A = nx.to_numpy_array(G, dtype=dtype, weight=None, nonedge=nonedges)
299: (4)                for attr in dtype.names:
300: (8)                    nonedge = nonedges[attr]
301: (8)                    expected = nx.to_numpy_array(G, dtype=float, weight=attr,
nonedge=nonedge)
302: (8)                    npt.assert_array_equal(A[attr], expected)
303: (0)            def test_to_numpy_array_structured_dtype_with_weight_raises():
304: (4)                """Using both a structured dtype (with named fields) and specifying a
`weight`
305: (4)                parameter is ambiguous."""
306: (4)                G = nx.path_graph(3)
307: (4)                dtype = np.dtype([("weight", int), ("cost", int)])
308: (4)                exception_msg = "Specifying `weight` not supported for structured dtypes"
309: (4)                with pytest.raises(ValueError, match=exception_msg):
310: (8)                    nx.to_numpy_array(G, dtype=dtype)  # Default is weight="weight"
311: (4)                with pytest.raises(ValueError, match=exception_msg):
312: (8)                    nx.to_numpy_array(G, dtype=dtype, weight="cost")
313: (0)            @pytest.mark.parametrize("graph_type", (nx.MultiGraph, nx.MultiDiGraph))
314: (0)            def test_to_numpy_array_structured_multigraph_raises(graph_type):
315: (4)                G = nx.path_graph(3, create_using=graph_type)
316: (4)                dtype = np.dtype([("weight", int), ("cost", int)])
317: (4)                with pytest.raises(nx.NetworkXError, match="Structured arrays are not
supported"):
318: (8)                    nx.to_numpy_array(G, dtype=dtype, weight=None)

----------------------------------------

File 540 - test_convert_pandas.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import edges_equal, graphs_equal, nodes_equal
4: (0)              np = pytest.importorskip("numpy")
5: (0)              pd = pytest.importorskip("pandas")
6: (0)              class TestConvertPandas:
7: (4)                  def setup_method(self):
8: (8)                      self.rng = np.random.RandomState(seed=5)
9: (8)                      ints = self.rng.randint(1, 11, size=(3, 2))
10: (8)                     a = ["A", "B", "C"]
```

```
 11: (8)                     b = ["D", "A", "E"]
 12: (8)                     df = pd.DataFrame(ints, columns=["weight", "cost"])
 13: (8)                     df[0] = a  # Column label 0 (int)
 14: (8)                     df["b"] = b  # Column label 'b' (str)
 15: (8)                     self.df = df
 16: (8)                     mdf = pd.DataFrame([[4, 16, "A", "D"]], columns=["weight", "cost", 0,
"b"])
 17: (8)                     self.mdf = pd.concat([df, mdf])
 18: (4)                 def test_exceptions(self):
 19: (8)                     G = pd.DataFrame(["a"])  # adj
 20: (8)                     pytest.raises(nx.NetworkXError, nx.to_networkx_graph, G)
 21: (8)                     G = pd.DataFrame(["a", 0.0])  # elist
 22: (8)                     pytest.raises(nx.NetworkXError, nx.to_networkx_graph, G)
 23: (8)                     df = pd.DataFrame([[1, 1], [1, 0]], dtype=int, index=[1, 2], columns=
["a", "b"])
 24: (8)                     pytest.raises(nx.NetworkXError, nx.from_pandas_adjacency, df)
 25: (4)                 def test_from_edgelist_all_attr(self):
 26: (8)                     Gtrue = nx.Graph(
 27: (12)                        [
 28: (16)                            ("E", "C", {"cost": 9, "weight": 10}),
 29: (16)                            ("B", "A", {"cost": 1, "weight": 7}),
 30: (16)                            ("A", "D", {"cost": 7, "weight": 4}),
 31: (12)                        ]
 32: (8)                     )
 33: (8)                     G = nx.from_pandas_edgelist(self.df, 0, "b", True)
 34: (8)                     assert graphs_equal(G, Gtrue)
 35: (8)                     MGtrue = nx.MultiGraph(Gtrue)
 36: (8)                     MGtrue.add_edge("A", "D", cost=16, weight=4)
 37: (8)                     MG = nx.from_pandas_edgelist(self.mdf, 0, "b", True, nx.MultiGraph())
 38: (8)                     assert graphs_equal(MG, MGtrue)
 39: (4)                 def test_from_edgelist_multi_attr(self):
 40: (8)                     Gtrue = nx.Graph(
 41: (12)                        [
 42: (16)                            ("E", "C", {"cost": 9, "weight": 10}),
 43: (16)                            ("B", "A", {"cost": 1, "weight": 7}),
 44: (16)                            ("A", "D", {"cost": 7, "weight": 4}),
 45: (12)                        ]
 46: (8)                     )
 47: (8)                     G = nx.from_pandas_edgelist(self.df, 0, "b", ["weight", "cost"])
 48: (8)                     assert graphs_equal(G, Gtrue)
 49: (4)                 def test_from_edgelist_multi_attr_incl_target(self):
 50: (8)                     Gtrue = nx.Graph(
 51: (12)                        [
 52: (16)                            ("E", "C", {0: "C", "b": "E", "weight": 10}),
 53: (16)                            ("B", "A", {0: "B", "b": "A", "weight": 7}),
 54: (16)                            ("A", "D", {0: "A", "b": "D", "weight": 4}),
 55: (12)                        ]
 56: (8)                     )
 57: (8)                     G = nx.from_pandas_edgelist(self.df, 0, "b", [0, "b", "weight"])
 58: (8)                     assert graphs_equal(G, Gtrue)
 59: (4)                 def test_from_edgelist_multidigraph_and_edge_attr(self):
 60: (8)                     edges = [
 61: (12)                        ("X1", "X4", {"Co": "zA", "Mi": 0, "St": "X1"}),
 62: (12)                        ("X1", "X4", {"Co": "zB", "Mi": 54, "St": "X2"}),
 63: (12)                        ("X1", "X4", {"Co": "zB", "Mi": 49, "St": "X3"}),
 64: (12)                        ("X1", "X4", {"Co": "zB", "Mi": 44, "St": "X4"}),
 65: (12)                        ("Y1", "Y3", {"Co": "zC", "Mi": 0, "St": "Y1"}),
 66: (12)                        ("Y1", "Y3", {"Co": "zC", "Mi": 34, "St": "Y2"}),
 67: (12)                        ("Y1", "Y3", {"Co": "zC", "Mi": 29, "St": "X2"}),
 68: (12)                        ("Y1", "Y3", {"Co": "zC", "Mi": 24, "St": "Y3"}),
 69: (12)                        ("Z1", "Z3", {"Co": "zD", "Mi": 0, "St": "Z1"}),
 70: (12)                        ("Z1", "Z3", {"Co": "zD", "Mi": 14, "St": "X3"}),
 71: (8)                     ]
 72: (8)                     Gtrue = nx.MultiDiGraph(edges)
 73: (8)                     data = {
 74: (12)                        "O": ["X1", "X1", "X1", "X1", "Y1", "Y1", "Y1", "Y1", "Z1", "Z1"],
 75: (12)                        "D": ["X4", "X4", "X4", "X4", "Y3", "Y3", "Y3", "Y3", "Z3", "Z3"],
 76: (12)                        "St": ["X1", "X2", "X3", "X4", "Y1", "Y2", "X2", "Y3", "Z1",
"X3"],
```

```
 77: (12)                        "Co": ["zA", "zB", "zB", "zB", "zC", "zC", "zC", "zC", "zD",
 "zD"],
 78: (12)                        "Mi": [0, 54, 49, 44, 0, 34, 29, 24, 0, 14],
 79: (8)                     }
 80: (8)                     df = pd.DataFrame.from_dict(data)
 81: (8)                     G1 = nx.from_pandas_edgelist(
 82: (12)                        df, source="O", target="D", edge_attr=True,
 create_using=nx.MultiDiGraph
 83: (8)                     )
 84: (8)                     G2 = nx.from_pandas_edgelist(
 85: (12)                        df,
 86: (12)                        source="O",
 87: (12)                        target="D",
 88: (12)                        edge_attr=["St", "Co", "Mi"],
 89: (12)                        create_using=nx.MultiDiGraph,
 90: (8)                     )
 91: (8)                     assert graphs_equal(G1, Gtrue)
 92: (8)                     assert graphs_equal(G2, Gtrue)
 93: (4)                 def test_from_edgelist_one_attr(self):
 94: (8)                     Gtrue = nx.Graph(
 95: (12)                        [
 96: (16)                            ("E", "C", {"weight": 10}),
 97: (16)                            ("B", "A", {"weight": 7}),
 98: (16)                            ("A", "D", {"weight": 4}),
 99: (12)                        ]
 100: (8)                    )
 101: (8)                    G = nx.from_pandas_edgelist(self.df, 0, "b", "weight")
 102: (8)                    assert graphs_equal(G, Gtrue)
 103: (4)                def test_from_edgelist_int_attr_name(self):
 104: (8)                    Gtrue = nx.Graph(
 105: (12)                       [("E", "C", {0: "C"}), ("B", "A", {0: "B"}), ("A", "D", {0: "A"})]
 106: (8)                    )
 107: (8)                    G = nx.from_pandas_edgelist(self.df, 0, "b", 0)
 108: (8)                    assert graphs_equal(G, Gtrue)
 109: (4)                def test_from_edgelist_invalid_attr(self):
 110: (8)                    pytest.raises(
 111: (12)                       nx.NetworkXError, nx.from_pandas_edgelist, self.df, 0, "b",
 "misspell"
 112: (8)                    )
 113: (8)                    pytest.raises(nx.NetworkXError, nx.from_pandas_edgelist, self.df, 0,
 "b", 1)
 114: (8)                    edgeframe = pd.DataFrame([[0, 1], [1, 2], [2, 0]], columns=["s", "t"])
 115: (8)                    pytest.raises(
 116: (12)                       nx.NetworkXError, nx.from_pandas_edgelist, edgeframe, "s", "t",
 True
 117: (8)                    )
 118: (8)                    pytest.raises(
 119: (12)                       nx.NetworkXError, nx.from_pandas_edgelist, edgeframe, "s", "t",
 "weight"
 120: (8)                    )
 121: (8)                    pytest.raises(
 122: (12)                       nx.NetworkXError,
 123: (12)                       nx.from_pandas_edgelist,
 124: (12)                       edgeframe,
 125: (12)                       "s",
 126: (12)                       "t",
 127: (12)                       ["weight", "size"],
 128: (8)                    )
 129: (4)                def test_from_edgelist_no_attr(self):
 130: (8)                    Gtrue = nx.Graph([("E", "C", {}), ("B", "A", {}), ("A", "D", {})])
 131: (8)                    G = nx.from_pandas_edgelist(self.df, 0, "b")
 132: (8)                    assert graphs_equal(G, Gtrue)
 133: (4)                def test_from_edgelist(self):
 134: (8)                    G = nx.cycle_graph(10)
 135: (8)                    G.add_weighted_edges_from((u, v, u) for u, v in list(G.edges))
 136: (8)                    edgelist = nx.to_edgelist(G)
 137: (8)                    source = [s for s, t, d in edgelist]
 138: (8)                    target = [t for s, t, d in edgelist]
 139: (8)                    weight = [d["weight"] for s, t, d in edgelist]
```

```
140: (8)                    edges = pd.DataFrame({"source": source, "target": target, "weight":
weight})
141: (8)                    GG = nx.from_pandas_edgelist(edges, edge_attr="weight")
142: (8)                    assert nodes_equal(G.nodes(), GG.nodes())
143: (8)                    assert edges_equal(G.edges(), GG.edges())
144: (8)                    GW = nx.to_networkx_graph(edges, create_using=nx.Graph)
145: (8)                    assert nodes_equal(G.nodes(), GW.nodes())
146: (8)                    assert edges_equal(G.edges(), GW.edges())
147: (4)                def test_to_edgelist_default_source_or_target_col_exists(self):
148: (8)                    G = nx.path_graph(10)
149: (8)                    G.add_weighted_edges_from((u, v, u) for u, v in list(G.edges))
150: (8)                    nx.set_edge_attributes(G, 0, name="source")
151: (8)                    pytest.raises(nx.NetworkXError, nx.to_pandas_edgelist, G)
152: (8)                    for u, v, d in G.edges(data=True):
153: (12)                       d.pop("source", None)
154: (8)                    nx.set_edge_attributes(G, 0, name="target")
155: (8)                    pytest.raises(nx.NetworkXError, nx.to_pandas_edgelist, G)
156: (4)                def test_to_edgelist_custom_source_or_target_col_exists(self):
157: (8)                    G = nx.path_graph(10)
158: (8)                    G.add_weighted_edges_from((u, v, u) for u, v in list(G.edges))
159: (8)                    nx.set_edge_attributes(G, 0, name="source_col_name")
160: (8)                    pytest.raises(
161: (12)                       nx.NetworkXError, nx.to_pandas_edgelist, G,
source="source_col_name"
162: (8)                    )
163: (8)                    for u, v, d in G.edges(data=True):
164: (12)                       d.pop("source_col_name", None)
165: (8)                    nx.set_edge_attributes(G, 0, name="target_col_name")
166: (8)                    pytest.raises(
167: (12)                       nx.NetworkXError, nx.to_pandas_edgelist, G,
target="target_col_name"
168: (8)                    )
169: (4)                def test_to_edgelist_edge_key_col_exists(self):
170: (8)                    G = nx.path_graph(10, create_using=nx.MultiGraph)
171: (8)                    G.add_weighted_edges_from((u, v, u) for u, v in list(G.edges()))
172: (8)                    nx.set_edge_attributes(G, 0, name="edge_key_name")
173: (8)                    pytest.raises(
174: (12)                       nx.NetworkXError, nx.to_pandas_edgelist, G,
edge_key="edge_key_name"
175: (8)                    )
176: (4)                def test_from_adjacency(self):
177: (8)                    nodelist = [1, 2]
178: (8)                    dftrue = pd.DataFrame(
179: (12)                       [[1, 1], [1, 0]], dtype=int, index=nodelist, columns=nodelist
180: (8)                    )
181: (8)                    G = nx.Graph([(1, 1), (1, 2)])
182: (8)                    df = nx.to_pandas_adjacency(G, dtype=int)
183: (8)                    pd.testing.assert_frame_equal(df, dftrue)
184: (4)                @pytest.mark.parametrize("graph", [nx.Graph, nx.MultiGraph])
185: (4)                def test_roundtrip(self, graph):
186: (8)                    Gtrue = graph([(1, 1), (1, 2)])
187: (8)                    df = nx.to_pandas_edgelist(Gtrue)
188: (8)                    G = nx.from_pandas_edgelist(df, create_using=graph)
189: (8)                    assert graphs_equal(Gtrue, G)
190: (8)                    adj = {1: {1: {"weight": 1}, 2: {"weight": 1}}, 2: {1: {"weight": 1}}}
191: (8)                    Gtrue = graph(adj)
192: (8)                    df = nx.to_pandas_adjacency(Gtrue, dtype=int)
193: (8)                    G = nx.from_pandas_adjacency(df, create_using=graph)
194: (8)                    assert graphs_equal(Gtrue, G)
195: (4)                def test_from_adjacency_named(self):
196: (8)                    data = {
197: (12)                       "A": {"A": 0, "B": 0, "C": 0},
198: (12)                       "B": {"A": 1, "B": 0, "C": 0},
199: (12)                       "C": {"A": 0, "B": 1, "C": 0},
200: (8)                    }
201: (8)                    dftrue = pd.DataFrame(data, dtype=np.intp)
202: (8)                    df = dftrue[["A", "C", "B"]]
203: (8)                    G = nx.from_pandas_adjacency(df, create_using=nx.DiGraph())
204: (8)                    df = nx.to_pandas_adjacency(G, dtype=np.intp)
```

```
(8)                     pd.testing.assert_frame_equal(df, dftrue)
(4)                 def test_edgekey_with_multigraph(self):
(8)                     df = pd.DataFrame(
(12)                        {
(16)                            "source": {"A": "N1", "B": "N2", "C": "N1", "D": "N1"},
(16)                            "target": {"A": "N2", "B": "N3", "C": "N1", "D": "N2"},
(16)                            "attr1": {"A": "F1", "B": "F2", "C": "F3", "D": "F4"},
(16)                            "attr2": {"A": 1, "B": 0, "C": 0, "D": 0},
(16)                            "attr3": {"A": 0, "B": 1, "C": 0, "D": 1},
(12)                        }
(8)                     )
(8)                     Gtrue = nx.MultiGraph(
(12)                        [
(16)                            ("N1", "N2", "F1", {"attr2": 1, "attr3": 0}),
(16)                            ("N2", "N3", "F2", {"attr2": 0, "attr3": 1}),
(16)                            ("N1", "N1", "F3", {"attr2": 0, "attr3": 0}),
(16)                            ("N1", "N2", "F4", {"attr2": 0, "attr3": 1}),
(12)                        ]
(8)                     )
(8)                     G = nx.from_pandas_edgelist(
(12)                        df,
(12)                        source="source",
(12)                        target="target",
(12)                        edge_attr=["attr2", "attr3"],
(12)                        edge_key="attr1",
(12)                        create_using=nx.MultiGraph(),
(8)                     )
(8)                     assert graphs_equal(G, Gtrue)
(8)                     df_roundtrip = nx.to_pandas_edgelist(G, edge_key="attr1")
(8)                     df_roundtrip = df_roundtrip.sort_values("attr1")
(8)                     df_roundtrip.index = ["A", "B", "C", "D"]
(8)                     pd.testing.assert_frame_equal(
(12)                        df, df_roundtrip[["source", "target", "attr1", "attr2", "attr3"]]
(8)                     )
(4)                 def test_edgekey_with_normal_graph_no_action(self):
(8)                     Gtrue = nx.Graph(
(12)                        [
(16)                            ("E", "C", {"cost": 9, "weight": 10}),
(16)                            ("B", "A", {"cost": 1, "weight": 7}),
(16)                            ("A", "D", {"cost": 7, "weight": 4}),
(12)                        ]
(8)                     )
(8)                     G = nx.from_pandas_edgelist(self.df, 0, "b", True, edge_key="weight")
(8)                     assert graphs_equal(G, Gtrue)
(4)                 def test_nonexisting_edgekey_raises(self):
(8)                     with pytest.raises(nx.exception.NetworkXError):
(12)                        nx.from_pandas_edgelist(
(16)                            self.df,
(16)                            source="source",
(16)                            target="target",
(16)                            edge_key="Not_real",
(16)                            edge_attr=True,
(16)                            create_using=nx.MultiGraph(),
(12)                        )
(0)             def test_to_pandas_adjacency_with_nodelist():
(4)                 G = nx.complete_graph(5)
(4)                 nodelist = [1, 4]
(4)                 expected = pd.DataFrame(
(8)                     [[0, 1], [1, 0]], dtype=int, index=nodelist, columns=nodelist
(4)                 )
(4)                 pd.testing.assert_frame_equal(
(8)                     expected, nx.to_pandas_adjacency(G, nodelist, dtype=int)
(4)                 )
(0)             def test_to_pandas_edgelist_with_nodelist():
(4)                 G = nx.Graph()
(4)                 G.add_edges_from([(0, 1), (1, 2), (1, 3)], weight=2.0)
(4)                 G.add_edge(0, 5, weight=100)
(4)                 df = nx.to_pandas_edgelist(G, nodelist=[1, 2])
(4)                 assert 0 not in df["source"].to_numpy()
```

```
274: (4)                assert 100 not in df["weight"].to_numpy()

----------------------------------------

File 541 - test_exceptions.py:

1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              def test_raises_networkxexception():
4: (4)                  with pytest.raises(nx.NetworkXException):
5: (8)                      raise nx.NetworkXException
6: (0)              def test_raises_networkxerr():
7: (4)                  with pytest.raises(nx.NetworkXError):
8: (8)                      raise nx.NetworkXError
9: (0)              def test_raises_networkx_pointless_concept():
10: (4)                 with pytest.raises(nx.NetworkXPointlessConcept):
11: (8)                     raise nx.NetworkXPointlessConcept
12: (0)             def test_raises_networkxalgorithmerr():
13: (4)                 with pytest.raises(nx.NetworkXAlgorithmError):
14: (8)                     raise nx.NetworkXAlgorithmError
15: (0)             def test_raises_networkx_unfeasible():
16: (4)                 with pytest.raises(nx.NetworkXUnfeasible):
17: (8)                     raise nx.NetworkXUnfeasible
18: (0)             def test_raises_networkx_no_path():
19: (4)                 with pytest.raises(nx.NetworkXNoPath):
20: (8)                     raise nx.NetworkXNoPath
21: (0)             def test_raises_networkx_unbounded():
22: (4)                 with pytest.raises(nx.NetworkXUnbounded):
23: (8)                     raise nx.NetworkXUnbounded

----------------------------------------

File 542 - test_convert_scipy.py:

1: (0)              import pytest
2: (0)              np = pytest.importorskip("numpy")
3: (0)              sp = pytest.importorskip("scipy")
4: (0)              import networkx as nx
5: (0)              from networkx.generators.classic import barbell_graph, cycle_graph, path_graph
6: (0)              from networkx.utils import graphs_equal
7: (0)              class TestConvertScipy:
8: (4)                  def setup_method(self):
9: (8)                      self.G1 = barbell_graph(10, 3)
10: (8)                     self.G2 = cycle_graph(10, create_using=nx.DiGraph)
11: (8)                     self.G3 = self.create_weighted(nx.Graph())
12: (8)                     self.G4 = self.create_weighted(nx.DiGraph())
13: (4)                 def test_exceptions(self):
14: (8)                     class G:
15: (12)                        format = None
16: (8)                     pytest.raises(nx.NetworkXError, nx.to_networkx_graph, G)
17: (4)                 def create_weighted(self, G):
18: (8)                     g = cycle_graph(4)
19: (8)                     e = list(g.edges())
20: (8)                     source = [u for u, v in e]
21: (8)                     dest = [v for u, v in e]
22: (8)                     weight = [s + 10 for s in source]
23: (8)                     ex = zip(source, dest, weight)
24: (8)                     G.add_weighted_edges_from(ex)
25: (8)                     return G
26: (4)                 def identity_conversion(self, G, A, create_using):
27: (8)                     GG = nx.from_scipy_sparse_array(A, create_using=create_using)
28: (8)                     assert nx.is_isomorphic(G, GG)
29: (8)                     GW = nx.to_networkx_graph(A, create_using=create_using)
30: (8)                     assert nx.is_isomorphic(G, GW)
31: (8)                     GI = nx.empty_graph(0, create_using).__class__(A)
32: (8)                     assert nx.is_isomorphic(G, GI)
33: (8)                     ACSR = A.tocsr()
34: (8)                     GI = nx.empty_graph(0, create_using).__class__(ACSR)
35: (8)                     assert nx.is_isomorphic(G, GI)
```

```
 36: (8)                     ACOO = A.tocoo()
 37: (8)                     GI = nx.empty_graph(0, create_using).__class__(ACOO)
 38: (8)                     assert nx.is_isomorphic(G, GI)
 39: (8)                     ACSC = A.tocsc()
 40: (8)                     GI = nx.empty_graph(0, create_using).__class__(ACSC)
 41: (8)                     assert nx.is_isomorphic(G, GI)
 42: (8)                     AD = A.todense()
 43: (8)                     GI = nx.empty_graph(0, create_using).__class__(AD)
 44: (8)                     assert nx.is_isomorphic(G, GI)
 45: (8)                     AA = A.toarray()
 46: (8)                     GI = nx.empty_graph(0, create_using).__class__(AA)
 47: (8)                     assert nx.is_isomorphic(G, GI)
 48: (4)                 def test_shape(self):
 49: (8)                     "Conversion from non-square sparse array."
 50: (8)                     A = sp.sparse.lil_array([[1, 2, 3], [4, 5, 6]])
 51: (8)                     pytest.raises(nx.NetworkXError, nx.from_scipy_sparse_array, A)
 52: (4)                 def test_identity_graph_matrix(self):
 53: (8)                     "Conversion from graph to sparse matrix to graph."
 54: (8)                     A = nx.to_scipy_sparse_array(self.G1)
 55: (8)                     self.identity_conversion(self.G1, A, nx.Graph())
 56: (4)                 def test_identity_digraph_matrix(self):
 57: (8)                     "Conversion from digraph to sparse matrix to digraph."
 58: (8)                     A = nx.to_scipy_sparse_array(self.G2)
 59: (8)                     self.identity_conversion(self.G2, A, nx.DiGraph())
 60: (4)                 def test_identity_weighted_graph_matrix(self):
 61: (8)                     """Conversion from weighted graph to sparse matrix to weighted
 graph."""
 62: (8)                     A = nx.to_scipy_sparse_array(self.G3)
 63: (8)                     self.identity_conversion(self.G3, A, nx.Graph())
 64: (4)                 def test_identity_weighted_digraph_matrix(self):
 65: (8)                     """Conversion from weighted digraph to sparse matrix to weighted
 digraph."""
 66: (8)                     A = nx.to_scipy_sparse_array(self.G4)
 67: (8)                     self.identity_conversion(self.G4, A, nx.DiGraph())
 68: (4)                 def test_nodelist(self):
 69: (8)                     """Conversion from graph to sparse matrix to graph with nodelist."""
 70: (8)                     P4 = path_graph(4)
 71: (8)                     P3 = path_graph(3)
 72: (8)                     nodelist = list(P3.nodes())
 73: (8)                     A = nx.to_scipy_sparse_array(P4, nodelist=nodelist)
 74: (8)                     GA = nx.Graph(A)
 75: (8)                     assert nx.is_isomorphic(GA, P3)
 76: (8)                     pytest.raises(nx.NetworkXError, nx.to_scipy_sparse_array, P3,
 nodelist=[])
 77: (8)                     long_nl = nodelist + [0]
 78: (8)                     pytest.raises(nx.NetworkXError, nx.to_scipy_sparse_array, P3,
 nodelist=long_nl)
 79: (8)                     non_nl = [-1, 0, 1, 2]
 80: (8)                     pytest.raises(nx.NetworkXError, nx.to_scipy_sparse_array, P3,
 nodelist=non_nl)
 81: (4)                 def test_weight_keyword(self):
 82: (8)                     WP4 = nx.Graph()
 83: (8)                     WP4.add_edges_from((n, n + 1, {"weight": 0.5, "other": 0.3}) for n in
 range(3))
 84: (8)                     P4 = path_graph(4)
 85: (8)                     A = nx.to_scipy_sparse_array(P4)
 86: (8)                     np.testing.assert_equal(
 87: (12)                        A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
 88: (8)                     )
 89: (8)                     np.testing.assert_equal(
 90: (12)                        0.5 * A.todense(), nx.to_scipy_sparse_array(WP4).todense()
 91: (8)                     )
 92: (8)                     np.testing.assert_equal(
 93: (12)                        0.3 * A.todense(), nx.to_scipy_sparse_array(WP4,
 weight="other").todense()
 94: (8)                     )
 95: (4)                 def test_format_keyword(self):
 96: (8)                     WP4 = nx.Graph()
 97: (8)                     WP4.add_edges_from((n, n + 1, {"weight": 0.5, "other": 0.3}) for n in
```

```
range(3))
        P4 = path_graph(4)
        A = nx.to_scipy_sparse_array(P4, format="csr")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
        A = nx.to_scipy_sparse_array(P4, format="csc")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
        A = nx.to_scipy_sparse_array(P4, format="coo")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
        A = nx.to_scipy_sparse_array(P4, format="bsr")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
        A = nx.to_scipy_sparse_array(P4, format="lil")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
        A = nx.to_scipy_sparse_array(P4, format="dia")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
        A = nx.to_scipy_sparse_array(P4, format="dok")
        np.testing.assert_equal(
            A.todense(), nx.to_scipy_sparse_array(WP4, weight=None).todense()
        )
    def test_format_keyword_raise(self):
        with pytest.raises(nx.NetworkXError):
            WP4 = nx.Graph()
            WP4.add_edges_from(
                (n, n + 1, {"weight": 0.5, "other": 0.3}) for n in range(3)
            )
            P4 = path_graph(4)
            nx.to_scipy_sparse_array(P4, format="any_other")
    def test_null_raise(self):
        with pytest.raises(nx.NetworkXError):
            nx.to_scipy_sparse_array(nx.Graph())
    def test_empty(self):
        G = nx.Graph()
        G.add_node(1)
        M = nx.to_scipy_sparse_array(G)
        np.testing.assert_equal(M.toarray(), np.array([[0]]))
    def test_ordering(self):
        G = nx.DiGraph()
        G.add_edge(1, 2)
        G.add_edge(2, 3)
        G.add_edge(3, 1)
        M = nx.to_scipy_sparse_array(G, nodelist=[3, 2, 1])
        np.testing.assert_equal(
            M.toarray(), np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]])
        )
    def test_selfloop_graph(self):
        G = nx.Graph([(1, 1)])
        M = nx.to_scipy_sparse_array(G)
        np.testing.assert_equal(M.toarray(), np.array([[1]]))
        G.add_edges_from([(2, 3), (3, 4)])
        M = nx.to_scipy_sparse_array(G, nodelist=[2, 3, 4])
        np.testing.assert_equal(
            M.toarray(), np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]])
        )
    def test_selfloop_digraph(self):
        G = nx.DiGraph([(1, 1)])
        M = nx.to_scipy_sparse_array(G)
        np.testing.assert_equal(M.toarray(), np.array([[1]]))
        G.add_edges_from([(2, 3), (3, 4)])
```

```
166: (8)                    M = nx.to_scipy_sparse_array(G, nodelist=[2, 3, 4])
167: (8)                    np.testing.assert_equal(
168: (12)                       M.toarray(), np.array([[0, 1, 0], [0, 0, 1], [0, 0, 0]])
169: (8)                    )
170: (4)                def test_from_scipy_sparse_array_parallel_edges(self):
171: (8)                    """Tests that the :func:`networkx.from_scipy_sparse_array` function
172: (8)                    interprets integer weights as the number of parallel edges when
173: (8)                    creating a multigraph.
174: (8)                    """
175: (8)                    A = sp.sparse.csr_array([[1, 1], [1, 2]])
176: (8)                    expected = nx.DiGraph()
177: (8)                    edges = [(0, 0), (0, 1), (1, 0)]
178: (8)                    expected.add_weighted_edges_from([(u, v, 1) for (u, v) in edges])
179: (8)                    expected.add_edge(1, 1, weight=2)
180: (8)                    actual = nx.from_scipy_sparse_array(
181: (12)                       A, parallel_edges=True, create_using=nx.DiGraph
182: (8)                    )
183: (8)                    assert graphs_equal(actual, expected)
184: (8)                    actual = nx.from_scipy_sparse_array(
185: (12)                       A, parallel_edges=False, create_using=nx.DiGraph
186: (8)                    )
187: (8)                    assert graphs_equal(actual, expected)
188: (8)                    edges = [(0, 0), (0, 1), (1, 0), (1, 1), (1, 1)]
189: (8)                    expected = nx.MultiDiGraph()
190: (8)                    expected.add_weighted_edges_from([(u, v, 1) for (u, v) in edges])
191: (8)                    actual = nx.from_scipy_sparse_array(
192: (12)                       A, parallel_edges=True, create_using=nx.MultiDiGraph
193: (8)                    )
194: (8)                    assert graphs_equal(actual, expected)
195: (8)                    expected = nx.MultiDiGraph()
196: (8)                    expected.add_edges_from(set(edges), weight=1)
197: (8)                    expected[1][1][0]["weight"] = 2
198: (8)                    actual = nx.from_scipy_sparse_array(
199: (12)                       A, parallel_edges=False, create_using=nx.MultiDiGraph
200: (8)                    )
201: (8)                    assert graphs_equal(actual, expected)
202: (4)                def test_symmetric(self):
203: (8)                    """Tests that a symmetric matrix has edges added only once to an
204: (8)                    undirected multigraph when using
205: (8)                    :func:`networkx.from_scipy_sparse_array`.
206: (8)                    """
207: (8)                    A = sp.sparse.csr_array([[0, 1], [1, 0]])
208: (8)                    G = nx.from_scipy_sparse_array(A, create_using=nx.MultiGraph)
209: (8)                    expected = nx.MultiGraph()
210: (8)                    expected.add_edge(0, 1, weight=1)
211: (8)                    assert graphs_equal(G, expected)
212: (0)            @pytest.mark.parametrize("sparse_format", ("csr", "csc", "dok"))
213: (0)            def test_from_scipy_sparse_array_formats(sparse_format):
214: (4)                """Test all formats supported by _generate_weighted_edges."""
215: (4)                expected = nx.Graph()
216: (4)                expected.add_edges_from(
217: (8)                    [
218: (12)                       (0, 1, {"weight": 3}),
219: (12)                       (0, 2, {"weight": 2}),
220: (12)                       (1, 0, {"weight": 3}),
221: (12)                       (1, 2, {"weight": 1}),
222: (12)                       (2, 0, {"weight": 2}),
223: (12)                       (2, 1, {"weight": 1}),
224: (8)                    ]
225: (4)                )
226: (4)                A = sp.sparse.coo_array([[0, 3, 2], [3, 0, 1], [2, 1,
0]]).asformat(sparse_format)
227: (4)                assert graphs_equal(expected, nx.from_scipy_sparse_array(A))

----------------------------------------

File 543 - test_import.py:

1: (0)              import pytest
```

```
2: (0)              def test_namespace_alias():
3: (4)                  with pytest.raises(ImportError):
4: (8)                      from networkx import nx
5: (0)              def test_namespace_nesting():
6: (4)                  with pytest.raises(ImportError):
7: (8)                      from networkx import networkx
```

File 544 - test_lazy_imports.py:

```
1: (0)              import importlib
2: (0)              import sys
3: (0)              import types
4: (0)              import pytest
5: (0)              import networkx.lazy_imports as lazy
6: (0)              def test_lazy_import_basics():
7: (4)                  math = lazy._lazy_import("math")
8: (4)                  anything_not_real = lazy._lazy_import("anything_not_real")
9: (4)                  assert math.sin(math.pi) == pytest.approx(0, 1e-6)
10: (4)                 try:
11: (8)                     anything_not_real.pi
12: (8)                     assert False  # Should not get here
13: (4)                 except ModuleNotFoundError:
14: (8)                     pass
15: (4)                 assert isinstance(anything_not_real, lazy.DelayedImportErrorModule)
16: (4)                 try:
17: (8)                     anything_not_real.pi
18: (8)                     assert False  # Should not get here
19: (4)                 except ModuleNotFoundError:
20: (8)                     pass
21: (0)             def test_lazy_import_impact_on_sys_modules():
22: (4)                 math = lazy._lazy_import("math")
23: (4)                 anything_not_real = lazy._lazy_import("anything_not_real")
24: (4)                 assert type(math) == types.ModuleType
25: (4)                 assert "math" in sys.modules
26: (4)                 assert type(anything_not_real) == lazy.DelayedImportErrorModule
27: (4)                 assert "anything_not_real" not in sys.modules
28: (4)                 np_test = pytest.importorskip("numpy")
29: (4)                 np = lazy._lazy_import("numpy")
30: (4)                 assert type(np) == types.ModuleType
31: (4)                 assert "numpy" in sys.modules
32: (4)                 np.pi  # trigger load of numpy
33: (4)                 assert type(np) == types.ModuleType
34: (4)                 assert "numpy" in sys.modules
35: (0)             def test_lazy_import_nonbuiltins():
36: (4)                 sp = lazy._lazy_import("scipy")
37: (4)                 np = lazy._lazy_import("numpy")
38: (4)                 if isinstance(sp, lazy.DelayedImportErrorModule):
39: (8)                     try:
40: (12)                        sp.special.erf
41: (12)                        assert False
42: (8)                     except ModuleNotFoundError:
43: (12)                        pass
44: (4)                 elif isinstance(np, lazy.DelayedImportErrorModule):
45: (8)                     try:
46: (12)                        np.sin(np.pi)
47: (12)                        assert False
48: (8)                     except ModuleNotFoundError:
49: (12)                        pass
50: (4)                 else:
51: (8)                     assert sp.special.erf(np.pi) == pytest.approx(1, 1e-4)
52: (0)             def test_lazy_attach():
53: (4)                 name = "mymod"
54: (4)                 submods = ["mysubmodule", "anothersubmodule"]
55: (4)                 myall = {"not_real_submod": ["some_var_or_func"]}
56: (4)                 locls = {
57: (8)                     "attach": lazy.attach,
58: (8)                     "name": name,
```

```
59: (8)                     "submods": submods,
60: (8)                     "myall": myall,
61: (4)                 }
62: (4)                 s = "__getattr__, __lazy_dir__, __all__ = attach(name, submods, myall)"
63: (4)                 exec(s, {}, locls)
64: (4)                 expected = {
65: (8)                     "attach": lazy.attach,
66: (8)                     "name": name,
67: (8)                     "submods": submods,
68: (8)                     "myall": myall,
69: (8)                     "__getattr__": None,
70: (8)                     "__lazy_dir__": None,
71: (8)                     "__all__": None,
72: (4)                 }
73: (4)                 assert locls.keys() == expected.keys()
74: (4)                 for k, v in expected.items():
75: (8)                     if v is not None:
76: (12)                        assert locls[k] == v
```

File 545 - test_relabel.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.generators.classic import empty_graph
4: (0)              from networkx.utils import edges_equal, nodes_equal
5: (0)              class TestRelabel:
6: (4)                  def test_convert_node_labels_to_integers(self):
7: (8)                      G = empty_graph()
8: (8)                      H = nx.convert_node_labels_to_integers(G, 100)
9: (8)                      assert list(H.nodes()) == []
10: (8)                     assert list(H.edges()) == []
11: (8)                     for opt in ["default", "sorted", "increasing degree", "decreasing
degree"]:
12: (12)                        G = empty_graph()
13: (12)                        H = nx.convert_node_labels_to_integers(G, 100, ordering=opt)
14: (12)                        assert list(H.nodes()) == []
15: (12)                        assert list(H.edges()) == []
16: (8)                     G = empty_graph()
17: (8)                     G.add_edges_from([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
18: (8)                     H = nx.convert_node_labels_to_integers(G)
19: (8)                     degH = (d for n, d in H.degree())
20: (8)                     degG = (d for n, d in G.degree())
21: (8)                     assert sorted(degH) == sorted(degG)
22: (8)                     H = nx.convert_node_labels_to_integers(G, 1000)
23: (8)                     degH = (d for n, d in H.degree())
24: (8)                     degG = (d for n, d in G.degree())
25: (8)                     assert sorted(degH) == sorted(degG)
26: (8)                     assert nodes_equal(H.nodes(), [1000, 1001, 1002, 1003])
27: (8)                     H = nx.convert_node_labels_to_integers(G, ordering="increasing
degree")
28: (8)                     degH = (d for n, d in H.degree())
29: (8)                     degG = (d for n, d in G.degree())
30: (8)                     assert sorted(degH) == sorted(degG)
31: (8)                     assert H.degree(0) == 1
32: (8)                     assert H.degree(1) == 2
33: (8)                     assert H.degree(2) == 2
34: (8)                     assert H.degree(3) == 3
35: (8)                     H = nx.convert_node_labels_to_integers(G, ordering="decreasing
degree")
36: (8)                     degH = (d for n, d in H.degree())
37: (8)                     degG = (d for n, d in G.degree())
38: (8)                     assert sorted(degH) == sorted(degG)
39: (8)                     assert H.degree(0) == 3
40: (8)                     assert H.degree(1) == 2
41: (8)                     assert H.degree(2) == 2
42: (8)                     assert H.degree(3) == 1
43: (8)                     H = nx.convert_node_labels_to_integers(
```

```
 44: (12)                        G, ordering="increasing degree", label_attribute="label"
 45: (8)                     )
 46: (8)                     degH = (d for n, d in H.degree())
 47: (8)                     degG = (d for n, d in G.degree())
 48: (8)                     assert sorted(degH) == sorted(degG)
 49: (8)                     assert H.degree(0) == 1
 50: (8)                     assert H.degree(1) == 2
 51: (8)                     assert H.degree(2) == 2
 52: (8)                     assert H.degree(3) == 3
 53: (8)                     assert H.nodes[3]["label"] == "C"
 54: (8)                     assert H.nodes[0]["label"] == "D"
 55: (8)                     assert H.nodes[1]["label"] == "A" or H.nodes[2]["label"] == "A"
 56: (8)                     assert H.nodes[1]["label"] == "B" or H.nodes[2]["label"] == "B"
 57: (4)                 def test_convert_to_integers2(self):
 58: (8)                     G = empty_graph()
 59: (8)                     G.add_edges_from([("C", "D"), ("A", "B"), ("A", "C"), ("B", "C")])
 60: (8)                     H = nx.convert_node_labels_to_integers(G, ordering="sorted")
 61: (8)                     degH = (d for n, d in H.degree())
 62: (8)                     degG = (d for n, d in G.degree())
 63: (8)                     assert sorted(degH) == sorted(degG)
 64: (8)                     H = nx.convert_node_labels_to_integers(
 65: (12)                        G, ordering="sorted", label_attribute="label"
 66: (8)                     )
 67: (8)                     assert H.nodes[0]["label"] == "A"
 68: (8)                     assert H.nodes[1]["label"] == "B"
 69: (8)                     assert H.nodes[2]["label"] == "C"
 70: (8)                     assert H.nodes[3]["label"] == "D"
 71: (4)                 def test_convert_to_integers_raise(self):
 72: (8)                     with pytest.raises(nx.NetworkXError):
 73: (12)                        G = nx.Graph()
 74: (12)                        H = nx.convert_node_labels_to_integers(G, ordering="increasing
age")
 75: (4)                 def test_relabel_nodes_copy(self):
 76: (8)                     G = nx.empty_graph()
 77: (8)                     G.add_edges_from([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
 78: (8)                     mapping = {"A": "aardvark", "B": "bear", "C": "cat", "D": "dog"}
 79: (8)                     H = nx.relabel_nodes(G, mapping)
 80: (8)                     assert nodes_equal(H.nodes(), ["aardvark", "bear", "cat", "dog"])
 81: (4)                 def test_relabel_nodes_function(self):
 82: (8)                     G = nx.empty_graph()
 83: (8)                     G.add_edges_from([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
 84: (8)                     def mapping(n):
 85: (12)                        return ord(n)
 86: (8)                     H = nx.relabel_nodes(G, mapping)
 87: (8)                     assert nodes_equal(H.nodes(), [65, 66, 67, 68])
 88: (4)                 def test_relabel_nodes_callable_type(self):
 89: (8)                     G = nx.path_graph(4)
 90: (8)                     H = nx.relabel_nodes(G, str)
 91: (8)                     assert nodes_equal(H.nodes, ["0", "1", "2", "3"])
 92: (4)                 @pytest.mark.parametrize("non_mc", ("0123", ["0", "1", "2", "3"]))
 93: (4)                 def test_relabel_nodes_non_mapping_or_callable(self, non_mc):
 94: (8)                     """If `mapping` is neither a Callable or a Mapping, an exception
 95: (8)                     should be raised."""
 96: (8)                     G = nx.path_graph(4)
 97: (8)                     with pytest.raises(AttributeError):
 98: (12)                        nx.relabel_nodes(G, non_mc)
 99: (4)                 def test_relabel_nodes_graph(self):
100: (8)                     G = nx.Graph([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
101: (8)                     mapping = {"A": "aardvark", "B": "bear", "C": "cat", "D": "dog"}
102: (8)                     H = nx.relabel_nodes(G, mapping)
103: (8)                     assert nodes_equal(H.nodes(), ["aardvark", "bear", "cat", "dog"])
104: (4)                 def test_relabel_nodes_orderedgraph(self):
105: (8)                     G = nx.Graph()
106: (8)                     G.add_nodes_from([1, 2, 3])
107: (8)                     G.add_edges_from([(1, 3), (2, 3)])
108: (8)                     mapping = {1: "a", 2: "b", 3: "c"}
109: (8)                     H = nx.relabel_nodes(G, mapping)
110: (8)                     assert list(H.nodes) == ["a", "b", "c"]
111: (4)                 def test_relabel_nodes_digraph(self):
```

```
(8)                 G = nx.DiGraph([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
(8)                 mapping = {"A": "aardvark", "B": "bear", "C": "cat", "D": "dog"}
(8)                 H = nx.relabel_nodes(G, mapping, copy=False)
(8)                 assert nodes_equal(H.nodes(), ["aardvark", "bear", "cat", "dog"])
(4)             def test_relabel_nodes_multigraph(self):
(8)                 G = nx.MultiGraph([("a", "b"), ("a", "b")])
(8)                 mapping = {"a": "aardvark", "b": "bear"}
(8)                 G = nx.relabel_nodes(G, mapping, copy=False)
(8)                 assert nodes_equal(G.nodes(), ["aardvark", "bear"])
(8)                 assert edges_equal(G.edges(), [("aardvark", "bear"), ("aardvark",
"bear")])
(4)             def test_relabel_nodes_multidigraph(self):
(8)                 G = nx.MultiDiGraph([("a", "b"), ("a", "b")])
(8)                 mapping = {"a": "aardvark", "b": "bear"}
(8)                 G = nx.relabel_nodes(G, mapping, copy=False)
(8)                 assert nodes_equal(G.nodes(), ["aardvark", "bear"])
(8)                 assert edges_equal(G.edges(), [("aardvark", "bear"), ("aardvark",
"bear")])
(4)             def test_relabel_isolated_nodes_to_same(self):
(8)                 G = nx.Graph()
(8)                 G.add_nodes_from(range(4))
(8)                 mapping = {1: 1}
(8)                 H = nx.relabel_nodes(G, mapping, copy=False)
(8)                 assert nodes_equal(H.nodes(), list(range(4)))
(4)             def test_relabel_nodes_missing(self):
(8)                 G = nx.Graph([("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")])
(8)                 mapping = {0: "aardvark"}
(8)                 H = nx.relabel_nodes(G, mapping, copy=True)
(8)                 assert nodes_equal(H.nodes, G.nodes)
(8)                 GG = G.copy()
(8)                 nx.relabel_nodes(G, mapping, copy=False)
(8)                 assert nodes_equal(G.nodes, GG.nodes)
(4)             def test_relabel_copy_name(self):
(8)                 G = nx.Graph()
(8)                 H = nx.relabel_nodes(G, {}, copy=True)
(8)                 assert H.graph == G.graph
(8)                 H = nx.relabel_nodes(G, {}, copy=False)
(8)                 assert H.graph == G.graph
(8)                 G.name = "first"
(8)                 H = nx.relabel_nodes(G, {}, copy=True)
(8)                 assert H.graph == G.graph
(8)                 H = nx.relabel_nodes(G, {}, copy=False)
(8)                 assert H.graph == G.graph
(4)             def test_relabel_toposort(self):
(8)                 K5 = nx.complete_graph(4)
(8)                 G = nx.complete_graph(4)
(8)                 G = nx.relabel_nodes(G, {i: i + 1 for i in range(4)}, copy=False)
(8)                 assert nx.is_isomorphic(K5, G)
(8)                 G = nx.complete_graph(4)
(8)                 G = nx.relabel_nodes(G, {i: i - 1 for i in range(4)}, copy=False)
(8)                 assert nx.is_isomorphic(K5, G)
(4)             def test_relabel_selfloop(self):
(8)                 G = nx.DiGraph([(1, 1), (1, 2), (2, 3)])
(8)                 G = nx.relabel_nodes(G, {1: "One", 2: "Two", 3: "Three"}, copy=False)
(8)                 assert nodes_equal(G.nodes(), ["One", "Three", "Two"])
(8)                 G = nx.MultiDiGraph([(1, 1), (1, 2), (2, 3)])
(8)                 G = nx.relabel_nodes(G, {1: "One", 2: "Two", 3: "Three"}, copy=False)
(8)                 assert nodes_equal(G.nodes(), ["One", "Three", "Two"])
(8)                 G = nx.MultiDiGraph([(1, 1)])
(8)                 G = nx.relabel_nodes(G, {1: 0}, copy=False)
(8)                 assert nodes_equal(G.nodes(), [0])
(4)             def test_relabel_multidigraph_inout_merge_nodes(self):
(8)                 for MG in (nx.MultiGraph, nx.MultiDiGraph):
(12)                    for cc in (True, False):
(16)                        G = MG([(0, 4), (1, 4), (4, 2), (4, 3)])
(16)                        G[0][4][0]["value"] = "a"
(16)                        G[1][4][0]["value"] = "b"
(16)                        G[4][2][0]["value"] = "c"
(16)                        G[4][3][0]["value"] = "d"
```

```
(16)                            G.add_edge(0, 4, key="x", value="e")
(16)                            G.add_edge(4, 3, key="x", value="f")
(16)                            mapping = {0: 9, 1: 9, 2: 9, 3: 9}
(16)                            H = nx.relabel_nodes(G, mapping, copy=cc)
(16)                            assert {"value": "a"} in H[9][4].values()
(16)                            assert {"value": "b"} in H[9][4].values()
(16)                            assert {"value": "c"} in H[4][9].values()
(16)                            assert len(H[4][9]) == 3 if G.is_directed() else 6
(16)                            assert {"value": "d"} in H[4][9].values()
(16)                            assert {"value": "e"} in H[9][4].values()
(16)                            assert {"value": "f"} in H[4][9].values()
(16)                            assert len(H[9][4]) == 3 if G.is_directed() else 6
(4)                 def test_relabel_multigraph_merge_inplace(self):
(8)                     G = nx.MultiGraph([(0, 1), (0, 2), (0, 3), (0, 1), (0, 2), (0, 3)])
(8)                     G[0][1][0]["value"] = "a"
(8)                     G[0][2][0]["value"] = "b"
(8)                     G[0][3][0]["value"] = "c"
(8)                     mapping = {1: 4, 2: 4, 3: 4}
(8)                     nx.relabel_nodes(G, mapping, copy=False)
(8)                     assert {"value": "a"} in G[0][4].values()
(8)                     assert {"value": "b"} in G[0][4].values()
(8)                     assert {"value": "c"} in G[0][4].values()
(4)                 def test_relabel_multidigraph_merge_inplace(self):
(8)                     G = nx.MultiDiGraph([(0, 1), (0, 2), (0, 3)])
(8)                     G[0][1][0]["value"] = "a"
(8)                     G[0][2][0]["value"] = "b"
(8)                     G[0][3][0]["value"] = "c"
(8)                     mapping = {1: 4, 2: 4, 3: 4}
(8)                     nx.relabel_nodes(G, mapping, copy=False)
(8)                     assert {"value": "a"} in G[0][4].values()
(8)                     assert {"value": "b"} in G[0][4].values()
(8)                     assert {"value": "c"} in G[0][4].values()
(4)                 def test_relabel_multidigraph_inout_copy(self):
(8)                     G = nx.MultiDiGraph([(0, 4), (1, 4), (4, 2), (4, 3)])
(8)                     G[0][4][0]["value"] = "a"
(8)                     G[1][4][0]["value"] = "b"
(8)                     G[4][2][0]["value"] = "c"
(8)                     G[4][3][0]["value"] = "d"
(8)                     G.add_edge(0, 4, key="x", value="e")
(8)                     G.add_edge(4, 3, key="x", value="f")
(8)                     mapping = {0: 9, 1: 9, 2: 9, 3: 9}
(8)                     H = nx.relabel_nodes(G, mapping, copy=True)
(8)                     assert {"value": "a"} in H[9][4].values()
(8)                     assert {"value": "b"} in H[9][4].values()
(8)                     assert {"value": "c"} in H[4][9].values()
(8)                     assert len(H[4][9]) == 3
(8)                     assert {"value": "d"} in H[4][9].values()
(8)                     assert {"value": "e"} in H[9][4].values()
(8)                     assert {"value": "f"} in H[4][9].values()
(8)                     assert len(H[9][4]) == 3
(4)                 def test_relabel_multigraph_merge_copy(self):
(8)                     G = nx.MultiGraph([(0, 1), (0, 2), (0, 3)])
(8)                     G[0][1][0]["value"] = "a"
(8)                     G[0][2][0]["value"] = "b"
(8)                     G[0][3][0]["value"] = "c"
(8)                     mapping = {1: 4, 2: 4, 3: 4}
(8)                     H = nx.relabel_nodes(G, mapping, copy=True)
(8)                     assert {"value": "a"} in H[0][4].values()
(8)                     assert {"value": "b"} in H[0][4].values()
(8)                     assert {"value": "c"} in H[0][4].values()
(4)                 def test_relabel_multidigraph_merge_copy(self):
(8)                     G = nx.MultiDiGraph([(0, 1), (0, 2), (0, 3)])
(8)                     G[0][1][0]["value"] = "a"
(8)                     G[0][2][0]["value"] = "b"
(8)                     G[0][3][0]["value"] = "c"
(8)                     mapping = {1: 4, 2: 4, 3: 4}
(8)                     H = nx.relabel_nodes(G, mapping, copy=True)
(8)                     assert {"value": "a"} in H[0][4].values()
(8)                     assert {"value": "b"} in H[0][4].values()
```

```
248: (8)                    assert {"value": "c"} in H[0][4].values()
249: (4)                def test_relabel_multigraph_nonnumeric_key(self):
250: (8)                    for MG in (nx.MultiGraph, nx.MultiDiGraph):
251: (12)                       for cc in (True, False):
252: (16)                           G = nx.MultiGraph()
253: (16)                           G.add_edge(0, 1, key="I", value="a")
254: (16)                           G.add_edge(0, 2, key="II", value="b")
255: (16)                           G.add_edge(0, 3, key="II", value="c")
256: (16)                           mapping = {1: 4, 2: 4, 3: 4}
257: (16)                           nx.relabel_nodes(G, mapping, copy=False)
258: (16)                           assert {"value": "a"} in G[0][4].values()
259: (16)                           assert {"value": "b"} in G[0][4].values()
260: (16)                           assert {"value": "c"} in G[0][4].values()
261: (16)                           assert 0 in G[0][4]
262: (16)                           assert "I" in G[0][4]
263: (16)                           assert "II" in G[0][4]
264: (4)                def test_relabel_circular(self):
265: (8)                    G = nx.path_graph(3)
266: (8)                    mapping = {0: 1, 1: 0}
267: (8)                    H = nx.relabel_nodes(G, mapping, copy=True)
268: (8)                    with pytest.raises(nx.NetworkXUnfeasible):
269: (12)                       H = nx.relabel_nodes(G, mapping, copy=False)
270: (4)                def test_relabel_preserve_node_order_full_mapping_with_copy_true(self):
271: (8)                    G = nx.path_graph(3)
272: (8)                    original_order = list(G.nodes())
273: (8)                    mapping = {2: "a", 1: "b", 0: "c"}  # dictionary keys out of order on
purpose
274: (8)                    H = nx.relabel_nodes(G, mapping, copy=True)
275: (8)                    new_order = list(H.nodes())
276: (8)                    assert [mapping.get(i, i) for i in original_order] == new_order
277: (4)                def test_relabel_preserve_node_order_full_mapping_with_copy_false(self):
278: (8)                    G = nx.path_graph(3)
279: (8)                    original_order = list(G)
280: (8)                    mapping = {2: "a", 1: "b", 0: "c"}  # dictionary keys out of order on
purpose
281: (8)                    H = nx.relabel_nodes(G, mapping, copy=False)
282: (8)                    new_order = list(H)
283: (8)                    assert [mapping.get(i, i) for i in original_order] == new_order
284: (4)                def test_relabel_preserve_node_order_partial_mapping_with_copy_true(self):
285: (8)                    G = nx.path_graph(3)
286: (8)                    original_order = list(G)
287: (8)                    mapping = {1: "a", 0: "b"}  # partial mapping and keys out of order on
purpose
288: (8)                    H = nx.relabel_nodes(G, mapping, copy=True)
289: (8)                    new_order = list(H)
290: (8)                    assert [mapping.get(i, i) for i in original_order] == new_order
291: (4)                def
test_relabel_preserve_node_order_partial_mapping_with_copy_false(self):
292: (8)                    G = nx.path_graph(3)
293: (8)                    original_order = list(G)
294: (8)                    mapping = {1: "a", 0: "b"}  # partial mapping and keys out of order on
purpose
295: (8)                    H = nx.relabel_nodes(G, mapping, copy=False)
296: (8)                    new_order = list(H)
297: (8)                    assert [mapping.get(i, i) for i in original_order] != new_order
```

File 546 - __init__.py:

```
1: (0)              from networkx.utils.misc import *
2: (0)              from networkx.utils.decorators import *
3: (0)              from networkx.utils.random_sequence import *
4: (0)              from networkx.utils.union_find import *
5: (0)              from networkx.utils.rcm import *
6: (0)              from networkx.utils.heaps import *
7: (0)              from networkx.utils.backends import *
8: (0)              from networkx.utils.configs import *
```

```
----------------------------------------

File 547 - backends.py:

1: (0)              """
2: (0)              NetworkX utilizes a plugin-dispatch architecture, which means we can plug in
and
3: (0)              out of backends with minimal code changes. A valid NetworkX backend specifies
4: (0)              `entry points <https://packaging.python.org/en/latest/specifications/entry-
points>`_,
5: (0)              named ``networkx.backends`` and an optional ``networkx.backend_info`` when it
is
6: (0)              installed (not imported). This allows NetworkX to dispatch (redirect) function
calls
7: (0)              to the backend so the execution flows to the designated backend
8: (0)              implementation, similar to how plugging a charger into a socket redirects the
9: (0)              electricity to your phone. This design enhances flexibility and integration,
making
10: (0)             NetworkX more adaptable and efficient.
11: (0)             There are three main ways to use a backend after the package is installed.
12: (0)             You can set environment variables and run the exact same code you run for
13: (0)             NetworkX. You can use a keyword argument ``backend=...`` with the NetworkX
14: (0)             function. Or, you can convert the NetworkX Graph to a backend graph type and
15: (0)             call a NetworkX function supported by that backend. Environment variables
16: (0)             and backend keywords automatically convert your NetworkX Graph to the
17: (0)             backend type. Manually converting it yourself allows you to use that same
18: (0)             backend graph for more than one function call, reducing conversion time.
19: (0)             For example, you can set an environment variable before starting python to
request
20: (0)             all dispatchable functions automatically dispatch to the given backend::
21: (4)                 bash> NETWORKX_AUTOMATIC_BACKENDS=cugraph python my_networkx_script.py
22: (0)             or you can specify the backend as a kwarg::
23: (4)                 nx.betweenness_centrality(G, k=10, backend="parallel")
24: (0)             or you can convert the NetworkX Graph object ``G`` into a Graph-like
25: (0)             object specific to the backend and then pass that in the NetworkX function::
26: (4)                 H = nx_parallel.ParallelGraph(G)
27: (4)                 nx.betweenness_centrality(H, k=10)
28: (0)             How it works: You might have seen the ``@nx._dispatchable`` decorator on
29: (0)             many of the NetworkX functions in the codebase. It decorates the function
30: (0)             with code that redirects execution to the function's backend implementation.
31: (0)             The code also manages any ``backend_kwargs`` you provide to the backend
32: (0)             version of the function. The code looks for the environment variable or
33: (0)             a ``backend`` keyword argument and if found, converts the input NetworkX
34: (0)             graph to the backend format before calling the backend's version of the
35: (0)             function. If no environment variable or backend keyword are found, the
36: (0)             dispatching code checks the input graph object for an attribute
37: (0)             called ``__networkx_backend__`` which tells it which backend provides this
38: (0)             graph type. That backend's version of the function is then called.
39: (0)             The backend system relies on Python ``entry_point`` system to signal
40: (0)             NetworkX that a backend is installed (even if not imported yet). Thus no
41: (0)             code needs to be changed between running with NetworkX and running with
42: (0)             a backend to NetworkX. The attribute ``__networkx_backend__`` holds a
43: (0)             string with the name of the ``entry_point``. If none of these options
44: (0)             are being used, the decorator code simply calls the NetworkX function
45: (0)             on the NetworkX graph as usual.
46: (0)             The NetworkX library does not need to know that a backend exists for it
47: (0)             to work. So long as the backend package creates the entry_point, and
48: (0)             provides the correct interface, it will be called when the user requests
49: (0)             it using one of the three approaches described above. Some backends have
50: (0)             been working with the NetworkX developers to ensure smooth operation.
51: (0)             They are the following::
52: (0)             - `graphblas <https://github.com/python-graphblas/graphblas-algorithms>`_
53: (0)             - `cugraph <https://github.com/rapidsai/cugraph/tree/branch-24.04/python/nx-
cugraph>`_
54: (0)             - `parallel <https://github.com/networkx/nx-parallel>`_
55: (0)             - ``loopback`` is for testing purposes only and is not a real backend.
56: (0)             Note that the ``backend_name`` is e.g. ``parallel``, the package installed
57: (0)             is ``nx-parallel``, and we use ``nx_parallel`` while importing the package.
58: (0)             Creating a Custom backend
```

```
59: (0)             -------------------------
60: (0)             1.  To be a valid backend that is discoverable by NetworkX, your package must
61: (4)                 register an `entry-point
<https://packaging.python.org/en/latest/specifications/entry-points/#entry-points>`_
62: (4)                 ``networkx.backends`` in the package's metadata, with a `key pointing to
your
63: (4)                 dispatch object <https://packaging.python.org/en/latest/guides/creating-
and-discovering-plugins/#using-package-metadata>`_ .
64: (4)                 For example, if you are using ``setuptools`` to manage your backend
package,
65: (4)                 you can `add the following to your pyproject.toml file
<https://setuptools.pypa.io/en/latest/userguide/entry_point.html>`_::
66: (8)                     [project.entry-points."networkx.backends"]
67: (8)                     backend_name = "your_dispatcher_class"
68: (4)                 You can also add the ``backend_info`` entry-point. It points towards the
``get_info``
69: (4)                 function that returns all the backend information, which is then used to
build the
70: (4)                 "Additional Backend Implementation" box at the end of algorithm's
documentation
71: (4)                 page (e.g. `nx-cugraph's get_info function
<https://github.com/rapidsai/cugraph/blob/branch-24.04/python/nx-
cugraph/_nx_cugraph/__init__.py>`_)::
72: (8)                     [project.entry-points."networkx.backend_info"]
73: (8)                     backend_name = "your_get_info_function"
74: (4)                 Note that this would only work if your backend is a trusted backend of
NetworkX,
75: (4)                 and is present in the `.circleci/config.yml` and
76: (4)                 `.github/workflows/deploy-docs.yml` files in the NetworkX repository.
77: (0)             2.  The backend must create an ``nx.Graph``-like object which contains an
attribute
78: (4)                 ``__networkx_backend__`` with a value of the entry point name::
79: (8)                     class BackendGraph:
80: (12)                        __networkx_backend__ = "backend_name"
81: (12)                        ...
82: (0)             Testing the Custom backend
83: (0)             -------------------------
84: (0)             To test your custom backend, you can run the NetworkX test suite with your
backend.
85: (0)             This also ensures that the custom backend is compatible with NetworkX's API.
86: (0)             Testing Environment Setup
87: (0)             ~~~~~~~~~~~~~~~~~~~~~~~~
88: (0)             To enable automatic testing with your custom backend, follow these steps:
89: (0)             1. Set Backend Environment Variables:
90: (4)                 - ``NETWORKX_TEST_BACKEND`` : Setting this to your registered backend key
will let
91: (6)                   the NetworkX's dispatch machinery automatically convert a regular
NetworkX
92: (6)                   ``Graph``, ``DiGraph``, ``MultiGraph``, etc. to their backend
equivalents, using
93: (6)                   ``your_dispatcher_class.convert_from_nx(G, ...)`` function.
94: (4)                 - ``NETWORKX_FALLBACK_TO_NX`` (default=False) : Setting this variable to
`True` will
95: (6)                   instruct tests to use a NetworkX ``Graph`` for algorithms not
implemented by your
96: (6)                   custom backend. Setting this to `False` will only run the tests for
algorithms
97: (6)                   implemented by your custom backend and tests for other algorithms will
``xfail``.
98: (0)             2. Defining ``convert_from_nx`` and ``convert_to_nx`` methods:
99: (4)                 The arguments to ``convert_from_nx`` are:
100: (4)                - ``G`` : NetworkX Graph
101: (4)                - ``edge_attrs`` : dict, optional
102: (8)                    Dictionary mapping edge attributes to default values if missing in
``G``.
103: (8)                    If None, then no edge attributes will be converted and default may be
1.
104: (4)                - ``node_attrs``: dict, optional
105: (8)                    Dictionary mapping node attributes to default values if missing in
```

```
``G``.
106: (8)                If None, then no node attributes will be converted.
107: (4)            - ``preserve_edge_attrs`` : bool
108: (8)                Whether to preserve all edge attributes.
109: (4)            - ``preserve_node_attrs`` : bool
110: (8)                Whether to preserve all node attributes.
111: (4)            - ``preserve_graph_attrs`` : bool
112: (8)                Whether to preserve all graph attributes.
113: (4)            - ``preserve_all_attrs`` : bool
114: (8)                Whether to preserve all graph, node, and edge attributes.
115: (4)            - ``name`` : str
116: (8)                The name of the algorithm.
117: (4)            - ``graph_name`` : str
118: (8)                The name of the graph argument being converted.
119: (0)            Running Tests
120: (0)            ~~~~~~~~~~~~~
121: (0)            You can invoke NetworkX tests for your custom backend with the following
commands::
122: (4)                NETWORKX_TEST_BACKEND=<backend_name>
123: (4)                NETWORKX_FALLBACK_TO_NX=True # or False
124: (4)                pytest --pyargs networkx
125: (0)            Conversions while running tests :
126: (0)            - Convert NetworkX graphs using ``<your_dispatcher_class>.convert_from_nx(G,
...)`` into
127: (2)              the backend graph.
128: (0)            - Pass the backend graph objects to the backend implementation of the
algorithm.
129: (0)            - Convert the result back to a form expected by NetworkX tests using
130: (2)              ``<your_dispatcher_class>.convert_to_nx(result, ...)``.
131: (0)            Notes
132: (0)            ~~~~~
133: (0)            -   Dispatchable algorithms that are not implemented by the backend
134: (4)                will cause a ``pytest.xfail``, giving some indication that not all
135: (4)                tests are running, while avoiding causing an explicit failure.
136: (0)            -   If a backend only partially implements some algorithms, it can define
137: (4)                a ``can_run(name, args, kwargs)`` function that returns True or False
138: (4)                indicating whether it can run the algorithm with the given arguments.
139: (4)                It may also return a string indicating why the algorithm can't be run;
140: (4)                this string may be used in the future to give helpful info to the user.
141: (0)            -   A backend may also define ``should_run(name, args, kwargs)`` that is
similar
142: (4)                to ``can_run``, but answers whether the backend *should* be run
(converting
143: (4)                if necessary). Like ``can_run``, it receives the original arguments so it
144: (4)                can decide whether it should be run by inspecting the arguments.
``can_run``
145: (4)                runs before ``should_run``, so ``should_run`` may assume ``can_run`` is
True.
146: (4)                If not implemented by the backend, ``can_run`` and ``should_run`` are
147: (4)                assumed to always return True if the backend implements the algorithm.
148: (0)            -   A special ``on_start_tests(items)`` function may be defined by the
backend.
149: (4)                It will be called with the list of NetworkX tests discovered. Each item
150: (4)                is a test object that can be marked as xfail if the backend does not
support
151: (4)                the test using ``item.add_marker(pytest.mark.xfail(reason=...))``.
152: (0)            -   A backend graph instance may have a ``G.__networkx_cache__`` dict to
enable
153: (4)                caching, and care should be taken to clear the cache when appropriate.
154: (0)            """
155: (0)            import inspect
156: (0)            import itertools
157: (0)            import os
158: (0)            import warnings
159: (0)            from functools import partial
160: (0)            from importlib.metadata import entry_points
161: (0)            import networkx as nx
162: (0)            from .decorators import argmap
163: (0)            __all__ = ["_dispatchable"]
```

```
164: (0)            def _do_nothing():
165: (4)                """This does nothing at all, yet it helps turn `_dispatchable` into
functions."""
166: (0)            def _get_backends(group, *, load_and_call=False):
167: (4)                """
168: (4)                Retrieve NetworkX ``backends`` and ``backend_info`` from the entry points.
169: (4)                Parameters
170: (4)                -----------
171: (4)                group : str
172: (8)                    The entry_point to be retrieved.
173: (4)                load_and_call : bool, optional
174: (8)                    If True, load and call the backend. Defaults to False.
175: (4)                Returns
176: (4)                --------
177: (4)                dict
178: (8)                    A dictionary mapping backend names to their respective backend
objects.
179: (4)                Notes
180: (4)                ------
181: (4)                If a backend is defined more than once, a warning is issued.
182: (4)                The `nx-loopback` backend is removed if it exists, as it is only available
during testing.
183: (4)                A warning is displayed if an error occurs while loading a backend.
184: (4)                """
185: (4)                items = entry_points(group=group)
186: (4)                rv = {}
187: (4)                for ep in items:
188: (8)                    if ep.name in rv:
189: (12)                       warnings.warn(
190: (16)                           f"networkx backend defined more than once: {ep.name}",
191: (16)                           RuntimeWarning,
192: (16)                           stacklevel=2,
193: (12)                       )
194: (8)                    elif load_and_call:
195: (12)                       try:
196: (16)                           rv[ep.name] = ep.load()()
197: (12)                       except Exception as exc:
198: (16)                           warnings.warn(
199: (20)                               f"Error encountered when loading info for backend
{ep.name}: {exc}",
200: (20)                               RuntimeWarning,
201: (20)                               stacklevel=2,
202: (16)                           )
203: (8)                    else:
204: (12)                       rv[ep.name] = ep
205: (4)                rv.pop("nx-loopback", None)
206: (4)                return rv
207: (0)            backends = _get_backends("networkx.backends")
208: (0)            backend_info = _get_backends("networkx.backend_info", load_and_call=True)
209: (0)            from .configs import Config, config
210: (0)            config.backend_priority = [
211: (4)                x.strip()
212: (4)                for x in os.environ.get(
213: (8)                    "NETWORKX_BACKEND_PRIORITY",
214: (8)                    os.environ.get("NETWORKX_AUTOMATIC_BACKENDS", ""),
215: (4)                ).split(",")
216: (4)                if x.strip()
217: (0)            ]
218: (0)            config.backends = Config(
219: (4)                **{
220: (8)                    backend: (
221: (12)                       cfg if isinstance(cfg := info["default_config"], Config) else
Config(**cfg)
222: (8)                    )
223: (8)                    if "default_config" in info
224: (8)                    else Config()
225: (8)                    for backend, info in backend_info.items()
226: (4)                }
227: (0)            )
```

```
228: (0)            type(config.backends).__doc__ = "All installed NetworkX backends and their
configs."
229: (0)            _loaded_backends = {}  # type: ignore[var-annotated]
230: (0)            def _always_run(name, args, kwargs):
231: (4)                return True
232: (0)            def _load_backend(backend_name):
233: (4)                if backend_name in _loaded_backends:
234: (8)                    return _loaded_backends[backend_name]
235: (4)                rv = _loaded_backends[backend_name] = backends[backend_name].load()
236: (4)                if not hasattr(rv, "can_run"):
237: (8)                    rv.can_run = _always_run
238: (4)                if not hasattr(rv, "should_run"):
239: (8)                    rv.should_run = _always_run
240: (4)                return rv
241: (0)            _registered_algorithms = {}
242: (0)            class _dispatchable:
243: (4)                """Allow any of the following decorator forms:
244: (4)                - @_dispatchable
245: (4)                - @_dispatchable()
246: (4)                - @_dispatchable(name="override_name")
247: (4)                - @_dispatchable(graphs="graph")
248: (4)                - @_dispatchable(edge_attrs="weight")
249: (4)                - @_dispatchable(graphs={"G": 0, "H": 1}, edge_attrs={"weight":
"default"})
250: (4)                These class attributes are currently used to allow backends to run
networkx tests.
251: (4)                For example: `PYTHONPATH=. pytest --backend graphblas --fallback-to-nx`
252: (4)                Future work: add configuration to control these.
253: (4)                """
254: (4)                _is_testing = False
255: (4)                _fallback_to_nx = (
256: (8)                    os.environ.get("NETWORKX_FALLBACK_TO_NX", "true").strip().lower() ==
"true"
257: (4)                )
258: (4)                def __new__(
259: (8)                    cls,
260: (8)                    func=None,
261: (8)                    *,
262: (8)                    name=None,
263: (8)                    graphs="G",
264: (8)                    edge_attrs=None,
265: (8)                    node_attrs=None,
266: (8)                    preserve_edge_attrs=False,
267: (8)                    preserve_node_attrs=False,
268: (8)                    preserve_graph_attrs=False,
269: (8)                    preserve_all_attrs=False,
270: (8)                    mutates_input=False,
271: (8)                    returns_graph=False,
272: (4)                ):
273: (8)                    """A decorator that makes certain input graph types dispatch to
``func``'s
274: (8)                    backend implementation.
275: (8)                    Usage can be any of the following decorator forms:
276: (8)                    - @_dispatchable
277: (8)                    - @_dispatchable()
278: (8)                    - @_dispatchable(name="override_name")
279: (8)                    - @_dispatchable(graphs="graph_var_name")
280: (8)                    - @_dispatchable(edge_attrs="weight")
281: (8)                    - @_dispatchable(graphs={"G": 0, "H": 1}, edge_attrs={"weight":
"default"})
282: (8)                    with 0 and 1 giving the position in the signature function for graph
objects.
283: (8)                    When edge_attrs is a dict, keys are keyword names and values are
defaults.
284: (8)                    The class attributes are used to allow backends to run networkx tests.
285: (8)                    For example: `PYTHONPATH=. pytest --backend graphblas --fallback-to-
nx`
286: (8)                    Future work: add configuration to control these.
287: (8)                    Parameters
```

```
                    ----------
                    func : callable, optional
                        The function to be decorated. If ``func`` is not provided, returns
a
                        partial object that can be used to decorate a function later. If
``func``
                        is provided, returns a new callable object that dispatches to a
backend
                        algorithm based on input graph types.
                    name : str, optional
                        The name of the algorithm to use for dispatching. If not provided,
                        the name of ``func`` will be used. ``name`` is useful to avoid
name
                        conflicts, as all dispatched algorithms live in a single
namespace.
                        For example, ``tournament.is_strongly_connected`` had a name
conflict
                        with the standard ``nx.is_strongly_connected``, so we used
                        ``@_dispatchable(name="tournament_is_strongly_connected")``.
                    graphs : str or dict or None, default "G"
                        If a string, the parameter name of the graph, which must be the
first
                        argument of the wrapped function. If more than one graph is
required
                        for the algorithm (or if the graph is not the first argument),
provide
                        a dict of parameter name to argument position for each graph
argument.
                        For example, ``@_dispatchable(graphs={"G": 0, "auxiliary?": 4})``
                        indicates the 0th parameter ``G`` of the function is a required
graph,
                        and the 4th parameter ``auxiliary`` is an optional graph.
                        To indicate an argument is a list of graphs, do e.g. ``"
[graphs]"``.
                        Use ``graphs=None`` if *no* arguments are NetworkX graphs such as
for
                        graph generators, readers, and conversion functions.
                    edge_attrs : str or dict, optional
                        ``edge_attrs`` holds information about edge attribute arguments
                        and default values for those edge attributes.
                        If a string, ``edge_attrs`` holds the function argument name that
                        indicates a single edge attribute to include in the converted
graph.
                        The default value for this attribute is 1. To indicate that an
argument
                        is a list of attributes (all with default value 1), use e.g. ``"
[attrs]"``.
                        If a dict, ``edge_attrs`` holds a dict keyed by argument names,
with
                        values that are either the default value or, if a string, the
argument
                        name that indicates the default value.
                    node_attrs : str or dict, optional
                        Like ``edge_attrs``, but for node attributes.
                    preserve_edge_attrs : bool or str or dict, optional
                        For bool, whether to preserve all edge attributes.
                        For str, the parameter name that may indicate (with ``True`` or a
                        callable argument) whether all edge attributes should be preserved
                        when converting.
                        For dict of ``{graph_name: {attr: default}}``, indicate pre-
determined
                        edge attributes (and defaults) to preserve for input graphs.
                    preserve_node_attrs : bool or str or dict, optional
                        Like ``preserve_edge_attrs``, but for node attributes.
                    preserve_graph_attrs : bool or set
                        For bool, whether to preserve all graph attributes.
                        For set, which input graph arguments to preserve graph attributes.
                    preserve_all_attrs : bool
                        Whether to preserve all edge, node and graph attributes.
```

```
338: (12)                        This overrides all the other preserve_*_attrs.
339: (8)                    mutates_input : bool or dict, default False
340: (12)                        For bool, whether the functions mutates an input graph argument.
341: (12)                        For dict of ``{arg_name: arg_pos}``, arguments that indicates
whether an
342: (12)                        input graph will be mutated, and ``arg_name`` may begin with
``"not "``
343: (12)                        to negate the logic (for example, this is used by ``copy=``
arguments).
344: (12)                        By default, dispatching doesn't convert input graphs to a
different
345: (12)                        backend for functions that mutate input graphs.
346: (8)                    returns_graph : bool, default False
347: (12)                        Whether the function can return or yield a graph object. By
default,
348: (12)                        dispatching doesn't convert input graphs to a different backend
for
349: (12)                        functions that return graphs.
350: (8)                    """
351: (8)                    if func is None:
352: (12)                        return partial(
353: (16)                            _dispatchable,
354: (16)                            name=name,
355: (16)                            graphs=graphs,
356: (16)                            edge_attrs=edge_attrs,
357: (16)                            node_attrs=node_attrs,
358: (16)                            preserve_edge_attrs=preserve_edge_attrs,
359: (16)                            preserve_node_attrs=preserve_node_attrs,
360: (16)                            preserve_graph_attrs=preserve_graph_attrs,
361: (16)                            preserve_all_attrs=preserve_all_attrs,
362: (16)                            mutates_input=mutates_input,
363: (16)                            returns_graph=returns_graph,
364: (12)                        )
365: (8)                    if isinstance(func, str):
366: (12)                        raise TypeError("'name' and 'graphs' must be passed by keyword")
from None
367: (8)                    if name is None:
368: (12)                        name = func.__name__
369: (8)                    self = object.__new__(cls)
370: (8)                    self.__name__ = func.__name__
371: (8)                    self.__defaults__ = func.__defaults__
372: (8)                    if func.__kwdefaults__:
373: (12)                        self.__kwdefaults__ = {**func.__kwdefaults__, "backend": None}
374: (8)                    else:
375: (12)                        self.__kwdefaults__ = {"backend": None}
376: (8)                    self.__module__ = func.__module__
377: (8)                    self.__qualname__ = func.__qualname__
378: (8)                    self.__dict__.update(func.__dict__)
379: (8)                    self.__wrapped__ = func
380: (8)                    self._orig_doc = func.__doc__
381: (8)                    self._cached_doc = None
382: (8)                    self.orig_func = func
383: (8)                    self.name = name
384: (8)                    self.edge_attrs = edge_attrs
385: (8)                    self.node_attrs = node_attrs
386: (8)                    self.preserve_edge_attrs = preserve_edge_attrs or preserve_all_attrs
387: (8)                    self.preserve_node_attrs = preserve_node_attrs or preserve_all_attrs
388: (8)                    self.preserve_graph_attrs = preserve_graph_attrs or preserve_all_attrs
389: (8)                    self.mutates_input = mutates_input
390: (8)                    self._returns_graph = returns_graph
391: (8)                    if edge_attrs is not None and not isinstance(edge_attrs, str | dict):
392: (12)                        raise TypeError(
393: (16)                            f"Bad type for edge_attrs: {type(edge_attrs)}. Expected str or
dict."
394: (12)                        ) from None
395: (8)                    if node_attrs is not None and not isinstance(node_attrs, str | dict):
396: (12)                        raise TypeError(
397: (16)                            f"Bad type for node_attrs: {type(node_attrs)}. Expected str or
dict."
```

```
398: (12)                        ) from None
399: (8)                    if not isinstance(self.preserve_edge_attrs, bool | str | dict):
400: (12)                        raise TypeError(
401: (16)                            f"Bad type for preserve_edge_attrs:
{type(self.preserve_edge_attrs)}."
402: (16)                            " Expected bool, str, or dict."
403: (12)                        ) from None
404: (8)                    if not isinstance(self.preserve_node_attrs, bool | str | dict):
405: (12)                        raise TypeError(
406: (16)                            f"Bad type for preserve_node_attrs:
{type(self.preserve_node_attrs)}."
407: (16)                            " Expected bool, str, or dict."
408: (12)                        ) from None
409: (8)                    if not isinstance(self.preserve_graph_attrs, bool | set):
410: (12)                        raise TypeError(
411: (16)                            f"Bad type for preserve_graph_attrs:
{type(self.preserve_graph_attrs)}."
412: (16)                            " Expected bool or set."
413: (12)                        ) from None
414: (8)                    if not isinstance(self.mutates_input, bool | dict):
415: (12)                        raise TypeError(
416: (16)                            f"Bad type for mutates_input: {type(self.mutates_input)}."
417: (16)                            " Expected bool or dict."
418: (12)                        ) from None
419: (8)                    if not isinstance(self._returns_graph, bool):
420: (12)                        raise TypeError(
421: (16)                            f"Bad type for returns_graph: {type(self._returns_graph)}."
422: (16)                            " Expected bool."
423: (12)                        ) from None
424: (8)                    if isinstance(graphs, str):
425: (12)                        graphs = {graphs: 0}
426: (8)                    elif graphs is None:
427: (12)                        pass
428: (8)                    elif not isinstance(graphs, dict):
429: (12)                        raise TypeError(
430: (16)                            f"Bad type for graphs: {type(graphs)}. Expected str or dict."
431: (12)                        ) from None
432: (8)                    elif len(graphs) == 0:
433: (12)                        raise KeyError("'graphs' must contain at least one variable name")
from None
434: (8)                    self.optional_graphs = set()
435: (8)                    self.list_graphs = set()
436: (8)                    if graphs is None:
437: (12)                        self.graphs = {}
438: (8)                    else:
439: (12)                        self.graphs = {
440: (16)                            self.optional_graphs.add(val := k[:-1]) or val
441: (16)                            if (last := k[-1]) == "?"
442: (16)                            else self.list_graphs.add(val := k[1:-1]) or val
443: (16)                            if last == "]"
444: (16)                            else k: v
445: (16)                            for k, v in graphs.items()
446: (12)                        }
447: (8)                    self._sig = None
448: (8)                    self.backends = {
449: (12)                        backend
450: (12)                        for backend, info in backend_info.items()
451: (12)                        if "functions" in info and name in info["functions"]
452: (8)                    }
453: (8)                    if name in _registered_algorithms:
454: (12)                        raise KeyError(
455: (16)                            f"Algorithm already exists in dispatch registry: {name}"
456: (12)                        ) from None
457: (8)                    self = argmap(_do_nothing)(self)
458: (8)                    _registered_algorithms[name] = self
459: (8)                    return self
460: (4)                @property
461: (4)                def __doc__(self):
462: (8)                    """If the cached documentation exists, it is returned.
```

```
463: (8)                    Otherwise, the documentation is generated using _make_doc() method,
464: (8)                    cached, and then returned."""
465: (8)                    if (rv := self._cached_doc) is not None:
466: (12)                       return rv
467: (8)                    rv = self._cached_doc = self._make_doc()
468: (8)                    return rv
469: (4)                @__doc__.setter
470: (4)                def __doc__(self, val):
471: (8)                    """Sets the original documentation to the given value and resets the
472: (8)                    cached documentation."""
473: (8)                    self._orig_doc = val
474: (8)                    self._cached_doc = None
475: (4)                @property
476: (4)                def __signature__(self):
477: (8)                    """Return the signature of the original function, with the addition of
478: (8)                    the `backend` and `backend_kwargs` parameters."""
479: (8)                    if self._sig is None:
480: (12)                       sig = inspect.signature(self.orig_func)
481: (12)                       if not any(
482: (16)                           p.kind == inspect.Parameter.VAR_KEYWORD for p in
sig.parameters.values()
483: (12)                       ):
484: (16)                           sig = sig.replace(
485: (20)                               parameters=[
486: (24)                                   *sig.parameters.values(),
487: (24)                                   inspect.Parameter(
488: (28)                                       "backend", inspect.Parameter.KEYWORD_ONLY,
default=None
489: (24)                                   ),
490: (24)                                   inspect.Parameter(
491: (28)                                       "backend_kwargs", inspect.Parameter.VAR_KEYWORD
492: (24)                                   ),
493: (20)                               ]
494: (16)                           )
495: (12)                       else:
496: (16)                           *parameters, var_keyword = sig.parameters.values()
497: (16)                           sig = sig.replace(
498: (20)                               parameters=[
499: (24)                                   *parameters,
500: (24)                                   inspect.Parameter(
501: (28)                                       "backend", inspect.Parameter.KEYWORD_ONLY,
default=None
502: (24)                                   ),
503: (24)                                   var_keyword,
504: (20)                               ]
505: (16)                           )
506: (12)                       self._sig = sig
507: (8)                    return self._sig
508: (4)                def __call__(self, /, *args, backend=None, **kwargs):
509: (8)                    """Returns the result of the original function, or the backend
function if
510: (8)                    the backend is specified and that backend implements `func`."""
511: (8)                    if not backends:
512: (12)                       return self.orig_func(*args, **kwargs)
513: (8)                    backend_name = backend
514: (8)                    if backend_name is not None and backend_name not in backends:
515: (12)                       raise ImportError(f"Unable to load backend: {backend_name}")
516: (8)                    graphs_resolved = {}
517: (8)                    for gname, pos in self.graphs.items():
518: (12)                       if pos < len(args):
519: (16)                           if gname in kwargs:
520: (20)                               raise TypeError(f"{self.name}() got multiple values for
{gname!r}")
521: (16)                           val = args[pos]
522: (12)                       elif gname in kwargs:
523: (16)                           val = kwargs[gname]
524: (12)                       elif gname not in self.optional_graphs:
525: (16)                           raise TypeError(
526: (20)                               f"{self.name}() missing required graph argument: {gname}"
```

```
527: (16)                            )
528: (12)                        else:
529: (16)                            continue
530: (12)                        if val is None:
531: (16)                            if gname not in self.optional_graphs:
532: (20)                                raise TypeError(
533: (24)                                    f"{self.name}() required graph argument {gname!r} is
None; must be a graph"
534: (20)                                )
535: (12)                        else:
536: (16)                            graphs_resolved[gname] = val
537: (8)                    if self.list_graphs:
538: (12)                        args = list(args)
539: (12)                        for gname in self.list_graphs & graphs_resolved.keys():
540: (16)                            val = list(graphs_resolved[gname])
541: (16)                            graphs_resolved[gname] = val
542: (16)                            if gname in kwargs:
543: (20)                                kwargs[gname] = val
544: (16)                            else:
545: (20)                                args[self.graphs[gname]] = val
546: (12)                        has_backends = any(
547: (16)                            hasattr(g, "__networkx_backend__")
548: (16)                            if gname not in self.list_graphs
549: (16)                            else any(hasattr(g2, "__networkx_backend__") for g2 in g)
550: (16)                            for gname, g in graphs_resolved.items()
551: (12)                        )
552: (12)                        if has_backends:
553: (16)                            graph_backend_names = {
554: (20)                                getattr(g, "__networkx_backend__", "networkx")
555: (20)                                for gname, g in graphs_resolved.items()
556: (20)                                if gname not in self.list_graphs
557: (16)                            }
558: (16)                            for gname in self.list_graphs & graphs_resolved.keys():
559: (20)                                graph_backend_names.update(
560: (24)                                    getattr(g, "__networkx_backend__", "networkx")
561: (24)                                    for g in graphs_resolved[gname]
562: (20)                                )
563: (8)                    else:
564: (12)                        has_backends = any(
565: (16)                            hasattr(g, "__networkx_backend__") for g in
graphs_resolved.values()
566: (12)                        )
567: (12)                        if has_backends:
568: (16)                            graph_backend_names = {
569: (20)                                getattr(g, "__networkx_backend__", "networkx")
570: (20)                                for g in graphs_resolved.values()
571: (16)                            }
572: (8)                    backend_priority = config.backend_priority
573: (8)                    if self._is_testing and backend_priority and backend_name is None:
574: (12)                        return self._convert_and_call_for_tests(
575: (16)                            backend_priority[0],
576: (16)                            args,
577: (16)                            kwargs,
578: (16)                            fallback_to_nx=self._fallback_to_nx,
579: (12)                        )
580: (8)                    if has_backends:
581: (12)                        backend_names = graph_backend_names - {"networkx"}
582: (12)                        if len(backend_names) != 1:
583: (16)                            raise TypeError(
584: (20)                                f"{self.name}() graphs must all be from the same backend,
found {backend_names}"
585: (16)                            )
586: (12)                        [graph_backend_name] = backend_names
587: (12)                        if backend_name is not None and backend_name !=
graph_backend_name:
588: (16)                            raise TypeError(
589: (20)                                f"{self.name}() is unable to convert graph from backend
{graph_backend_name!r} "
590: (20)                                f"to the specified backend {backend_name!r}."
```

```
591: (16)                            )
592: (12)                        if graph_backend_name not in backends:
593: (16)                            raise ImportError(f"Unable to load backend:
{graph_backend_name}")
594: (12)                        if (
595: (16)                            "networkx" in graph_backend_names
596: (16)                            and graph_backend_name not in backend_priority
597: (12)                        ):
598: (16)                            raise TypeError(
599: (20)                                f"Unable to convert inputs and run {self.name}. "
600: (20)                                f"{self.name}() has networkx and {graph_backend_name}
graphs, but NetworkX is not "
601: (20)                                f"configured to automatically convert graphs from networkx
to {graph_backend_name}."
602: (16)                            )
603: (12)                        backend = _load_backend(graph_backend_name)
604: (12)                        if hasattr(backend, self.name):
605: (16)                            if "networkx" in graph_backend_names:
606: (20)                                return self._convert_and_call(
607: (24)                                    graph_backend_name,
608: (24)                                    args,
609: (24)                                    kwargs,
610: (24)                                    fallback_to_nx=self._fallback_to_nx,
611: (20)                                )
612: (16)                            return getattr(backend, self.name)(*args, **kwargs)
613: (12)                        raise nx.NetworkXNotImplemented(
614: (16)                            f"'{self.name}' not implemented by {graph_backend_name}"
615: (12)                        )
616: (8)                    if backend_name is not None:
617: (12)                        return self._convert_and_call(
618: (16)                            backend_name, args, kwargs, fallback_to_nx=False
619: (12)                        )
620: (8)                    if (
621: (12)                        not self._returns_graph
622: (12)                        and (
623: (16)                            not self.mutates_input
624: (16)                            or isinstance(self.mutates_input, dict)
625: (16)                            and any(
626: (20)                                not (
627: (24)                                    args[arg_pos]
628: (24)                                    if len(args) > arg_pos
629: (24)                                    else kwargs.get(arg_name[4:], True)
630: (20)                                )
631: (20)                                if arg_name.startswith("not ")
632: (20)                                else (
633: (24)                                    args[arg_pos] if len(args) > arg_pos else
kwargs.get(arg_name)
634: (20)                                )
635: (20)                                is not None
636: (20)                                for arg_name, arg_pos in self.mutates_input.items()
637: (16)                            )
638: (12)                        )
639: (8)                    ):
640: (12)                        for backend_name in backend_priority:
641: (16)                            if self._should_backend_run(backend_name, *args, **kwargs):
642: (20)                                return self._convert_and_call(
643: (24)                                    backend_name,
644: (24)                                    args,
645: (24)                                    kwargs,
646: (24)                                    fallback_to_nx=self._fallback_to_nx,
647: (20)                                )
648: (8)                    return self.orig_func(*args, **kwargs)
649: (4)                def _can_backend_run(self, backend_name, /, *args, **kwargs):
650: (8)                    """Can the specified backend run this algorithm with these
arguments?"""
651: (8)                    backend = _load_backend(backend_name)
652: (8)                    return (
653: (12)                        hasattr(backend, self.name)
654: (12)                        and (can_run := backend.can_run(self.name, args, kwargs))
```

```
655: (12)                        and not isinstance(can_run, str)
656: (8)                    )
657: (4)                def _should_backend_run(self, backend_name, /, *args, **kwargs):
658: (8)                    """Can/should the specified backend run this algorithm with these
arguments?"""
659: (8)                    backend = _load_backend(backend_name)
660: (8)                    return (
661: (12)                        hasattr(backend, self.name)
662: (12)                        and (can_run := backend.can_run(self.name, args, kwargs))
663: (12)                        and not isinstance(can_run, str)
664: (12)                        and (should_run := backend.should_run(self.name, args, kwargs))
665: (12)                        and not isinstance(should_run, str)
666: (8)                    )
667: (4)                def _convert_arguments(self, backend_name, args, kwargs, *, use_cache):
668: (8)                    """Convert graph arguments to the specified backend.
669: (8)                    Returns
670: (8)                    -------
671: (8)                    args tuple and kwargs dict
672: (8)                    """
673: (8)                    bound = self.__signature__.bind(*args, **kwargs)
674: (8)                    bound.apply_defaults()
675: (8)                    if not self.graphs:
676: (12)                        bound_kwargs = bound.kwargs
677: (12)                        del bound_kwargs["backend"]
678: (12)                        return bound.args, bound_kwargs
679: (8)                    preserve_edge_attrs = self.preserve_edge_attrs
680: (8)                    edge_attrs = self.edge_attrs
681: (8)                    if preserve_edge_attrs is False:
682: (12)                        pass
683: (8)                    elif preserve_edge_attrs is True:
684: (12)                        edge_attrs = None
685: (8)                    elif isinstance(preserve_edge_attrs, str):
686: (12)                        if bound.arguments[preserve_edge_attrs] is True or callable(
687: (16)                            bound.arguments[preserve_edge_attrs]
688: (12)                        ):
689: (16)                            preserve_edge_attrs = True
690: (16)                            edge_attrs = None
691: (12)                        elif bound.arguments[preserve_edge_attrs] is False and (
692: (16)                            isinstance(edge_attrs, str)
693: (16)                            and edge_attrs == preserve_edge_attrs
694: (16)                            or isinstance(edge_attrs, dict)
695: (16)                            and preserve_edge_attrs in edge_attrs
696: (12)                        ):
697: (16)                            preserve_edge_attrs = False
698: (16)                            edge_attrs = None
699: (12)                        else:
700: (16)                            preserve_edge_attrs = False
701: (8)                    if edge_attrs is None:
702: (12)                        pass
703: (8)                    elif isinstance(edge_attrs, str):
704: (12)                        if edge_attrs[0] == "[":
705: (16)                            edge_attrs = {
706: (20)                                edge_attr: 1 for edge_attr in
bound.arguments[edge_attrs[1:-1]]
707: (16)                            }
708: (12)                        elif callable(bound.arguments[edge_attrs]):
709: (16)                            preserve_edge_attrs = True
710: (16)                            edge_attrs = None
711: (12)                        elif bound.arguments[edge_attrs] is not None:
712: (16)                            edge_attrs = {bound.arguments[edge_attrs]: 1}
713: (12)                        elif self.name == "to_numpy_array" and hasattr(
714: (16)                            bound.arguments["dtype"], "names"
715: (12)                        ):
716: (16)                            edge_attrs = {
717: (20)                                edge_attr: 1 for edge_attr in
bound.arguments["dtype"].names
718: (16)                            }
719: (12)                        else:
720: (16)                            edge_attrs = None
```

```
(8)                     else:
(12)                        edge_attrs = {
(16)                            edge_attr: bound.arguments.get(val, 1) if isinstance(val, str)
else val
(16)                            for key, val in edge_attrs.items()
(16)                            if (edge_attr := bound.arguments[key]) is not None
(12)                        }
(8)                     preserve_node_attrs = self.preserve_node_attrs
(8)                     node_attrs = self.node_attrs
(8)                     if preserve_node_attrs is False:
(12)                        pass
(8)                     elif preserve_node_attrs is True:
(12)                        node_attrs = None
(8)                     elif isinstance(preserve_node_attrs, str):
(12)                        if bound.arguments[preserve_node_attrs] is True or callable(
(16)                            bound.arguments[preserve_node_attrs]
(12)                        ):
(16)                            preserve_node_attrs = True
(16)                            node_attrs = None
(12)                        elif bound.arguments[preserve_node_attrs] is False and (
(16)                            isinstance(node_attrs, str)
(16)                            and node_attrs == preserve_node_attrs
(16)                            or isinstance(node_attrs, dict)
(16)                            and preserve_node_attrs in node_attrs
(12)                        ):
(16)                            preserve_node_attrs = False
(16)                            node_attrs = None
(12)                        else:
(16)                            preserve_node_attrs = False
(8)                     if node_attrs is None:
(12)                        pass
(8)                     elif isinstance(node_attrs, str):
(12)                        if node_attrs[0] == "[":
(16)                            node_attrs = {
(20)                                node_attr: None for node_attr in
bound.arguments[node_attrs[1:-1]]
(16)                            }
(12)                        elif callable(bound.arguments[node_attrs]):
(16)                            preserve_node_attrs = True
(16)                            node_attrs = None
(12)                        elif bound.arguments[node_attrs] is not None:
(16)                            node_attrs = {bound.arguments[node_attrs]: None}
(12)                        else:
(16)                            node_attrs = None
(8)                     else:
(12)                        node_attrs = {
(16)                            node_attr: bound.arguments.get(val) if isinstance(val, str)
else val
(16)                            for key, val in node_attrs.items()
(16)                            if (node_attr := bound.arguments[key]) is not None
(12)                        }
(8)                     preserve_graph_attrs = self.preserve_graph_attrs
(8)                     for gname in self.graphs:
(12)                        if gname in self.list_graphs:
(16)                            bound.arguments[gname] = [
(20)                                self._convert_graph(
(24)                                    backend_name,
(24)                                    g,
(24)                                    edge_attrs=edge_attrs,
(24)                                    node_attrs=node_attrs,
(24)                                    preserve_edge_attrs=preserve_edge_attrs,
(24)                                    preserve_node_attrs=preserve_node_attrs,
(24)                                    preserve_graph_attrs=preserve_graph_attrs,
(24)                                    graph_name=gname,
(24)                                    use_cache=use_cache,
(20)                                )
(20)                                if getattr(g, "__networkx_backend__", "networkx") ==
"networkx"
(20)                                else g
```

```
786: (20)                                for g in bound.arguments[gname]
787: (16)                            ]
788: (12)                        else:
789: (16)                            graph = bound.arguments[gname]
790: (16)                            if graph is None:
791: (20)                                if gname in self.optional_graphs:
792: (24)                                    continue
793: (20)                                raise TypeError(
794: (24)                                    f"Missing required graph argument `{gname}` in
{self.name} function"
795: (20)                                )
796: (16)                            if isinstance(preserve_edge_attrs, dict):
797: (20)                                preserve_edges = False
798: (20)                                edges = preserve_edge_attrs.get(gname, edge_attrs)
799: (16)                            else:
800: (20)                                preserve_edges = preserve_edge_attrs
801: (20)                                edges = edge_attrs
802: (16)                            if isinstance(preserve_node_attrs, dict):
803: (20)                                preserve_nodes = False
804: (20)                                nodes = preserve_node_attrs.get(gname, node_attrs)
805: (16)                            else:
806: (20)                                preserve_nodes = preserve_node_attrs
807: (20)                                nodes = node_attrs
808: (16)                            if isinstance(preserve_graph_attrs, set):
809: (20)                                preserve_graph = gname in preserve_graph_attrs
810: (16)                            else:
811: (20)                                preserve_graph = preserve_graph_attrs
812: (16)                            if getattr(graph, "__networkx_backend__", "networkx") ==
"networkx":
813: (20)                                bound.arguments[gname] = self._convert_graph(
814: (24)                                    backend_name,
815: (24)                                    graph,
816: (24)                                    edge_attrs=edges,
817: (24)                                    node_attrs=nodes,
818: (24)                                    preserve_edge_attrs=preserve_edges,
819: (24)                                    preserve_node_attrs=preserve_nodes,
820: (24)                                    preserve_graph_attrs=preserve_graph,
821: (24)                                    graph_name=gname,
822: (24)                                    use_cache=use_cache,
823: (20)                                )
824: (8)                    bound_kwargs = bound.kwargs
825: (8)                    del bound_kwargs["backend"]
826: (8)                    return bound.args, bound_kwargs
827: (4)                def _convert_graph(
828: (8)                    self,
829: (8)                    backend_name,
830: (8)                    graph,
831: (8)                    *,
832: (8)                    edge_attrs,
833: (8)                    node_attrs,
834: (8)                    preserve_edge_attrs,
835: (8)                    preserve_node_attrs,
836: (8)                    preserve_graph_attrs,
837: (8)                    graph_name,
838: (8)                    use_cache,
839: (4)                ):
840: (8)                    if (
841: (12)                       use_cache
842: (12)                       and (nx_cache := getattr(graph, "__networkx_cache__", None)) is
not None
843: (8)                    ):
844: (12)                       cache = nx_cache.setdefault("backends",
{}).setdefault(backend_name, {})
845: (12)                       key = edge_key, node_key, graph_key = (
846: (16)                           frozenset(edge_attrs.items())
847: (16)                           if edge_attrs is not None
848: (16)                           else preserve_edge_attrs,
849: (16)                           frozenset(node_attrs.items())
850: (16)                           if node_attrs is not None
```

```
(16)                            else preserve_node_attrs,
(16)                            preserve_graph_attrs,
(12)                        )
(12)                        if cache:
(16)                            warning_message = (
(20)                                f"Using cached graph for {backend_name!r} backend in "
(20)                                f"call to {self.name}.\n\nFor the cache to be consistent "
(20)                                "(i.e., correct), the input graph must not have been "
(20)                                "manually mutated since the cached graph was created. "
(20)                                "Examples of manually mutating the graph data structures "
(20)                                "resulting in an inconsistent cache include:\n\n"
(20)                                "    >>> G[u][v][key] = val\n\n"
(20)                                "and\n\n"
(20)                                "    >>> for u, v, d in G.edges(data=True):\n"
(20)                                "    ...     d[key] = val\n\n"
(20)                                "Using methods such as `G.add_edge(u, v, weight=val)` "
(20)                                "will correctly clear the cache to keep it consistent. "
(20)                                "You may also use `G.__networkx_cache__.clear()` to "
(20)                                "manually clear the cache, or set `G.__networkx_cache__` "
(20)                                "to None to disable caching for G. Enable or disable "
(20)                                "caching via `nx.config.cache_converted_graphs` config."
(16)                            )
(16)                            for compat_key in itertools.product(
(20)                                (edge_key, True) if edge_key is not True else (True,),
(20)                                (node_key, True) if node_key is not True else (True,),
(20)                                (graph_key, True) if graph_key is not True else (True,),
(16)                            ):
(20)                                if (rv := cache.get(compat_key)) is not None:
(24)                                    warnings.warn(warning_message)
(24)                                    return rv
(16)                            if edge_key is not True and node_key is not True:
(20)                                for (ekey, nkey, gkey), val in list(cache.items()):
(24)                                    if edge_key is False or ekey is True:
(28)                                        pass
(24)                                    elif (
(28)                                        edge_key is True
(28)                                        or ekey is False
(28)                                        or not edge_key.issubset(ekey)
(24)                                    ):
(28)                                        continue
(24)                                    if node_key is False or nkey is True:
(28)                                        pass
(24)                                    elif (
(28)                                        node_key is True
(28)                                        or nkey is False
(28)                                        or not node_key.issubset(nkey)
(24)                                    ):
(28)                                        continue
(24)                                    if graph_key and not gkey:
(28)                                        continue
(24)                                    warnings.warn(warning_message)
(24)                                    return val
(8)                     backend = _load_backend(backend_name)
(8)                     rv = backend.convert_from_nx(
(12)                        graph,
(12)                        edge_attrs=edge_attrs,
(12)                        node_attrs=node_attrs,
(12)                        preserve_edge_attrs=preserve_edge_attrs,
(12)                        preserve_node_attrs=preserve_node_attrs,
(12)                        preserve_graph_attrs=preserve_graph_attrs,
(12)                        name=self.name,
(12)                        graph_name=graph_name,
(8)                     )
(8)                     if use_cache and nx_cache is not None:
(12)                        cache[key] = rv  # Set at beginning to be thread-safe
(12)                        for cur_key in list(cache):
(16)                            if cur_key == key:
(20)                                continue
(16)                            ekey, nkey, gkey = cur_key
```

```
920: (16)                           if ekey is False or edge_key is True:
921: (20)                               pass
922: (16)                           elif ekey is True or edge_key is False or not
ekey.issubset(edge_key):
923: (20)                               continue
924: (16)                           if nkey is False or node_key is True:
925: (20)                               pass
926: (16)                           elif nkey is True or node_key is False or not
nkey.issubset(node_key):
927: (20)                               continue
928: (16)                           if gkey and not graph_key:
929: (20)                               continue
930: (16)                           cache.pop(cur_key, None)  # Use pop instead of del to be
thread-safe
931: (8)                    return rv
932: (4)                def _convert_and_call(self, backend_name, args, kwargs, *,
fallback_to_nx=False):
933: (8)                    """Call this dispatchable function with a backend, converting graphs
if necessary."""
934: (8)                    backend = _load_backend(backend_name)
935: (8)                    if not self._can_backend_run(backend_name, *args, **kwargs):
936: (12)                       if fallback_to_nx:
937: (16)                           return self.orig_func(*args, **kwargs)
938: (12)                       msg = f"'{self.name}' not implemented by {backend_name}"
939: (12)                       if hasattr(backend, self.name):
940: (16)                           msg += " with the given arguments"
941: (12)                       raise RuntimeError(msg)
942: (8)                    try:
943: (12)                       converted_args, converted_kwargs = self._convert_arguments(
944: (16)                           backend_name, args, kwargs,
use_cache=config.cache_converted_graphs
945: (12)                       )
946: (12)                       result = getattr(backend, self.name)(*converted_args,
**converted_kwargs)
947: (8)                    except (NotImplementedError, nx.NetworkXNotImplemented) as exc:
948: (12)                       if fallback_to_nx:
949: (16)                           return self.orig_func(*args, **kwargs)
950: (12)                       raise
951: (8)                    return result
952: (4)                def _convert_and_call_for_tests(
953: (8)                    self, backend_name, args, kwargs, *, fallback_to_nx=False
954: (4)                ):
955: (8)                    """Call this dispatchable function with a backend; for use with
testing."""
956: (8)                    backend = _load_backend(backend_name)
957: (8)                    if not self._can_backend_run(backend_name, *args, **kwargs):
958: (12)                       if fallback_to_nx or not self.graphs:
959: (16)                           return self.orig_func(*args, **kwargs)
960: (12)                       import pytest
961: (12)                       msg = f"'{self.name}' not implemented by {backend_name}"
962: (12)                       if hasattr(backend, self.name):
963: (16)                           msg += " with the given arguments"
964: (12)                       pytest.xfail(msg)
965: (8)                    from collections.abc import Iterable, Iterator, Mapping
966: (8)                    from copy import copy
967: (8)                    from io import BufferedReader, BytesIO, StringIO, TextIOWrapper
968: (8)                    from itertools import tee
969: (8)                    from random import Random
970: (8)                    import numpy as np
971: (8)                    from numpy.random import Generator, RandomState
972: (8)                    from scipy.sparse import sparray
973: (8)                    if not args:
974: (12)                       args1 = args2 = args
975: (8)                    else:
976: (12)                       args1, args2 = zip(
977: (16)                           *(
978: (20)                               (arg, copy(arg))
979: (20)                               if isinstance(
980: (24)                                   arg, BytesIO | StringIO | Random | Generator |
```

```
RandomState
981: (20)                                )
982: (20)                                else tee(arg)
983: (20)                                if isinstance(arg, Iterator)
984: (20)                                and not isinstance(arg, BufferedReader | TextIOWrapper)
985: (20)                                else (arg, arg)
986: (20)                                for arg in args
987: (16)                            )
988: (12)                        )
989: (8)                    if not kwargs:
990: (12)                        kwargs1 = kwargs2 = kwargs
991: (8)                    else:
992: (12)                        kwargs1, kwargs2 = zip(
993: (16)                            *(
994: (20)                                ((k, v), (k, copy(v)))
995: (20)                                if isinstance(
996: (24)                                    v, BytesIO | StringIO | Random | Generator |
RandomState
997: (20)                                )
998: (20)                                else ((k, (teed := tee(v))[0]), (k, teed[1]))
999: (20)                                if isinstance(v, Iterator)
1000: (20)                               and not isinstance(v, BufferedReader | TextIOWrapper)
1001: (20)                               else ((k, v), (k, v))
1002: (20)                               for k, v in kwargs.items()
1003: (16)                           )
1004: (12)                       )
1005: (12)                       kwargs1 = dict(kwargs1)
1006: (12)                       kwargs2 = dict(kwargs2)
1007: (8)                   try:
1008: (12)                       converted_args, converted_kwargs = self._convert_arguments(
1009: (16)                           backend_name, args1, kwargs1, use_cache=False
1010: (12)                       )
1011: (12)                       result = getattr(backend, self.name)(*converted_args,
**converted_kwargs)
1012: (8)                   except (NotImplementedError, nx.NetworkXNotImplemented) as exc:
1013: (12)                       if fallback_to_nx:
1014: (16)                           return self.orig_func(*args2, **kwargs2)
1015: (12)                       import pytest
1016: (12)                       pytest.xfail(
1017: (16)                           exc.args[0] if exc.args else f"{self.name} raised
{type(exc).__name__}"
1018: (12)                       )
1019: (8)                   if (
1020: (12)                       self._returns_graph
1021: (12)                       != (
1022: (16)                           isinstance(result, nx.Graph)
1023: (16)                           or hasattr(result, "__networkx_backend__")
1024: (16)                           or isinstance(result, tuple | list)
1025: (16)                           and any(
1026: (20)                               isinstance(x, nx.Graph) or hasattr(x,
"__networkx_backend__")
1027: (20)                               for x in result
1028: (16)                           )
1029: (12)                       )
1030: (12)                       and not (
1031: (16)                           self.name in {"check_planarity", "check_planarity_recursive"}
1032: (16)                           and any(x is None for x in result)
1033: (12)                       )
1034: (12)                       and not (
1035: (16)                           self.name in {"held_karp_ascent"}
1036: (16)                           and any(isinstance(x, dict) for x in result)
1037: (12)                       )
1038: (12)                       and self.name
1039: (12)                       not in {
1040: (16)                           "all_triads",
1041: (16)                           "general_k_edge_subgraphs",
1042: (16)                           "nonisomorphic_trees",
1043: (12)                       }
1044: (8)                   ):
```

```
1045: (12)                      raise RuntimeError(f"`returns_graph` is incorrect for
{self.name}")
1046: (8)                   def check_result(val, depth=0):
1047: (12)                      if isinstance(val, np.number):
1048: (16)                          raise RuntimeError(
1049: (20)                              f"{self.name} returned a numpy scalar {val} ({type(val)},
depth={depth})"
1050: (16)                          )
1051: (12)                      if isinstance(val, np.ndarray | sparray):
1052: (16)                          return
1053: (12)                      if isinstance(val, nx.Graph):
1054: (16)                          check_result(val._node, depth=depth + 1)
1055: (16)                          check_result(val._adj, depth=depth + 1)
1056: (16)                          return
1057: (12)                      if isinstance(val, Iterator):
1058: (16)                          raise NotImplementedError
1059: (12)                      if isinstance(val, Iterable) and not isinstance(val, str):
1060: (16)                          for x in val:
1061: (20)                              check_result(x, depth=depth + 1)
1062: (12)                      if isinstance(val, Mapping):
1063: (16)                          for x in val.values():
1064: (20)                              check_result(x, depth=depth + 1)
1065: (8)                   def check_iterator(it):
1066: (12)                      for val in it:
1067: (16)                          try:
1068: (20)                              check_result(val)
1069: (16)                          except RuntimeError as exc:
1070: (20)                              raise RuntimeError(
1071: (24)                                  f"{self.name} returned a numpy scalar {val}
({type(val)})"
1072: (20)                              ) from exc
1073: (16)                          yield val
1074: (8)                   if self.name in {"from_edgelist"}:
1075: (12)                      pass
1076: (8)                   elif isinstance(result, Iterator):
1077: (12)                      result = check_iterator(result)
1078: (8)                   else:
1079: (12)                      try:
1080: (16)                          check_result(result)
1081: (12)                      except RuntimeError as exc:
1082: (16)                          raise RuntimeError(
1083: (20)                              f"{self.name} returned a numpy scalar {result}
({type(result)})"
1084: (16)                          ) from exc
1085: (12)                      check_result(result)
1086: (8)                   if self.name in {
1087: (12)                      "edmonds_karp",
1088: (12)                      "barycenter",
1089: (12)                      "contracted_edge",
1090: (12)                      "contracted_nodes",
1091: (12)                      "stochastic_graph",
1092: (12)                      "relabel_nodes",
1093: (12)                      "maximum_branching",
1094: (12)                      "incremental_closeness_centrality",
1095: (12)                      "minimal_branching",
1096: (12)                      "minimum_spanning_arborescence",
1097: (12)                      "recursive_simple_cycles",
1098: (12)                      "connected_double_edge_swap",
1099: (8)                   }:
1100: (12)                      bound = self.__signature__.bind(*converted_args,
**converted_kwargs)
1101: (12)                      bound.apply_defaults()
1102: (12)                      bound2 = self.__signature__.bind(*args2, **kwargs2)
1103: (12)                      bound2.apply_defaults()
1104: (12)                      if self.name in {
1105: (16)                          "minimal_branching",
1106: (16)                          "minimum_spanning_arborescence",
1107: (16)                          "recursive_simple_cycles",
1108: (16)                          "connected_double_edge_swap",
```

```
(12)                        }:
(16)                            G1 = backend.convert_to_nx(bound.arguments["G"])
(16)                            G2 = bound2.arguments["G"]
(16)                            G2._adj = G1._adj
(16)                            nx._clear_cache(G2)
(12)                        elif self.name == "edmonds_karp":
(16)                            R1 = backend.convert_to_nx(bound.arguments["residual"])
(16)                            R2 = bound2.arguments["residual"]
(16)                            if R1 is not None and R2 is not None:
(20)                                for k, v in R1.edges.items():
(24)                                    R2.edges[k]["flow"] = v["flow"]
(20)                                R2.graph.update(R1.graph)
(20)                                nx._clear_cache(R2)
(12)                        elif self.name == "barycenter" and bound.arguments["attr"] is not
None:
(16)                            G1 = backend.convert_to_nx(bound.arguments["G"])
(16)                            G2 = bound2.arguments["G"]
(16)                            attr = bound.arguments["attr"]
(16)                            for k, v in G1.nodes.items():
(20)                                G2.nodes[k][attr] = v[attr]
(16)                            nx._clear_cache(G2)
(12)                        elif (
(16)                            self.name in {"contracted_nodes", "contracted_edge"}
(16)                            and not bound.arguments["copy"]
(12)                        ):
(16)                            G1 = backend.convert_to_nx(bound.arguments["G"])
(16)                            G2 = bound2.arguments["G"]
(16)                            G2.__dict__.update(G1.__dict__)
(16)                            nx._clear_cache(G2)
(12)                        elif self.name == "stochastic_graph" and not
bound.arguments["copy"]:
(16)                            G1 = backend.convert_to_nx(bound.arguments["G"])
(16)                            G2 = bound2.arguments["G"]
(16)                            for k, v in G1.edges.items():
(20)                                G2.edges[k]["weight"] = v["weight"]
(16)                            nx._clear_cache(G2)
(12)                        elif (
(16)                            self.name == "relabel_nodes"
(16)                            and not bound.arguments["copy"]
(16)                            or self.name in {"incremental_closeness_centrality"}
(12)                        ):
(16)                            G1 = backend.convert_to_nx(bound.arguments["G"])
(16)                            G2 = bound2.arguments["G"]
(16)                            if G1 is G2:
(20)                                return G2
(16)                            G2._node.clear()
(16)                            G2._node.update(G1._node)
(16)                            G2._adj.clear()
(16)                            G2._adj.update(G1._adj)
(16)                            if hasattr(G1, "_pred") and hasattr(G2, "_pred"):
(20)                                G2._pred.clear()
(20)                                G2._pred.update(G1._pred)
(16)                            if hasattr(G1, "_succ") and hasattr(G2, "_succ"):
(20)                                G2._succ.clear()
(20)                                G2._succ.update(G1._succ)
(16)                            nx._clear_cache(G2)
(16)                            if self.name == "relabel_nodes":
(20)                                return G2
(12)                        return backend.convert_to_nx(result)
(8)                     converted_result = backend.convert_to_nx(result)
(8)                     if isinstance(converted_result, nx.Graph) and self.name not in {
(12)                        "boykov_kolmogorov",
(12)                        "preflow_push",
(12)                        "quotient_graph",
(12)                        "shortest_augmenting_path",
(12)                        "spectral_graph_forge",
(12)                        "read_gml",
(12)                        "read_graph6",
(12)                        "read_sparse6",
```

```
1176: (12)                      "bipartite_read_edgelist",
1177: (12)                      "read_adjlist",
1178: (12)                      "read_edgelist",
1179: (12)                      "read_graphml",
1180: (12)                      "read_multiline_adjlist",
1181: (12)                      "read_pajek",
1182: (12)                      "from_pydot",
1183: (12)                      "pydot_read_dot",
1184: (12)                      "agraph_read_dot",
1185: (12)                      "read_gexf",
1186: (8)                   }:
1187: (12)                      G = self.orig_func(*args2, **kwargs2)
1188: (12)                      if not nx.utils.graphs_equal(G, converted_result):
1189: (16)                          assert G.number_of_nodes() ==
converted_result.number_of_nodes()
1190: (16)                          assert G.number_of_edges() ==
converted_result.number_of_edges()
1191: (16)                          assert G.graph == converted_result.graph
1192: (16)                          assert G.nodes == converted_result.nodes
1193: (16)                          assert G.adj == converted_result.adj
1194: (16)                          assert type(G) is type(converted_result)
1195: (16)                          raise AssertionError("Graphs are not equal")
1196: (12)                      return G
1197: (8)                   return converted_result
1198: (4)               def _make_doc(self):
1199: (8)                   """Generate the backends section at the end for functions having an
alternate
1200: (8)                   backend implementation(s) using the `backend_info` entry-point."""
1201: (8)                   if not self.backends:
1202: (12)                      return self._orig_doc
1203: (8)                   lines = [
1204: (12)                      "Backends",
1205: (12)                      "--------",
1206: (8)                   ]
1207: (8)                   for backend in sorted(self.backends):
1208: (12)                      info = backend_info[backend]
1209: (12)                      if "short_summary" in info:
1210: (16)                          lines.append(f"{backend} : {info['short_summary']}")
1211: (12)                      else:
1212: (16)                          lines.append(backend)
1213: (12)                      if "functions" not in info or self.name not in info["functions"]:
1214: (16)                          lines.append("")
1215: (16)                          continue
1216: (12)                      func_info = info["functions"][self.name]
1217: (12)                      if func_docs := (
1218: (16)                          func_info.get("additional_docs") or
func_info.get("extra_docstring")
1219: (12)                      ):
1220: (16)                          lines.extend(
1221: (20)                              f"  {line}" if line else line for line in
func_docs.split("\n")
1222: (16)                          )
1223: (16)                          add_gap = True
1224: (12)                      else:
1225: (16)                          add_gap = False
1226: (12)                      if extra_parameters := (
1227: (16)                          func_info.get("extra_parameters")
1228: (16)                          or func_info.get("additional_parameters")
1229: (12)                      ):
1230: (16)                          if add_gap:
1231: (20)                              lines.append("")
1232: (16)                          lines.append("  Additional parameters:")
1233: (16)                          for param in sorted(extra_parameters):
1234: (20)                              lines.append(f"    {param}")
1235: (20)                              if desc := extra_parameters[param]:
1236: (24)                                  lines.append(f"      {desc}")
1237: (20)                              lines.append("")
1238: (12)                      else:
1239: (16)                          lines.append("")
```

```
1240: (12)                      if func_url := func_info.get("url"):
1241: (16)                          lines.append(f"[`Source <{func_url}>`_]")
1242: (16)                          lines.append("")
1243: (8)                   lines.pop()  # Remove last empty line
1244: (8)                   to_add = "\n    ".join(lines)
1245: (8)                   return f"{self._orig_doc.rstrip()}\n\n    {to_add}"
1246: (4)               def __reduce__(self):
1247: (8)                   """Allow this object to be serialized with pickle.
1248: (8)                   This uses the global registry `_registered_algorithms` to deserialize.
1249: (8)                   """
1250: (8)                   return _restore_dispatchable, (self.name,)
1251: (0)           def _restore_dispatchable(name):
1252: (4)               return _registered_algorithms[name]
1253: (0)           if os.environ.get("_NETWORKX_BUILDING_DOCS_"):
1254: (4)               _orig_dispatchable = _dispatchable
1255: (4)               def _dispatchable(func=None, **kwargs):  # type: ignore[no-redef]
1256: (8)                   if func is None:
1257: (12)                      return partial(_dispatchable, **kwargs)
1258: (8)                   dispatched_func = _orig_dispatchable(func, **kwargs)
1259: (8)                   func.__doc__ = dispatched_func.__doc__
1260: (8)                   return func
1261: (4)               _dispatchable.__doc__ = _orig_dispatchable.__new__.__doc__  # type:
ignore[method-assign,assignment]
1262: (4)               _sig = inspect.signature(_orig_dispatchable.__new__)
1263: (4)               _dispatchable.__signature__ = _sig.replace(  # type: ignore[method-
assign,assignment]
1264: (8)                   parameters=[v for k, v in _sig.parameters.items() if k != "cls"]
1265: (4)               )
```

File 548 - configs.py:

```
1: (0)              import collections
2: (0)              import os
3: (0)              import typing
4: (0)              from dataclasses import dataclass
5: (0)              __all__ = ["Config", "config"]
6: (0)              @dataclass(init=False, eq=False, slots=True, kw_only=True, match_args=False)
7: (0)              class Config:
8: (4)                  """The base class for NetworkX configuration.
9: (4)                  There are two ways to use this to create configurations. The first is to
10: (4)                 simply pass the initial configuration as keyword arguments to ``Config``:
11: (4)                 >>> cfg = Config(eggs=1, spam=5)
12: (4)                 >>> cfg
13: (4)                 Config(eggs=1, spam=5)
14: (4)                 The second--and preferred--way is to subclass ``Config`` with docs and
annotations.
15: (4)                 >>> class MyConfig(Config):
16: (4)                 ...     '''Breakfast!'''
17: (4)                 ...
18: (4)                 ...     eggs: int
19: (4)                 ...     spam: int
20: (4)                 ...
21: (4)                 ...     def _check_config(self, key, value):
22: (4)                 ...         assert isinstance(value, int) and value >= 0
23: (4)                 >>> cfg = MyConfig(eggs=1, spam=5)
24: (4)                 Once defined, config items may be modified, but can't be added or deleted
by default.
25: (4)                 ``Config`` is a ``Mapping``, and can get and set configs via attributes or
brackets:
26: (4)                 >>> cfg.eggs = 2
27: (4)                 >>> cfg.eggs
28: (4)                 2
29: (4)                 >>> cfg["spam"] = 42
30: (4)                 >>> cfg["spam"]
31: (4)                 42
32: (4)                 Subclasses may also define ``_check_config`` (as done in the example
above)
```

```
(4)                 to ensure the value being assigned is valid:
(4)                 >>> cfg.spam = -1
(4)                 Traceback (most recent call last):
(8)                     ...
(4)                 AssertionError
(4)                 If a more flexible configuration object is needed that allows adding and deleting
(4)                 configurations, then pass ``strict=False`` when defining the subclass:
(4)                 >>> class FlexibleConfig(Config, strict=False):
(4)                 ...     default_greeting: str = "Hello"
(4)                 >>> flexcfg = FlexibleConfig()
(4)                 >>> flexcfg.name = "Mr. Anderson"
(4)                 >>> flexcfg
(4)                 FlexibleConfig(default_greeting='Hello', name='Mr. Anderson')
(4)                 """
(4)                 def __init_subclass__(cls, strict=True):
(8)                     cls._strict = strict
(4)                 def __new__(cls, **kwargs):
(8)                     orig_class = cls
(8)                     if cls is Config:
(12)                        cls = type(
(16)                            cls.__name__,
(16)                            (cls,),
(16)                            {"__annotations__": {key: typing.Any for key in kwargs}},
(12)                        )
(8)                     cls = dataclass(
(12)                        eq=False,
(12)                        repr=cls._strict,
(12)                        slots=cls._strict,
(12)                        kw_only=True,
(12)                        match_args=False,
(8)                     )(cls)
(8)                     if not cls._strict:
(12)                        cls.__repr__ = _flexible_repr
(8)                     cls._orig_class = orig_class  # Save original class so we can pickle
(8)                     instance = object.__new__(cls)
(8)                     instance.__init__(**kwargs)
(8)                     return instance
(4)                 def _check_config(self, key, value):
(8)                     """Check whether config value is valid. This is useful for subclasses."""
(4)                 def __dir__(self):
(8)                     return self.__dataclass_fields__.keys()
(4)                 def __setattr__(self, key, value):
(8)                     if self._strict and key not in self.__dataclass_fields__:
(12)                        raise AttributeError(f"Invalid config name: {key!r}")
(8)                     self._check_config(key, value)
(8)                     object.__setattr__(self, key, value)
(4)                 def __delattr__(self, key):
(8)                     if self._strict:
(12)                        raise TypeError(
(16)                            f"Configuration items can't be deleted (can't delete {key!r})."
(12)                        )
(8)                     object.__delattr__(self, key)
(4)                 def __contains__(self, key):
(8)                     return (
(12)                        key in self.__dataclass_fields__ if self._strict else key in self.__dict__
(8)                     )
(4)                 def __iter__(self):
(8)                     return iter(self.__dataclass_fields__ if self._strict else self.__dict__)
(4)                 def __len__(self):
(8)                     return len(self.__dataclass_fields__ if self._strict else self.__dict__)
(4)                 def __reversed__(self):
(8)                     return reversed(self.__dataclass_fields__ if self._strict else self.__dict__)
```

```
95: (4)                 def __getitem__(self, key):
96: (8)                     try:
97: (12)                        return getattr(self, key)
98: (8)                     except AttributeError as err:
99: (12)                        raise KeyError(*err.args) from None
100: (4)                def __setitem__(self, key, value):
101: (8)                    try:
102: (12)                       self.__setattr__(key, value)
103: (8)                    except AttributeError as err:
104: (12)                       raise KeyError(*err.args) from None
105: (4)                def __delitem__(self, key):
106: (8)                    try:
107: (12)                       self.__delattr__(key)
108: (8)                    except AttributeError as err:
109: (12)                       raise KeyError(*err.args) from None
110: (4)                _ipython_key_completions_ = __dir__  # config["<TAB>
111: (4)                def get(self, key, default=None):
112: (8)                    return getattr(self, key, default)
113: (4)                def items(self):
114: (8)                    return collections.abc.ItemsView(self)
115: (4)                def keys(self):
116: (8)                    return collections.abc.KeysView(self)
117: (4)                def values(self):
118: (8)                    return collections.abc.ValuesView(self)
119: (4)                def __eq__(self, other):
120: (8)                    if not isinstance(other, Config):
121: (12)                       return NotImplemented
122: (8)                    return self._orig_class == other._orig_class and self.items() ==
other.items()
123: (4)                def __reduce__(self):
124: (8)                    return self._deserialize, (self._orig_class, dict(self))
125: (4)                @staticmethod
126: (4)                def _deserialize(cls, kwargs):
127: (8)                    return cls(**kwargs)
128: (0)            def _flexible_repr(self):
129: (4)                return (
130: (8)                    f"{self.__class__.__qualname__}("
131: (8)                    + ", ".join(f"{key}={val!r}" for key, val in self.__dict__.items())
132: (8)                    + ")"
133: (4)                )
134: (0)            collections.abc.Mapping.register(Config)
135: (0)            class NetworkXConfig(Config):
136: (4)                """Configuration for NetworkX that controls behaviors such as how to use
backends.
137: (4)                Attribute and bracket notation are supported for getting and setting
configurations:
138: (4)                >>> nx.config.backend_priority == nx.config["backend_priority"]
139: (4)                True
140: (4)                Parameters
141: (4)                ----------
142: (4)                backend_priority : list of backend names
143: (8)                    Enable automatic conversion of graphs to backend graphs for algorithms
144: (8)                    implemented by the backend. Priority is given to backends listed
earlier.
145: (8)                    Default is empty list.
146: (4)                backends : Config mapping of backend names to backend Config
147: (8)                    The keys of the Config mapping are names of all installed NetworkX
backends,
148: (8)                    and the values are their configurations as Config mappings.
149: (4)                cache_converted_graphs : bool
150: (8)                    If True, then save converted graphs to the cache of the input graph.
Graph
151: (8)                    conversion may occur when automatically using a backend from
`backend_priority`
152: (8)                    or when using the `backend=` keyword argument to a function call.
Caching can
153: (8)                    improve performance by avoiding repeated conversions, but it uses more
memory.
154: (8)                    Care should be taken to not manually mutate a graph that has cached
```

```
graphs; for
155: (8)                    example, ``G[u][v][k] = val`` changes the graph, but does not clear
the cache.
156: (8)                    Using methods such as ``G.add_edge(u, v, weight=val)`` will clear the
cache to
157: (8)                    keep it consistent. ``G.__networkx_cache__.clear()`` manually clears
the cache.
158: (8)                    Default is False.
159: (4)                Notes
160: (4)                -----
161: (4)                Environment variables may be used to control some default configurations:
162: (4)                - NETWORKX_BACKEND_PRIORITY: set `backend_priority` from comma-separated
names.
163: (4)                - NETWORKX_CACHE_CONVERTED_GRAPHS: set `cache_converted_graphs` to True if
nonempty.
164: (4)                This is a global configuration. Use with caution when using from multiple
threads.
165: (4)                """
166: (4)                backend_priority: list[str]
167: (4)                backends: Config
168: (4)                cache_converted_graphs: bool
169: (4)                def _check_config(self, key, value):
170: (8)                    from .backends import backends
171: (8)                    if key == "backend_priority":
172: (12)                       if not (isinstance(value, list) and all(isinstance(x, str) for x
in value)):
173: (16)                           raise TypeError(
174: (20)                               f"{key!r} config must be a list of backend names; got
{value!r}"
175: (16)                           )
176: (12)                       if missing := {x for x in value if x not in backends}:
177: (16)                           missing = ", ".join(map(repr, sorted(missing)))
178: (16)                           raise ValueError(f"Unknown backend when setting {key!r}:
{missing}")
179: (8)                    elif key == "backends":
180: (12)                       if not (
181: (16)                           isinstance(value, Config)
182: (16)                           and all(isinstance(key, str) for key in value)
183: (16)                           and all(isinstance(val, Config) for val in value.values())
184: (12)                       ):
185: (16)                           raise TypeError(
186: (20)                               f"{key!r} config must be a Config of backend configs; got
{value!r}"
187: (16)                           )
188: (12)                       if missing := {x for x in value if x not in backends}:
189: (16)                           missing = ", ".join(map(repr, sorted(missing)))
190: (16)                           raise ValueError(f"Unknown backend when setting {key!r}:
{missing}")
191: (8)                    elif key == "cache_converted_graphs":
192: (12)                       if not isinstance(value, bool):
193: (16)                           raise TypeError(f"{key!r} config must be True or False; got
{value!r}")
194: (0)            config = NetworkXConfig(
195: (4)                backend_priority=[],
196: (4)                backends=Config(),
197: (4)
cache_converted_graphs=bool(os.environ.get("NETWORKX_CACHE_CONVERTED_GRAPHS", "")),
198: (0)            )

----------------------------------------

File 549 - decorators.py:

1: (0)              import bz2
2: (0)              import collections
3: (0)              import gzip
4: (0)              import inspect
5: (0)              import itertools
6: (0)              import re
```

```
7: (0)              import warnings
8: (0)              from collections import defaultdict
9: (0)              from contextlib import contextmanager
10: (0)             from functools import wraps
11: (0)             from inspect import Parameter, signature
12: (0)             from os.path import splitext
13: (0)             from pathlib import Path
14: (0)             import networkx as nx
15: (0)             from networkx.utils import create_py_random_state, create_random_state
16: (0)             __all__ = [
17: (4)                 "not_implemented_for",
18: (4)                 "open_file",
19: (4)                 "nodes_or_number",
20: (4)                 "np_random_state",
21: (4)                 "py_random_state",
22: (4)                 "argmap",
23: (4)                 "deprecate_positional_args",
24: (0)             ]
25: (0)             def not_implemented_for(*graph_types):
26: (4)                 """Decorator to mark algorithms as not implemented
27: (4)                 Parameters
28: (4)                 ----------
29: (4)                 graph_types : container of strings
30: (8)                     Entries must be one of "directed", "undirected", "multigraph", or
"graph".
31: (4)                 Returns
32: (4)                 -------
33: (4)                 _require : function
34: (8)                     The decorated function.
35: (4)                 Raises
36: (4)                 ------
37: (4)                 NetworkXNotImplemented
38: (4)                 If any of the packages cannot be imported
39: (4)                 Notes
40: (4)                 -----
41: (4)                 Multiple types are joined logically with "and".
42: (4)                 For "or" use multiple @not_implemented_for() lines.
43: (4)                 Examples
44: (4)                 --------
45: (4)                 Decorate functions like this::
46: (7)                    @not_implemented_for("directed")
47: (7)                    def sp_function(G):
48: (11)                       pass
49: (7)                    @not_implemented_for("directed", "multigraph")
50: (7)                    def sp_np_function(G):
51: (11)                       pass
52: (7)                    @not_implemented_for("undirected")
53: (7)                    @not_implemented_for("multigraph")
54: (7)                    def sp_np_function(G):
55: (11)                       pass
56: (4)                 """
57: (4)                 if ("directed" in graph_types) and ("undirected" in graph_types):
58: (8)                     raise ValueError("Function not implemented on directed AND undirected
graphs?")
59: (4)                 if ("multigraph" in graph_types) and ("graph" in graph_types):
60: (8)                     raise ValueError("Function not implemented on graph AND multigraphs?")
61: (4)                 if not set(graph_types) < {"directed", "undirected", "multigraph",
"graph"}:
62: (8)                     raise KeyError(
63: (12)                        "use one or more of directed, undirected, multigraph, graph.  "
64: (12)                        f"You used {graph_types}"
65: (8)                     )
66: (4)                 dval = ("directed" in graph_types) or "undirected" not in graph_types and
None
67: (4)                 mval = ("multigraph" in graph_types) or "graph" not in graph_types and
None
68: (4)                 errmsg = f"not implemented for {' '.join(graph_types)} type"
69: (4)                 def _not_implemented_for(g):
70: (8)                     if (mval is None or mval == g.is_multigraph()) and (
```

```
 71: (12)                        dval is None or dval == g.is_directed()
 72: (8)                     ):
 73: (12)                        raise nx.NetworkXNotImplemented(errmsg)
 74: (8)                     return g
 75: (4)                 return argmap(_not_implemented_for, 0)
 76: (0)             fopeners = {
 77: (4)                 ".gz": gzip.open,
 78: (4)                 ".gzip": gzip.open,
 79: (4)                 ".bz2": bz2.BZ2File,
 80: (0)             }
 81: (0)             _dispatch_dict = defaultdict(lambda: open, **fopeners)
 82: (0)             def open_file(path_arg, mode="r"):
 83: (4)                 """Decorator to ensure clean opening and closing of files.
 84: (4)                 Parameters
 85: (4)                 ----------
 86: (4)                 path_arg : string or int
 87: (8)                     Name or index of the argument that is a path.
 88: (4)                 mode : str
 89: (8)                     String for opening mode.
 90: (4)                 Returns
 91: (4)                 -------
 92: (4)                 _open_file : function
 93: (8)                     Function which cleanly executes the io.
 94: (4)                 Examples
 95: (4)                 --------
 96: (4)                 Decorate functions like this::
 97: (7)                    @open_file(0, "r")
 98: (7)                    def read_function(pathname):
 99: (11)                       pass
 100: (7)                   @open_file(1, "w")
 101: (7)                   def write_function(G, pathname):
 102: (11)                      pass
 103: (7)                   @open_file(1, "w")
 104: (7)                   def write_function(G, pathname="graph.dot"):
 105: (11)                      pass
 106: (7)                   @open_file("pathname", "w")
 107: (7)                   def write_function(G, pathname="graph.dot"):
 108: (11)                      pass
 109: (7)                   @open_file("path", "w+")
 110: (7)                   def another_function(arg, **kwargs):
 111: (11)                      path = kwargs["path"]
 112: (11)                      pass
 113: (4)                 Notes
 114: (4)                 -----
 115: (4)                 Note that this decorator solves the problem when a path argument is
 116: (4)                 specified as a string, but it does not handle the situation when the
 117: (4)                 function wants to accept a default of None (and then handle it).
 118: (4)                 Here is an example of how to handle this case::
 119: (6)                   @open_file("path")
 120: (6)                   def some_function(arg1, arg2, path=None):
 121: (10)                      if path is None:
 122: (14)                          fobj = tempfile.NamedTemporaryFile(delete=False)
 123: (10)                      else:
 124: (14)                          fobj = path
 125: (10)                      try:
 126: (14)                          fobj.write("blah")
 127: (10)                      finally:
 128: (14)                          if path is None:
 129: (18)                              fobj.close()
 130: (4)                 Normally, we'd want to use "with" to ensure that fobj gets closed.
 131: (4)                 However, the decorator will make `path` a file object for us,
 132: (4)                 and using "with" would undesirably close that file object.
 133: (4)                 Instead, we use a try block, as shown above.
 134: (4)                 When we exit the function, fobj will be closed, if it should be, by the
decorator.
 135: (4)                 """
 136: (4)                 def _open_file(path):
 137: (8)                     if isinstance(path, str):
 138: (12)                        ext = splitext(path)[1]
```

```
139: (8)                    elif isinstance(path, Path):
140: (12)                       ext = path.suffix
141: (12)                       path = str(path)
142: (8)                    else:
143: (12)                       return path, lambda: None
144: (8)                    fobj = _dispatch_dict[ext](path, mode=mode)
145: (8)                    return fobj, lambda: fobj.close()
146: (4)                return argmap(_open_file, path_arg, try_finally=True)
147: (0)            def nodes_or_number(which_args):
148: (4)                """Decorator to allow number of nodes or container of nodes.
149: (4)                With this decorator, the specified argument can be either a number or a
container
150: (4)                of nodes. If it is a number, the nodes used are `range(n)`.
151: (4)                This allows `nx.complete_graph(50)` in place of
`nx.complete_graph(list(range(50)))`.
152: (4)                And it also allows `nx.complete_graph(any_list_of_nodes)`.
153: (4)                Parameters
154: (4)                ----------
155: (4)                which_args : string or int or sequence of strings or ints
156: (8)                    If string, the name of the argument to be treated.
157: (8)                    If int, the index of the argument to be treated.
158: (8)                    If more than one node argument is allowed, can be a list of locations.
159: (4)                Returns
160: (4)                -------
161: (4)                _nodes_or_numbers : function
162: (8)                    Function which replaces int args with ranges.
163: (4)                Examples
164: (4)                --------
165: (4)                Decorate functions like this::
166: (7)                   @nodes_or_number("nodes")
167: (7)                   def empty_graph(nodes):
168: (7)                   @nodes_or_number(0)
169: (7)                   def empty_graph(nodes):
170: (7)                   @nodes_or_number(["m1", "m2"])
171: (7)                   def grid_2d_graph(m1, m2, periodic=False):
172: (7)                   @nodes_or_number([0, 1])
173: (7)                   def grid_2d_graph(m1, m2, periodic=False):
174: (7)                   @nodes_or_number(1)
175: (7)                   def full_rary_tree(r, n)
176: (4)                """
177: (4)                def _nodes_or_number(n):
178: (8)                    try:
179: (12)                       nodes = list(range(n))
180: (8)                    except TypeError:
181: (12)                       nodes = tuple(n)
182: (8)                    else:
183: (12)                       if n < 0:
184: (16)                           raise nx.NetworkXError(f"Negative number of nodes not valid:
{n}")
185: (8)                    return (n, nodes)
186: (4)                try:
187: (8)                    iter_wa = iter(which_args)
188: (4)                except TypeError:
189: (8)                    iter_wa = (which_args,)
190: (4)                return argmap(_nodes_or_number, *iter_wa)
191: (0)            def np_random_state(random_state_argument):
192: (4)                """Decorator to generate a numpy RandomState or Generator instance.
193: (4)                The decorator processes the argument indicated by `random_state_argument`
194: (4)                using :func:`nx.utils.create_random_state`.
195: (4)                The argument value can be a seed (integer), or a
`numpy.random.RandomState`
196: (4)                or `numpy.random.RandomState` instance or (`None` or `numpy.random`).
197: (4)                The latter two options use the global random number generator for
`numpy.random`.
198: (4)                The returned instance is a `numpy.random.RandomState` or
`numpy.random.Generator`.
199: (4)                Parameters
200: (4)                ----------
201: (4)                random_state_argument : string or int
```

```
(8)                     The name or index of the argument to be converted
(8)                     to a `numpy.random.RandomState` instance.
(4)                 Returns
(4)                 -------
(4)                 _random_state : function
(8)                     Function whose random_state keyword argument is a RandomState
instance.
(4)                 Examples
(4)                 --------
(4)                 Decorate functions like this::
(7)                    @np_random_state("seed")
(7)                    def random_float(seed=None):
(11)                       return seed.rand()
(7)                    @np_random_state(0)
(7)                    def random_float(rng=None):
(11)                       return rng.rand()
(7)                    @np_random_state(1)
(7)                    def random_array(dims, random_state=1):
(11)                       return random_state.rand(*dims)
(4)                 See Also
(4)                 --------
(4)                 py_random_state
(4)                 """
(4)                 return argmap(create_random_state, random_state_argument)
(0)             def py_random_state(random_state_argument):
(4)                 """Decorator to generate a random.Random instance (or equiv).
(4)                 This decorator processes `random_state_argument` using
(4)                 :func:`nx.utils.create_py_random_state`.
(4)                 The input value can be a seed (integer), or a random number generator::
(8)                     If int, return a random.Random instance set with seed=int.
(8)                     If random.Random instance, return it.
(8)                     If None or the `random` package, return the global random number
(8)                     generator used by `random`.
(8)                     If np.random package, or the default numpy RandomState instance,
(8)                     return the default numpy random number generator wrapped in a
(8)                     `PythonRandomViaNumpyBits`  class.
(8)                     If np.random.Generator instance, return it wrapped in a
(8)                     `PythonRandomViaNumpyBits`  class.
(8)                     If np.random.RandomState instance, return it wrapped in a
(8)                     `PythonRandomInterface` class.
(8)                     If a `PythonRandomInterface` instance, return it
(4)                 Parameters
(4)                 ----------
(4)                 random_state_argument : string or int
(8)                     The name of the argument or the index of the argument in args that is
(8)                     to be converted to the random.Random instance or
numpy.random.RandomState
(8)                     instance that mimics basic methods of random.Random.
(4)                 Returns
(4)                 -------
(4)                 _random_state : function
(8)                     Function whose random_state_argument is converted to a Random
instance.
(4)                 Examples
(4)                 --------
(4)                 Decorate functions like this::
(7)                    @py_random_state("random_state")
(7)                    def random_float(random_state=None):
(11)                       return random_state.rand()
(7)                    @py_random_state(0)
(7)                    def random_float(rng=None):
(11)                       return rng.rand()
(7)                    @py_random_state(1)
(7)                    def random_array(dims, seed=12345):
(11)                       return seed.rand(*dims)
(4)                 See Also
(4)                 --------
(4)                 np_random_state
(4)                 """
```

```
    return argmap(create_py_random_state, random_state_argument)
class argmap:
    """A decorator to apply a map to arguments before calling the function
    This class provides a decorator that maps (transforms) arguments of the function
    before the function is called. Thus for example, we have similar code
    in many functions to determine whether an argument is the number of nodes
    to be created, or a list of nodes to be handled. The decorator provides
    the code to accept either -- transforming the indicated argument into a
    list of nodes before the actual function is called.
    This decorator class allows us to process single or multiple arguments.
    The arguments to be processed can be specified by string, naming the argument,
    or by index, specifying the item in the args list.
    Parameters
    ----------
    func : callable
        The function to apply to arguments
    *args : iterable of (int, str or tuple)
        A list of parameters, specified either as strings (their names), ints
        (numerical indices) or tuples, which may contain ints, strings, and
        (recursively) tuples. Each indicates which parameters the decorator
        should map. Tuples indicate that the map function takes (and returns)
        multiple parameters in the same order and nested structure as indicated
        here.
    try_finally : bool (default: False)
        When True, wrap the function call in a try-finally block with code
        for the finally block created by `func`. This is used when the map
        function constructs an object (like a file handle) that requires
        post-processing (like closing).
        Note: try_finally decorators cannot be used to decorate generator
        functions.
    Examples
    --------
    Most of these examples use `@argmap(...)` to apply the decorator to
    the function defined on the next line.
    In the NetworkX codebase however, `argmap` is used within a function to
    construct a decorator. That is, the decorator defines a mapping function
    and then uses `argmap` to build and return a decorated function.
    A simple example is a decorator that specifies which currency to report money.
    The decorator (named `convert_to`) would be used like::
        @convert_to("US_Dollars", "income")
        def show_me_the_money(name, income):
            print(f"{name} : {income}")
    And the code to create the decorator might be::
        def convert_to(currency, which_arg):
            def _convert(amount):
                if amount.currency != currency:
                    amount = amount.to_currency(currency)
                return amount
            return argmap(_convert, which_arg)
    Despite this common idiom for argmap, most of the following examples
    use the `@argmap(...)` idiom to save space.
    Here's an example use of argmap to sum the elements of two of the functions
    arguments. The decorated function::
        @argmap(sum, "xlist", "zlist")
        def foo(xlist, y, zlist):
            return xlist - y + zlist
    is syntactic sugar for::
        def foo(xlist, y, zlist):
            x = sum(xlist)
            z = sum(zlist)
            return x - y + z
    and is equivalent to (using argument indexes)::
        @argmap(sum, "xlist", 2)
        def foo(xlist, y, zlist):
```

```
332: (12)                        return xlist - y + zlist
333: (4)                 or::
334: (8)                     @argmap(sum, "zlist", 0)
335: (8)                     def foo(xlist, y, zlist):
336: (12)                        return xlist - y + zlist
337: (4)                 Transforming functions can be applied to multiple arguments, such as::
338: (8)                     def swap(x, y):
339: (12)                        return y, x
340: (8)                     @argmap(swap, ("a", "b")):
341: (8)                     def foo(a, b, c):
342: (12)                        return a / b * c
343: (4)                 is equivalent to::
344: (8)                     def foo(a, b, c):
345: (12)                        a, b = swap(a, b)
346: (12)                        return a / b * c
347: (4)                 More generally, the applied arguments can be nested tuples of strings or
ints.
348: (4)                 The syntax `@argmap(some_func, ("a", ("b", "c")))` would expect
`some_func` to
349: (4)                 accept 2 inputs with the second expected to be a 2-tuple. It should then
return
350: (4)                 2 outputs with the second a 2-tuple. The returns values would replace
input "a"
351: (4)                 "b" and "c" respectively. Similarly for `@argmap(some_func, (0, ("b",
2)))`.
352: (4)                 Also, note that an index larger than the number of named parameters is
allowed
353: (4)                 for variadic functions. For example::
354: (8)                     def double(a):
355: (12)                        return 2 * a
356: (8)                     @argmap(double, 3)
357: (8)                     def overflow(a, *args):
358: (12)                        return a, args
359: (8)                     print(overflow(1, 2, 3, 4, 5, 6))  # output is 1, (2, 3, 8, 5, 6)
360: (4)                 **Try Finally**
361: (4)                 Additionally, this `argmap` class can be used to create a decorator that
362: (4)                 initiates a try...finally block. The decorator must be written to return
363: (4)                 both the transformed argument and a closing function.
364: (4)                 This feature was included to enable the `open_file` decorator which might
365: (4)                 need to close the file or not depending on whether it had to open that
file.
366: (4)                 This feature uses the keyword-only `try_finally` argument to `@argmap`.
367: (4)                 For example this map opens a file and then makes sure it is closed::
368: (8)                     def open_file(fn):
369: (12)                        f = open(fn)
370: (12)                        return f, lambda: f.close()
371: (4)                 The decorator applies that to the function `foo`::
372: (8)                     @argmap(open_file, "file", try_finally=True)
373: (8)                     def foo(file):
374: (12)                        print(file.read())
375: (4)                 is syntactic sugar for::
376: (8)                     def foo(file):
377: (12)                        file, close_file = open_file(file)
378: (12)                        try:
379: (16)                            print(file.read())
380: (12)                        finally:
381: (16)                            close_file()
382: (4)                 and is equivalent to (using indexes)::
383: (8)                     @argmap(open_file, 0, try_finally=True)
384: (8)                     def foo(file):
385: (12)                        print(file.read())
386: (4)                 Here's an example of the try_finally feature used to create a decorator::
387: (8)                     def my_closing_decorator(which_arg):
388: (12)                        def _opener(path):
389: (16)                            if path is None:
390: (20)                                path = open(path)
391: (20)                                fclose = path.close
392: (16)                            else:
393: (20)                                fclose = lambda: None
```

```
            return path, fclose
        return argmap(_opener, which_arg, try_finally=True)
which can then be used as::
    @my_closing_decorator("file")
    def fancy_reader(file=None):
        print(file.read())
Decorators with try_finally = True cannot be used with generator functions,
because the `finally` block is evaluated before the generator is exhausted::
    @argmap(open_file, "file", try_finally=True)
    def file_to_lines(file):
        for line in file.readlines():
            yield line
is equivalent to::
    def file_to_lines_wrapped(file):
        for line in file.readlines():
            yield line
    def file_to_lines_wrapper(file):
        try:
            file = open_file(file)
            return file_to_lines_wrapped(file)
        finally:
            file.close()
which behaves similarly to::
    def file_to_lines_whoops(file):
        file = open_file(file)
        file.close()
        for line in file.readlines():
            yield line
because the `finally` block of `file_to_lines_wrapper` is executed before
the caller has a chance to exhaust the iterator.
Notes
-----
An object of this class is callable and intended to be used when
defining a decorator. Generally, a decorator takes a function as input
and constructs a function as output. Specifically, an `argmap` object
returns the input function decorated/wrapped so that specified arguments
are mapped (transformed) to new values before the decorated function is called.
As an overview, the argmap object returns a new function with all the
dunder values of the original function (like `__doc__`, `__name__`, etc).
Code for this decorated function is built based on the original function's
signature. It starts by mapping the input arguments to potentially new
values. Then it calls the decorated function with these new values in place
of the indicated arguments that have been mapped. The return value of the
original function is then returned. This new function is the function that
is actually called by the user.
Three additional features are provided.
    1) The code is lazily compiled. That is, the new function is returned
    as an object without the code compiled, but with all information
    needed so it can be compiled upon it's first invocation. This saves
    time on import at the cost of additional time on the first call of
    the function. Subsequent calls are then just as fast as normal.
    2) If the "try_finally" keyword-only argument is True, a try block
    follows each mapped argument, matched on the other side of the wrapped
    call, by a finally block closing that mapping.  We expect func to return
    a 2-tuple: the mapped value and a function to be called in the finally
    clause.  This feature was included so the `open_file` decorator could
    provide a file handle to the decorated function and close the file handle
    after the function call. It even keeps track of whether to close the file
    handle or not based on whether it had to open the file or the input was
    already open. So, the decorated function does not need to include any
    code to open or close files.
```

```
(8)                 3) The maps applied can process multiple arguments. For example,
(8)                 you could swap two arguments using a mapping, or transform
(8)                 them to their sum and their difference. This was included to allow
(8)                 a decorator in the `quality.py` module that checks that an input
(8)                 `partition` is a valid partition of the nodes of the input graph `G`.
(8)                 In this example, the map has inputs `(G, partition)`. After checking
(8)                 for a valid partition, the map either raises an exception or leaves
(8)                 the inputs unchanged. Thus many functions that make this check can
(8)                 use the decorator rather than copy the checking code into each
function.
(8)                 More complicated nested argument structures are described below.
(4)             The remaining notes describe the code structure and methods for this
(4)             class in broad terms to aid in understanding how to use it.
(4)             Instantiating an `argmap` object simply stores the mapping function and
(4)             the input identifiers of which arguments to map. The resulting decorator
(4)             is ready to use this map to decorate any function. Calling that object
(4)             (`argmap.__call__`, but usually done via `@my_decorator`) a lazily
(4)             compiled thin wrapper of the decorated function is constructed,
(4)             wrapped with the necessary function dunder attributes like `__doc__`
(4)             and `__name__`. That thinly wrapped function is returned as the
(4)             decorated function. When that decorated function is called, the thin
(4)             wrapper of code calls `argmap._lazy_compile` which compiles the decorated
(4)             function (using `argmap.compile`) and replaces the code of the thin
(4)             wrapper with the newly compiled code. This saves the compilation step
(4)             every import of networkx, at the cost of compiling upon the first call
(4)             to the decorated function.
(4)             When the decorated function is compiled, the code is recursively assembled
(4)             using the `argmap.assemble` method. The recursive nature is needed in
(4)             case of nested decorators. The result of the assembly is a number of
(4)             useful objects.
(6)               sig : the function signature of the original decorated function as
(10)                  constructed by :func:`argmap.signature`. This is constructed
(10)                  using `inspect.signature` but enhanced with attribute
(10)                  strings `sig_def` and `sig_call`, and other information
(10)                  specific to mapping arguments of this function.
(10)                  This information is used to construct a string of code defining
(10)                  the new decorated function.
(6)               wrapped_name : a unique internally used name constructed by argmap
(10)                  for the decorated function.
(6)               functions : a dict of the functions used inside the code of this
(10)                  decorated function, to be used as `globals` in `exec`.
(10)                  This dict is recursively updated to allow for nested decorating.
(6)               mapblock : code (as a list of strings) to map the incoming argument
(10)                  values to their mapped values.
(6)               finallys : code (as a list of strings) to provide the possibly nested
(10)                  set of finally clauses if needed.
(6)               mutable_args : a bool indicating whether the `sig.args` tuple should be
(10)                  converted to a list so mutation can occur.
(4)             After this recursive assembly process, the `argmap.compile` method
(4)             constructs code (as strings) to convert the tuple `sig.args` to a list
(4)             if needed. It joins the defining code with appropriate indents and
(4)             compiles the result.  Finally, this code is evaluated and the original
(4)             wrapper's implementation is replaced with the compiled version (see
(4)             `argmap._lazy_compile` for more details).
(4)             Other `argmap` methods include `_name` and `_count` which allow internally
(4)             generated names to be unique within a python session.
(4)             The methods `_flatten` and `_indent` process the nested lists of strings
(4)             into properly indented python code ready to be compiled.
(4)             More complicated nested tuples of arguments also allowed though
(4)             usually not used. For the simple 2 argument case, the argmap
(4)             input ("a", "b") implies the mapping function will take 2 arguments
(4)             and return a 2-tuple of mapped values. A more complicated example
(4)             with argmap input `("a", ("b", "c"))` requires the mapping function
(4)             take 2 inputs, with the second being a 2-tuple. It then must output
(4)             the 3 mapped values in the same nested structure `(newa, (newb, newc))`.
(4)             This level of generality is not often needed, but was convenient
(4)             to implement when handling the multiple arguments.
(4)             See Also
(4)             --------
```

```
    not_implemented_for
    open_file
    nodes_or_number
    py_random_state
    networkx.algorithms.community.quality.require_partition
    """
    def __init__(self, func, *args, try_finally=False):
        self._func = func
        self._args = args
        self._finally = try_finally
    @staticmethod
    def _lazy_compile(func):
        """Compile the source of a wrapped function
        Assemble and compile the decorated function, and intrusively replace its
        code with the compiled version's.  The thinly wrapped function becomes
        the decorated function.
        Parameters
        ----------
        func : callable
            A function returned by argmap.__call__ which is in the process
            of being called for the first time.
        Returns
        -------
        func : callable
            The same function, with a new __code__ object.
        Notes
        -----
        It was observed in NetworkX issue #4732 [1] that the import time of
        NetworkX was significantly bloated by the use of decorators: over half
        of the import time was being spent decorating functions.  This was
        somewhat improved by a change made to the `decorator` library, at the
        cost of a relatively heavy-weight call to `inspect.Signature.bind`
        for each call to the decorated function.
        The workaround we arrived at is to do minimal work at the time of
        decoration.  When the decorated function is called for the first time,
        we compile a function with the same function signature as the wrapped
        function.  The resulting decorated function is faster than one made by
        the `decorator` library, so that the overhead of the first call is
        'paid off' after a small number of calls.
        References
        ----------
        [1] https://github.com/networkx/networkx/issues/4732
        """
        real_func = func.__argmap__.compile(func.__wrapped__)
        func.__code__ = real_func.__code__
        func.__globals__.update(real_func.__globals__)
        func.__dict__.update(real_func.__dict__)
        return func
    def __call__(self, f):
        """Construct a lazily decorated wrapper of f.
        The decorated function will be compiled when it is called for the first time,
        and it will replace its own __code__ object so subsequent calls are fast.
        Parameters
        ----------
        f : callable
            A function to be decorated.
        Returns
        -------
        func : callable
            The decorated function.
        See Also
        --------
        argmap._lazy_compile
        """
        def func(*args, __wrapper=None, **kwargs):
            return argmap._lazy_compile(__wrapper)(*args, **kwargs)
```

```
589: (8)                    func.__name__ = f.__name__
590: (8)                    func.__doc__ = f.__doc__
591: (8)                    func.__defaults__ = f.__defaults__
592: (8)                    func.__kwdefaults__.update(f.__kwdefaults__ or {})
593: (8)                    func.__module__ = f.__module__
594: (8)                    func.__qualname__ = f.__qualname__
595: (8)                    func.__dict__.update(f.__dict__)
596: (8)                    func.__wrapped__ = f
597: (8)                    func.__kwdefaults__["_argmap__wrapper"] = func
598: (8)                    func.__self__ = func
599: (8)                    func.__argmap__ = self
600: (8)                    if hasattr(f, "__argmap__"):
601: (12)                       func.__is_generator = f.__is_generator
602: (8)                    else:
603: (12)                       func.__is_generator = inspect.isgeneratorfunction(f)
604: (8)                    if self._finally and func.__is_generator:
605: (12)                       raise nx.NetworkXError("argmap cannot decorate generators with
try_finally")
606: (8)                    return func
607: (4)                __count = 0
608: (4)                @classmethod
609: (4)                def _count(cls):
610: (8)                    """Maintain a globally-unique identifier for function names and "file"
names
611: (8)                    Note that this counter is a class method reporting a class variable
612: (8)                    so the count is unique within a Python session. It could differ from
613: (8)                    session to session for a specific decorator depending on the order
614: (8)                    that the decorators are created. But that doesn't disrupt `argmap`.
615: (8)                    This is used in two places: to construct unique variable names
616: (8)                    in the `_name` method and to construct unique fictitious filenames
617: (8)                    in the `_compile` method.
618: (8)                    Returns
619: (8)                    -------
620: (8)                    count : int
621: (12)                       An integer unique to this Python session (simply counts from zero)
622: (8)                    """
623: (8)                    cls.__count += 1
624: (8)                    return cls.__count
625: (4)                _bad_chars = re.compile("[^a-zA-Z0-9_]")
626: (4)                @classmethod
627: (4)                def _name(cls, f):
628: (8)                    """Mangle the name of a function to be unique but somewhat human-
readable
629: (8)                    The names are unique within a Python session and set using `_count`.
630: (8)                    Parameters
631: (8)                    ----------
632: (8)                    f : str or object
633: (8)                    Returns
634: (8)                    -------
635: (8)                    name : str
636: (12)                       The mangled version of `f.__name__` (if `f.__name__` exists) or
`f`
637: (8)                    """
638: (8)                    f = f.__name__ if hasattr(f, "__name__") else f
639: (8)                    fname = re.sub(cls._bad_chars, "_", f)
640: (8)                    return f"argmap_{fname}_{cls._count()}"
641: (4)                def compile(self, f):
642: (8)                    """Compile the decorated function.
643: (8)                    Called once for a given decorated function -- collects the code from
all
644: (8)                    argmap decorators in the stack, and compiles the decorated function.
645: (8)                    Much of the work done here uses the `assemble` method to allow
recursive
646: (8)                    treatment of multiple argmap decorators on a single decorated
function.
647: (8)                    That flattens the argmap decorators, collects the source code to
construct
648: (8)                    a single decorated function, then compiles/executes/returns that
function.
```

```
(8)                     The source code for the decorated function is stored as an attribute
(8)                     `_code` on the function object itself.
(8)                     Note that Python's `compile` function requires a filename, but this
(8)                     code is constructed without a file, so a fictitious filename is used
(8)                     to describe where the function comes from. The name is something like:
(8)                     "argmap compilation 4".
(8)                     Parameters
(8)                     ----------
(8)                     f : callable
(12)                        The function to be decorated
(8)                     Returns
(8)                     -------
(8)                     func : callable
(12)                        The decorated file
(8)                     """
(8)                     sig, wrapped_name, functions, mapblock, finallys, mutable_args =
self.assemble(
(12)                        f
(8)                     )
(8)                     call = f"{sig.call_sig.format(wrapped_name)}#"
(8)                     mut_args = f"{sig.args} = list({sig.args})" if mutable_args else ""
(8)                     body = argmap._indent(sig.def_sig, mut_args, mapblock, call, finallys)
(8)                     code = "\n".join(body)
(8)                     locl = {}
(8)                     globl = dict(functions.values())
(8)                     filename = f"{self.__class__} compilation {self._count()}"
(8)                     compiled = compile(code, filename, "exec")
(8)                     exec(compiled, globl, locl)
(8)                     func = locl[sig.name]
(8)                     func._code = code
(8)                     return func
(4)                 def assemble(self, f):
(8)                     """Collects components of the source for the decorated function
wrapping f.
(8)                     If `f` has multiple argmap decorators, we recursively assemble the
stack of
(8)                     decorators into a single flattened function.
(8)                     This method is part of the `compile` method's process yet separated
(8)                     from that method to allow recursive processing. The outputs are
(8)                     strings, dictionaries and lists that collect needed info to
(8)                     flatten any nested argmap-decoration.
(8)                     Parameters
(8)                     ----------
(8)                     f : callable
(12)                        The function to be decorated.  If f is argmapped, we assemble it.
(8)                     Returns
(8)                     -------
(8)                     sig : argmap.Signature
(12)                        The function signature as an `argmap.Signature` object.
(8)                     wrapped_name : str
(12)                        The mangled name used to represent the wrapped function in the
code
(12)                        being assembled.
(8)                     functions : dict
(12)                        A dictionary mapping id(g) -> (mangled_name(g), g) for functions g
(12)                        referred to in the code being assembled. These need to be present
(12)                        in the ``globals`` scope of ``exec`` when defining the decorated
(12)                        function.
(8)                     mapblock : list of lists and/or strings
(12)                        Code that implements mapping of parameters including any try
blocks
(12)                        if needed. This code will precede the decorated function call.
(8)                     finallys : list of lists and/or strings
(12)                        Code that implements the finally blocks to post-process the
(12)                        arguments (usually close any files if needed) after the
(12)                        decorated function is called.
(8)                     mutable_args : bool
(12)                        True if the decorator needs to modify positional arguments
(12)                        via their indices. The compile method then turns the argument
```

```
713: (12)                        tuple into a list so that the arguments can be modified.
714: (8)                    """
715: (8)                    if hasattr(f, "__argmap__") and f.__self__ is f:
716: (12)                        (
717: (16)                            sig,
718: (16)                            wrapped_name,
719: (16)                            functions,
720: (16)                            mapblock,
721: (16)                            finallys,
722: (16)                            mutable_args,
723: (12)                        ) = f.__argmap__.assemble(f.__wrapped__)
724: (12)                        functions = dict(functions)  # shallow-copy just in case
725: (8)                    else:
726: (12)                        sig = self.signature(f)
727: (12)                        wrapped_name = self._name(f)
728: (12)                        mapblock, finallys = [], []
729: (12)                        functions = {id(f): (wrapped_name, f)}
730: (12)                        mutable_args = False
731: (8)                    if id(self._func) in functions:
732: (12)                        fname, _ = functions[id(self._func)]
733: (8)                    else:
734: (12)                        fname, _ = functions[id(self._func)] = self._name(self._func),
self._func
735: (8)                    applied = set()
736: (8)                    def get_name(arg, first=True):
737: (12)                        nonlocal mutable_args
738: (12)                        if isinstance(arg, tuple):
739: (16)                            name = ", ".join(get_name(x, False) for x in arg)
740: (16)                            return name if first else f"({name})"
741: (12)                        if arg in applied:
742: (16)                            raise nx.NetworkXError(f"argument {arg} is specified multiple
times")
743: (12)                        applied.add(arg)
744: (12)                        if arg in sig.names:
745: (16)                            return sig.names[arg]
746: (12)                        elif isinstance(arg, str):
747: (16)                            if sig.kwargs is None:
748: (20)                                raise nx.NetworkXError(
749: (24)                                    f"name {arg} is not a named parameter and this
function doesn't have kwargs"
750: (20)                                )
751: (16)                            return f"{sig.kwargs}[{arg!r}]"
752: (12)                        else:
753: (16)                            if sig.args is None:
754: (20)                                raise nx.NetworkXError(
755: (24)                                    f"index {arg} not a parameter index and this function
doesn't have args"
756: (20)                                )
757: (16)                            mutable_args = True
758: (16)                            return f"{sig.args}[{arg - sig.n_positional}]"
759: (8)                    if self._finally:
760: (12)                        for a in self._args:
761: (16)                            name = get_name(a)
762: (16)                            final = self._name(name)
763: (16)                            mapblock.append(f"{name}, {final} = {fname}({name})")
764: (16)                            mapblock.append("try:")
765: (16)                            finallys = ["finally:", f"{final}()#", "#", finallys]
766: (8)                    else:
767: (12)                        mapblock.extend(
768: (16)                            f"{name} = {fname}({name})" for name in map(get_name,
self._args)
769: (12)                        )
770: (8)                    return sig, wrapped_name, functions, mapblock, finallys, mutable_args
771: (4)                @classmethod
772: (4)                def signature(cls, f):
773: (8)                    r"""Construct a Signature object describing `f`
774: (8)                    Compute a Signature so that we can write a function wrapping f with
775: (8)                    the same signature and call-type.
776: (8)                    Parameters
```

```
(8)                     ----------
(8)                     f : callable
(12)                        A function to be decorated
(8)                     Returns
(8)                     -------
(8)                     sig : argmap.Signature
(12)                        The Signature of f
(8)                     Notes
(8)                     -----
(8)                     The Signature is a namedtuple with names:
(12)                        name : a unique version of the name of the decorated function
(12)                        signature : the inspect.signature of the decorated function
(12)                        def_sig : a string used as code to define the new function
(12)                        call_sig : a string used as code to call the decorated function
(12)                        names : a dict keyed by argument name and index to the argument's
name
(12)                        n_positional : the number of positional arguments in the signature
(12)                        args : the name of the VAR_POSITIONAL argument if any, i.e.
\*theseargs
(12)                        kwargs : the name of the VAR_KEYWORDS argument if any, i.e.
\*\*kwargs
(8)                     These named attributes of the signature are used in `assemble` and
`compile`
(8)                     to construct a string of source code for the decorated function.
(8)                     """
(8)                     sig = inspect.signature(f, follow_wrapped=False)
(8)                     def_sig = []
(8)                     call_sig = []
(8)                     names = {}
(8)                     kind = None
(8)                     args = None
(8)                     kwargs = None
(8)                     npos = 0
(8)                     for i, param in enumerate(sig.parameters.values()):
(12)                        prev = kind
(12)                        kind = param.kind
(12)                        if prev == param.POSITIONAL_ONLY != kind:
(16)                            def_sig.append("/")
(12)                        if (
(16)                            param.VAR_POSITIONAL
(16)                            != prev
(16)                            != param.KEYWORD_ONLY
(16)                            == kind
(16)                            != param.VAR_POSITIONAL
(12)                        ):
(16)                            def_sig.append("*")
(12)                        if kind == param.VAR_POSITIONAL:
(16)                            name = "*" + param.name
(16)                            args = param.name
(16)                            count = 0
(12)                        elif kind == param.VAR_KEYWORD:
(16)                            name = "**" + param.name
(16)                            kwargs = param.name
(16)                            count = 0
(12)                        else:
(16)                            names[i] = names[param.name] = param.name
(16)                            name = param.name
(16)                            count = 1
(12)                        if kind == param.KEYWORD_ONLY:
(16)                            call_sig.append(f"{name} = {name}")
(12)                        else:
(16)                            npos += count
(16)                            call_sig.append(name)
(12)                        def_sig.append(name)
(8)                     fname = cls._name(f)
(8)                     def_sig = f'def {fname}({", ".join(def_sig)}):'
(8)                     call_sig = f"return {{}}({', '.join(call_sig)})"
(8)                     return cls.Signature(fname, sig, def_sig, call_sig, names, npos, args,
kwargs)
```

```
    Signature = collections.namedtuple(
        "Signature",
        [
            "name",
            "signature",
            "def_sig",
            "call_sig",
            "names",
            "n_positional",
            "args",
            "kwargs",
        ],
    )
    @staticmethod
    def _flatten(nestlist, visited):
        """flattens a recursive list of lists that doesn't have cyclic references

        Parameters
        ----------
        nestlist : iterable
            A recursive list of objects to be flattened into a single iterable
        visited : set
            A set of object ids which have been walked -- initialize with an
            empty set
        Yields
        ------
        Non-list objects contained in nestlist
        """
        for thing in nestlist:
            if isinstance(thing, list):
                if id(thing) in visited:
                    raise ValueError("A cycle was found in nestlist.  Be a tree.")
                else:
                    visited.add(id(thing))
                yield from argmap._flatten(thing, visited)
            else:
                yield thing
    _tabs = " " * 64
    @staticmethod
    def _indent(*lines):
        """Indent list of code lines to make executable Python code
        Indents a tree-recursive list of strings, following the rule that one
        space is added to the tab after a line that ends in a colon, and one is

        removed after a line that ends in an hashmark.
        Parameters
        ----------
        *lines : lists and/or strings
            A recursive list of strings to be assembled into properly indented
            code.
        Returns
        -------
        code : str
        Examples
        --------
            argmap._indent(*["try:", "try:", "pass#", "finally:", "pass#", "#",
                             "finally:", "pass#"])
        renders to
            '''try:
             try:
              pass#
             finally:
              pass#
            finally:
             pass#'''
        """
        depth = 0
```

```
906: (8)                    for line in argmap._flatten(lines, set()):
907: (12)                       yield f"{argmap._tabs[:depth]}{line}"
908: (12)                       depth += (line[-1:] == ":") - (line[-1:] == "#")
909: (0)            def deprecate_positional_args(func=None, *, version):
910: (4)                """Decorator for methods that issues warnings for positional arguments.
911: (4)                Using the keyword-only argument syntax in pep 3102, arguments after the
912: (4)                * will issue a warning when passed as a positional argument.
913: (4)                Parameters
914: (4)                ----------
915: (4)                func : callable, default=None
916: (8)                    Function to check arguments on.
917: (4)                version : callable, default="1.3"
918: (8)                    The version when positional arguments will result in error.
919: (4)                """
920: (4)                def _inner_deprecate_positional_args(f):
921: (8)                    sig = signature(f)
922: (8)                    kwonly_args = []
923: (8)                    all_args = []
924: (8)                    for name, param in sig.parameters.items():
925: (12)                       if param.kind == Parameter.POSITIONAL_OR_KEYWORD:
926: (16)                           all_args.append(name)
927: (12)                       elif param.kind == Parameter.KEYWORD_ONLY:
928: (16)                           kwonly_args.append(name)
929: (8)                    @wraps(f)
930: (8)                    def inner_f(*args, **kwargs):
931: (12)                       extra_args = len(args) - len(all_args)
932: (12)                       if extra_args <= 0:
933: (16)                           return f(*args, **kwargs)
934: (12)                       args_msg = [
935: (16)                           f"{name}={arg}"
936: (16)                           for name, arg in zip(kwonly_args[:extra_args], args[-
extra_args:])
937: (12)                       ]
938: (12)                       args_msg = ", ".join(args_msg)
939: (12)                       warnings.warn(
940: (16)                           (
941: (20)                               f"Pass {args_msg} as keyword args. From NetworkX version "
942: (20)                               f"{version} passing these as positional arguments "
943: (20)                               "will result in an error"
944: (16)                           ),
945: (16)                           FutureWarning,
946: (12)                       )
947: (12)                       kwargs.update(zip(sig.parameters, args))
948: (12)                       return f(**kwargs)
949: (8)                    return inner_f
950: (4)                if func is not None:
951: (8)                    return _inner_deprecate_positional_args(func)
952: (4)                return _inner_deprecate_positional_args
```

File 550 - heaps.py:

```
1: (0)              """
2: (0)              Min-heaps.
3: (0)              """
4: (0)              from heapq import heappop, heappush
5: (0)              from itertools import count
6: (0)              import networkx as nx
7: (0)              __all__ = ["MinHeap", "PairingHeap", "BinaryHeap"]
8: (0)              class MinHeap:
9: (4)                  """Base class for min-heaps.
10: (4)                 A MinHeap stores a collection of key-value pairs ordered by their values.
11: (4)                 It supports querying the minimum pair, inserting a new pair, decreasing
the
12: (4)                 value in an existing pair and deleting the minimum pair.
13: (4)                 """
14: (4)                 class _Item:
15: (8)                     """Used by subclassess to represent a key-value pair."""
```

```
(8)                     __slots__ = ("key", "value")
(8)                     def __init__(self, key, value):
(12)                        self.key = key
(12)                        self.value = value
(8)                     def __repr__(self):
(12)                        return repr((self.key, self.value))
(4)                 def __init__(self):
(8)                     """Initialize a new min-heap."""
(8)                     self._dict = {}
(4)                 def min(self):
(8)                     """Query the minimum key-value pair.
(8)                     Returns
(8)                     -------
(8)                     key, value : tuple
(12)                        The key-value pair with the minimum value in the heap.
(8)                     Raises
(8)                     ------
(8)                     NetworkXError
(12)                        If the heap is empty.
(8)                     """
(8)                     raise NotImplementedError
(4)                 def pop(self):
(8)                     """Delete the minimum pair in the heap.
(8)                     Returns
(8)                     -------
(8)                     key, value : tuple
(12)                        The key-value pair with the minimum value in the heap.
(8)                     Raises
(8)                     ------
(8)                     NetworkXError
(12)                        If the heap is empty.
(8)                     """
(8)                     raise NotImplementedError
(4)                 def get(self, key, default=None):
(8)                     """Returns the value associated with a key.
(8)                     Parameters
(8)                     ----------
(8)                     key : hashable object
(12)                        The key to be looked up.
(8)                     default : object
(12)                        Default value to return if the key is not present in the heap.
(12)                        Default value: None.
(8)                     Returns
(8)                     -------
(8)                     value : object.
(12)                        The value associated with the key.
(8)                     """
(8)                     raise NotImplementedError
(4)                 def insert(self, key, value, allow_increase=False):
(8)                     """Insert a new key-value pair or modify the value in an existing
(8)                     pair.
(8)                     Parameters
(8)                     ----------
(8)                     key : hashable object
(12)                        The key.
(8)                     value : object comparable with existing values.
(12)                        The value.
(8)                     allow_increase : bool
(12)                        Whether the value is allowed to increase. If False, attempts to
(12)                        increase an existing value have no effect. Default value: False.
(8)                     Returns
(8)                     -------
(8)                     decreased : bool
(12)                        True if a pair is inserted or the existing value is decreased.
(8)                     """
(8)                     raise NotImplementedError
(4)                 def __nonzero__(self):
(8)                     """Returns whether the heap if empty."""
(8)                     return bool(self._dict)
```

```
(4)     def __bool__(self):
(8)         """Returns whether the heap if empty."""
(8)         return bool(self._dict)
(4)     def __len__(self):
(8)         """Returns the number of key-value pairs in the heap."""
(8)         return len(self._dict)
(4)     def __contains__(self, key):
(8)         """Returns whether a key exists in the heap.
(8)         Parameters
(8)         ----------
(8)         key : any hashable object.
(12)            The key to be looked up.
(8)         """
(8)         return key in self._dict
(0) class PairingHeap(MinHeap):
(4)     """A pairing heap."""
(4)     class _Node(MinHeap._Item):
(8)         """A node in a pairing heap.
(8)         A tree in a pairing heap is stored using the left-child, right-sibling
(8)         representation.
(8)         """
(8)         __slots__ = ("left", "next", "prev", "parent")
(8)         def __init__(self, key, value):
(12)            super().__init__(key, value)
(12)            self.left = None
(12)            self.next = None
(12)            self.prev = None
(12)            self.parent = None
(4)     def __init__(self):
(8)         """Initialize a pairing heap."""
(8)         super().__init__()
(8)         self._root = None
(4)     def min(self):
(8)         if self._root is None:
(12)            raise nx.NetworkXError("heap is empty.")
(8)         return (self._root.key, self._root.value)
(4)     def pop(self):
(8)         if self._root is None:
(12)            raise nx.NetworkXError("heap is empty.")
(8)         min_node = self._root
(8)         self._root = self._merge_children(self._root)
(8)         del self._dict[min_node.key]
(8)         return (min_node.key, min_node.value)
(4)     def get(self, key, default=None):
(8)         node = self._dict.get(key)
(8)         return node.value if node is not None else default
(4)     def insert(self, key, value, allow_increase=False):
(8)         node = self._dict.get(key)
(8)         root = self._root
(8)         if node is not None:
(12)            if value < node.value:
(16)                node.value = value
(16)                if node is not root and value < node.parent.value:
(20)                    self._cut(node)
(20)                    self._root = self._link(root, node)
(16)                return True
(12)            elif allow_increase and value > node.value:
(16)                node.value = value
(16)                child = self._merge_children(node)
(16)                if child is not None:
(20)                    self._root = self._link(self._root, child)
(12)            return False
(8)         else:
(12)            node = self._Node(key, value)
(12)            self._dict[key] = node
(12)            self._root = self._link(root, node) if root is not None else node
(12)            return True
(4)     def _link(self, root, other):
(8)         """Link two nodes, making the one with the smaller value the parent of
```

```
154: (8)                    the other.
155: (8)                    """
156: (8)                    if other.value < root.value:
157: (12)                       root, other = other, root
158: (8)                    next = root.left
159: (8)                    other.next = next
160: (8)                    if next is not None:
161: (12)                       next.prev = other
162: (8)                    other.prev = None
163: (8)                    root.left = other
164: (8)                    other.parent = root
165: (8)                    return root
166: (4)                def _merge_children(self, root):
167: (8)                    """Merge the subtrees of the root using the standard two-pass method.
168: (8)                    The resulting subtree is detached from the root.
169: (8)                    """
170: (8)                    node = root.left
171: (8)                    root.left = None
172: (8)                    if node is not None:
173: (12)                       link = self._link
174: (12)                       prev = None
175: (12)                       while True:
176: (16)                           next = node.next
177: (16)                           if next is None:
178: (20)                               node.prev = prev
179: (20)                               break
180: (16)                           next_next = next.next
181: (16)                           node = link(node, next)
182: (16)                           node.prev = prev
183: (16)                           prev = node
184: (16)                           if next_next is None:
185: (20)                               break
186: (16)                           node = next_next
187: (12)                       prev = node.prev
188: (12)                       while prev is not None:
189: (16)                           prev_prev = prev.prev
190: (16)                           node = link(prev, node)
191: (16)                           prev = prev_prev
192: (12)                       node.prev = None
193: (12)                       node.next = None
194: (12)                       node.parent = None
195: (8)                    return node
196: (4)                def _cut(self, node):
197: (8)                    """Cut a node from its parent."""
198: (8)                    prev = node.prev
199: (8)                    next = node.next
200: (8)                    if prev is not None:
201: (12)                       prev.next = next
202: (8)                    else:
203: (12)                       node.parent.left = next
204: (8)                    node.prev = None
205: (8)                    if next is not None:
206: (12)                       next.prev = prev
207: (12)                       node.next = None
208: (8)                    node.parent = None
209: (0)            class BinaryHeap(MinHeap):
210: (4)                """A binary heap."""
211: (4)                def __init__(self):
212: (8)                    """Initialize a binary heap."""
213: (8)                    super().__init__()
214: (8)                    self._heap = []
215: (8)                    self._count = count()
216: (4)                def min(self):
217: (8)                    dict = self._dict
218: (8)                    if not dict:
219: (12)                       raise nx.NetworkXError("heap is empty")
220: (8)                    heap = self._heap
221: (8)                    pop = heappop
222: (8)                    while True:
```

```
223: (12)                        value, _, key = heap[0]
224: (12)                        if key in dict and value == dict[key]:
225: (16)                            break
226: (12)                        pop(heap)
227: (8)                    return (key, value)
228: (4)                def pop(self):
229: (8)                    dict = self._dict
230: (8)                    if not dict:
231: (12)                       raise nx.NetworkXError("heap is empty")
232: (8)                    heap = self._heap
233: (8)                    pop = heappop
234: (8)                    while True:
235: (12)                       value, _, key = heap[0]
236: (12)                       pop(heap)
237: (12)                       if key in dict and value == dict[key]:
238: (16)                           break
239: (8)                    del dict[key]
240: (8)                    return (key, value)
241: (4)                def get(self, key, default=None):
242: (8)                    return self._dict.get(key, default)
243: (4)                def insert(self, key, value, allow_increase=False):
244: (8)                    dict = self._dict
245: (8)                    if key in dict:
246: (12)                       old_value = dict[key]
247: (12)                       if value < old_value or (allow_increase and value > old_value):
248: (16)                           dict[key] = value
249: (16)                           heappush(self._heap, (value, next(self._count), key))
250: (16)                           return value < old_value
251: (12)                       return False
252: (8)                    else:
253: (12)                       dict[key] = value
254: (12)                       heappush(self._heap, (value, next(self._count), key))
255: (12)                       return True
```

File 551 - mapped_queue.py:

```
1: (0)              """Priority queue class with updatable priorities.
2: (0)              """
3: (0)              import heapq
4: (0)              __all__ = ["MappedQueue"]
5: (0)              class _HeapElement:
6: (4)                  """This proxy class separates the heap element from its priority.
7: (4)                  The idea is that using a 2-tuple (priority, element) works
8: (4)                  for sorting, but not for dict lookup because priorities are
9: (4)                  often floating point values so round-off can mess up equality.
10: (4)                 So, we need inequalities to look at the priority (for sorting)
11: (4)                 and equality (and hash) to look at the element to enable
12: (4)                 updates to the priority.
13: (4)                 Unfortunately, this class can be tricky to work with if you forget that
14: (4)                 `__lt__` compares the priority while `__eq__` compares the element.
15: (4)                 In `greedy_modularity_communities()` the following code is
16: (4)                 used to check that two _HeapElements differ in either element or priority:
17: (8)                     if d_oldmax != row_max or d_oldmax.priority != row_max.priority:
18: (4)                 If the priorities are the same, this implementation uses the element
19: (4)                 as a tiebreaker. This provides compatibility with older systems that
20: (4)                 use tuples to combine priority and elements.
21: (4)                 """
22: (4)                 __slots__ = ["priority", "element", "_hash"]
23: (4)                 def __init__(self, priority, element):
24: (8)                     self.priority = priority
25: (8)                     self.element = element
26: (8)                     self._hash = hash(element)
27: (4)                 def __lt__(self, other):
28: (8)                     try:
29: (12)                        other_priority = other.priority
30: (8)                     except AttributeError:
31: (12)                        return self.priority < other
```

```
32: (8)                     if self.priority == other_priority:
33: (12)                        try:
34: (16)                            return self.element < other.element
35: (12)                        except TypeError as err:
36: (16)                            raise TypeError(
37: (20)                                "Consider using a tuple, with a priority value that can be
compared."
38: (16)                            )
39: (8)                     return self.priority < other_priority
40: (4)                 def __gt__(self, other):
41: (8)                     try:
42: (12)                        other_priority = other.priority
43: (8)                     except AttributeError:
44: (12)                        return self.priority > other
45: (8)                     if self.priority == other_priority:
46: (12)                        try:
47: (16)                            return self.element > other.element
48: (12)                        except TypeError as err:
49: (16)                            raise TypeError(
50: (20)                                "Consider using a tuple, with a priority value that can be
compared."
51: (16)                            )
52: (8)                     return self.priority > other_priority
53: (4)                 def __eq__(self, other):
54: (8)                     try:
55: (12)                        return self.element == other.element
56: (8)                     except AttributeError:
57: (12)                        return self.element == other
58: (4)                 def __hash__(self):
59: (8)                     return self._hash
60: (4)                 def __getitem__(self, indx):
61: (8)                     return self.priority if indx == 0 else self.element[indx - 1]
62: (4)                 def __iter__(self):
63: (8)                     yield self.priority
64: (8)                     try:
65: (12)                        yield from self.element
66: (8)                     except TypeError:
67: (12)                        yield self.element
68: (4)                 def __repr__(self):
69: (8)                     return f"_HeapElement({self.priority}, {self.element})"
70: (0)             class MappedQueue:
71: (4)                 """The MappedQueue class implements a min-heap with removal and update-
priority.
72: (4)                 The min heap uses heapq as well as custom written _siftup and _siftdown
73: (4)                 methods to allow the heap positions to be tracked by an additional dict
74: (4)                 keyed by element to position. The smallest element can be popped in O(1)
time,
75: (4)                 new elements can be pushed in O(log n) time, and any element can be
removed
76: (4)                 or updated in O(log n) time. The queue cannot contain duplicate elements
77: (4)                 and an attempt to push an element already in the queue will have no
effect.
78: (4)                 MappedQueue complements the heapq package from the python standard
79: (4)                 library. While MappedQueue is designed for maximum compatibility with
80: (4)                 heapq, it adds element removal, lookup, and priority update.
81: (4)                 Parameters
82: (4)                 ----------
83: (4)                 data : dict or iterable
84: (4)                 Examples
85: (4)                 --------
86: (4)                 A `MappedQueue` can be created empty, or optionally, given a dictionary
87: (4)                 of initial elements and priorities.  The methods `push`, `pop`,
88: (4)                 `remove`, and `update` operate on the queue.
89: (4)                 >>> colors_nm = {"red": 665, "blue": 470, "green": 550}
90: (4)                 >>> q = MappedQueue(colors_nm)
91: (4)                 >>> q.remove("red")
92: (4)                 >>> q.update("green", "violet", 400)
93: (4)                 >>> q.push("indigo", 425)
94: (4)                 True
```

```
95: (4)                 >>> [q.pop().element for i in range(len(q.heap))]
96: (4)                 ['violet', 'indigo', 'blue']
97: (4)                 A `MappedQueue` can also be initialized with a list or other iterable. The
priority is assumed
98: (4)                 to be the sort order of the items in the list.
99: (4)                 >>> q = MappedQueue([916, 50, 4609, 493, 237])
100: (4)                >>> q.remove(493)
101: (4)                >>> q.update(237, 1117)
102: (4)                >>> [q.pop() for i in range(len(q.heap))]
103: (4)                [50, 916, 1117, 4609]
104: (4)                An exception is raised if the elements are not comparable.
105: (4)                >>> q = MappedQueue([100, "a"])
106: (4)                Traceback (most recent call last):
107: (4)                ...
108: (4)                TypeError: '<' not supported between instances of 'int' and 'str'
109: (4)                To avoid the exception, use a dictionary to assign priorities to the
elements.
110: (4)                >>> q = MappedQueue({100: 0, "a": 1})
111: (4)                References
112: (4)                ----------
113: (4)                .. [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001).
114: (7)                   Introduction to algorithms second edition.
115: (4)                .. [2] Knuth, D. E. (1997). The art of computer programming (Vol. 3).
116: (7)                   Pearson Education.
117: (4)                """
118: (4)                def __init__(self, data=None):
119: (8)                    """Priority queue class with updatable priorities."""
120: (8)                    if data is None:
121: (12)                       self.heap = []
122: (8)                    elif isinstance(data, dict):
123: (12)                       self.heap = [_HeapElement(v, k) for k, v in data.items()]
124: (8)                    else:
125: (12)                       self.heap = list(data)
126: (8)                    self.position = {}
127: (8)                    self._heapify()
128: (4)                def _heapify(self):
129: (8)                    """Restore heap invariant and recalculate map."""
130: (8)                    heapq.heapify(self.heap)
131: (8)                    self.position = {elt: pos for pos, elt in enumerate(self.heap)}
132: (8)                    if len(self.heap) != len(self.position):
133: (12)                       raise AssertionError("Heap contains duplicate elements")
134: (4)                def __len__(self):
135: (8)                    return len(self.heap)
136: (4)                def push(self, elt, priority=None):
137: (8)                    """Add an element to the queue."""
138: (8)                    if priority is not None:
139: (12)                       elt = _HeapElement(priority, elt)
140: (8)                    if elt in self.position:
141: (12)                       return False
142: (8)                    pos = len(self.heap)
143: (8)                    self.heap.append(elt)
144: (8)                    self.position[elt] = pos
145: (8)                    self._siftdown(0, pos)
146: (8)                    return True
147: (4)                def pop(self):
148: (8)                    """Remove and return the smallest element in the queue."""
149: (8)                    elt = self.heap[0]
150: (8)                    del self.position[elt]
151: (8)                    if len(self.heap) == 1:
152: (12)                       self.heap.pop()
153: (12)                       return elt
154: (8)                    last = self.heap.pop()
155: (8)                    self.heap[0] = last
156: (8)                    self.position[last] = 0
157: (8)                    self._siftup(0)
158: (8)                    return elt
159: (4)                def update(self, elt, new, priority=None):
160: (8)                    """Replace an element in the queue with a new one."""
161: (8)                    if priority is not None:
```

```
(12)                        new = _HeapElement(priority, new)
(8)                     pos = self.position[elt]
(8)                     self.heap[pos] = new
(8)                     del self.position[elt]
(8)                     self.position[new] = pos
(8)                     self._siftup(pos)
(4)                 def remove(self, elt):
(8)                     """Remove an element from the queue."""
(8)                     try:
(12)                        pos = self.position[elt]
(12)                        del self.position[elt]
(8)                     except KeyError:
(12)                        raise
(8)                     if pos == len(self.heap) - 1:
(12)                        self.heap.pop()
(12)                        return
(8)                     last = self.heap.pop()
(8)                     self.heap[pos] = last
(8)                     self.position[last] = pos
(8)                     self._siftup(pos)
(4)                 def _siftup(self, pos):
(8)                     """Move smaller child up until hitting a leaf.
(8)                     Built to mimic code for heapq._siftup
(8)                     only updating position dict too.
(8)                     """
(8)                     heap, position = self.heap, self.position
(8)                     end_pos = len(heap)
(8)                     startpos = pos
(8)                     newitem = heap[pos]
(8)                     child_pos = (pos << 1) + 1  # start with leftmost child position
(8)                     while child_pos < end_pos:
(12)                        child = heap[child_pos]
(12)                        right_pos = child_pos + 1
(12)                        if right_pos < end_pos:
(16)                            right = heap[right_pos]
(16)                            if not child < right:
(20)                                child = right
(20)                                child_pos = right_pos
(12)                        heap[pos] = child
(12)                        position[child] = pos
(12)                        pos = child_pos
(12)                        child_pos = (pos << 1) + 1
(8)                     while pos > 0:
(12)                        parent_pos = (pos - 1) >> 1
(12)                        parent = heap[parent_pos]
(12)                        if not newitem < parent:
(16)                            break
(12)                        heap[pos] = parent
(12)                        position[parent] = pos
(12)                        pos = parent_pos
(8)                     heap[pos] = newitem
(8)                     position[newitem] = pos
(4)                 def _siftdown(self, start_pos, pos):
(8)                     """Restore invariant. keep swapping with parent until smaller.
(8)                     Built to mimic code for heapq._siftdown
(8)                     only updating position dict too.
(8)                     """
(8)                     heap, position = self.heap, self.position
(8)                     newitem = heap[pos]
(8)                     while pos > start_pos:
(12)                        parent_pos = (pos - 1) >> 1
(12)                        parent = heap[parent_pos]
(12)                        if not newitem < parent:
(16)                            break
(12)                        heap[pos] = parent
(12)                        position[parent] = pos
(12)                        pos = parent_pos
(8)                     heap[pos] = newitem
(8)                     position[newitem] = pos
```

```
----------------------------------------

File 552 - misc.py:

1: (0)              """
2: (0)              Miscellaneous Helpers for NetworkX.
3: (0)              These are not imported into the base networkx namespace but
4: (0)              can be accessed, for example, as
5: (0)              >>> import networkx
6: (0)              >>> networkx.utils.make_list_of_ints({1, 2, 3})
7: (0)              [1, 2, 3]
8: (0)              >>> networkx.utils.arbitrary_element({5, 1, 7})  # doctest: +SKIP
9: (0)              1
10: (0)             """
11: (0)             import random
12: (0)             import sys
13: (0)             import uuid
14: (0)             import warnings
15: (0)             from collections import defaultdict, deque
16: (0)             from collections.abc import Iterable, Iterator, Sized
17: (0)             from itertools import chain, tee
18: (0)             import networkx as nx
19: (0)             __all__ = [
20: (4)                 "flatten",
21: (4)                 "make_list_of_ints",
22: (4)                 "dict_to_numpy_array",
23: (4)                 "arbitrary_element",
24: (4)                 "pairwise",
25: (4)                 "groups",
26: (4)                 "create_random_state",
27: (4)                 "create_py_random_state",
28: (4)                 "PythonRandomInterface",
29: (4)                 "PythonRandomViaNumpyBits",
30: (4)                 "nodes_equal",
31: (4)                 "edges_equal",
32: (4)                 "graphs_equal",
33: (4)                 "_clear_cache",
34: (0)             ]
35: (0)             def flatten(obj, result=None):
36: (4)                 """Return flattened version of (possibly nested) iterable object."""
37: (4)                 if not isinstance(obj, Iterable | Sized) or isinstance(obj, str):
38: (8)                     return obj
39: (4)                 if result is None:
40: (8)                     result = []
41: (4)                 for item in obj:
42: (8)                     if not isinstance(item, Iterable | Sized) or isinstance(item, str):
43: (12)                        result.append(item)
44: (8)                     else:
45: (12)                        flatten(item, result)
46: (4)                 return tuple(result)
47: (0)             def make_list_of_ints(sequence):
48: (4)                 """Return list of ints from sequence of integral numbers.
49: (4)                 All elements of the sequence must satisfy int(element) == element
50: (4)                 or a ValueError is raised. Sequence is iterated through once.
51: (4)                 If sequence is a list, the non-int values are replaced with ints.
52: (4)                 So, no new list is created
53: (4)                 """
54: (4)                 if not isinstance(sequence, list):
55: (8)                     result = []
56: (8)                     for i in sequence:
57: (12)                        errmsg = f"sequence is not all integers: {i}"
58: (12)                        try:
59: (16)                            ii = int(i)
60: (12)                        except ValueError:
61: (16)                            raise nx.NetworkXError(errmsg) from None
62: (12)                        if ii != i:
63: (16)                            raise nx.NetworkXError(errmsg)
64: (12)                        result.append(ii)
```

```
 65: (8)                    return result
 66: (4)                for indx, i in enumerate(sequence):
 67: (8)                    errmsg = f"sequence is not all integers: {i}"
 68: (8)                    if isinstance(i, int):
 69: (12)                       continue
 70: (8)                    try:
 71: (12)                       ii = int(i)
 72: (8)                    except ValueError:
 73: (12)                       raise nx.NetworkXError(errmsg) from None
 74: (8)                    if ii != i:
 75: (12)                       raise nx.NetworkXError(errmsg)
 76: (8)                    sequence[indx] = ii
 77: (4)                return sequence
 78: (0)            def dict_to_numpy_array(d, mapping=None):
 79: (4)                """Convert a dictionary of dictionaries to a numpy array
 80: (4)                with optional mapping."""
 81: (4)                try:
 82: (8)                    return _dict_to_numpy_array2(d, mapping)
 83: (4)                except (AttributeError, TypeError):
 84: (8)                    return _dict_to_numpy_array1(d, mapping)
 85: (0)            def _dict_to_numpy_array2(d, mapping=None):
 86: (4)                """Convert a dictionary of dictionaries to a 2d numpy array
 87: (4)                with optional mapping.
 88: (4)                """
 89: (4)                import numpy as np
 90: (4)                if mapping is None:
 91: (8)                    s = set(d.keys())
 92: (8)                    for k, v in d.items():
 93: (12)                       s.update(v.keys())
 94: (8)                    mapping = dict(zip(s, range(len(s))))
 95: (4)                n = len(mapping)
 96: (4)                a = np.zeros((n, n))
 97: (4)                for k1, i in mapping.items():
 98: (8)                    for k2, j in mapping.items():
 99: (12)                       try:
 100: (16)                          a[i, j] = d[k1][k2]
 101: (12)                      except KeyError:
 102: (16)                          pass
 103: (4)               return a
 104: (0)           def _dict_to_numpy_array1(d, mapping=None):
 105: (4)               """Convert a dictionary of numbers to a 1d numpy array with optional
mapping."""
 106: (4)               import numpy as np
 107: (4)               if mapping is None:
 108: (8)                   s = set(d.keys())
 109: (8)                   mapping = dict(zip(s, range(len(s))))
 110: (4)               n = len(mapping)
 111: (4)               a = np.zeros(n)
 112: (4)               for k1, i in mapping.items():
 113: (8)                   i = mapping[k1]
 114: (8)                   a[i] = d[k1]
 115: (4)               return a
 116: (0)           def arbitrary_element(iterable):
 117: (4)               """Returns an arbitrary element of `iterable` without removing it.
 118: (4)               This is most useful for "peeking" at an arbitrary element of a set,
 119: (4)               but can be used for any list, dictionary, etc., as well.
 120: (4)               Parameters
 121: (4)               ----------
 122: (4)               iterable : `abc.collections.Iterable` instance
 123: (8)                   Any object that implements ``__iter__``, e.g. set, dict, list, tuple,
 124: (8)                   etc.
 125: (4)               Returns
 126: (4)               -------
 127: (4)               The object that results from ``next(iter(iterable))``
 128: (4)               Raises
 129: (4)               ------
 130: (4)               ValueError
 131: (8)                   If `iterable` is an iterator (because the current implementation of
 132: (8)                   this function would consume an element from the iterator).
```

```
(4)                 Examples
(4)                 --------
(4)                 Arbitrary elements from common Iterable objects:
(4)                 >>> nx.utils.arbitrary_element([1, 2, 3])  # list
(4)                 1
(4)                 >>> nx.utils.arbitrary_element((1, 2, 3))  # tuple
(4)                 1
(4)                 >>> nx.utils.arbitrary_element({1, 2, 3})  # set
(4)                 1
(4)                 >>> d = {k: v for k, v in zip([1, 2, 3], [3, 2, 1])}
(4)                 >>> nx.utils.arbitrary_element(d)  # dict_keys
(4)                 1
(4)                 >>> nx.utils.arbitrary_element(d.values())  # dict values
(4)                 3
(4)                 `str` is also an Iterable:
(4)                 >>> nx.utils.arbitrary_element("hello")
(4)                 'h'
(4)                 :exc:`ValueError` is raised if `iterable` is an iterator:
(4)                 >>> iterator = iter([1, 2, 3])  # Iterator, *not* Iterable
(4)                 >>> nx.utils.arbitrary_element(iterator)
(4)                 Traceback (most recent call last):
(8)                     ...
(4)                 ValueError: cannot return an arbitrary item from an iterator
(4)                 Notes
(4)                 -----
(4)                 This function does not return a *random* element. If `iterable` is
(4)                 ordered, sequential calls will return the same value::
(8)                     >>> l = [1, 2, 3]
(8)                     >>> nx.utils.arbitrary_element(l)
(8)                     1
(8)                     >>> nx.utils.arbitrary_element(l)
(8)                     1
(4)                 """
(4)                 if isinstance(iterable, Iterator):
(8)                     raise ValueError("cannot return an arbitrary item from an iterator")
(4)                 return next(iter(iterable))
(0)             def pairwise(iterable, cyclic=False):
(4)                 "s -> (s0, s1), (s1, s2), (s2, s3), ..."
(4)                 a, b = tee(iterable)
(4)                 first = next(b, None)
(4)                 if cyclic is True:
(8)                     return zip(a, chain(b, (first,)))
(4)                 return zip(a, b)
(0)             def groups(many_to_one):
(4)                 """Converts a many-to-one mapping into a one-to-many mapping.
(4)                 `many_to_one` must be a dictionary whose keys and values are all
(4)                 :term:`hashable`.
(4)                 The return value is a dictionary mapping values from `many_to_one`
(4)                 to sets of keys from `many_to_one` that have that value.
(4)                 Examples
(4)                 --------
(4)                 >>> from networkx.utils import groups
(4)                 >>> many_to_one = {"a": 1, "b": 1, "c": 2, "d": 3, "e": 3}
(4)                 >>> groups(many_to_one)  # doctest: +SKIP
(4)                 {1: {'a', 'b'}, 2: {'c'}, 3: {'e', 'd'}}
(4)                 """
(4)                 one_to_many = defaultdict(set)
(4)                 for v, k in many_to_one.items():
(8)                     one_to_many[k].add(v)
(4)                 return dict(one_to_many)
(0)             def create_random_state(random_state=None):
(4)                 """Returns a numpy.random.RandomState or numpy.random.Generator instance
(4)                 depending on input.
(4)                 Parameters
(4)                 ----------
(4)                 random_state : int or NumPy RandomState or Generator instance, optional
(default=None)
(8)                     If int, return a numpy.random.RandomState instance set with seed=int.
(8)                     if `numpy.random.RandomState` instance, return it.
```

```
201: (8)                    if `numpy.random.Generator` instance, return it.
202: (8)                    if None or numpy.random, return the global random number generator
used
203: (8)                    by numpy.random.
204: (4)                """
205: (4)                import numpy as np
206: (4)                if random_state is None or random_state is np.random:
207: (8)                    return np.random.mtrand._rand
208: (4)                if isinstance(random_state, np.random.RandomState):
209: (8)                    return random_state
210: (4)                if isinstance(random_state, int):
211: (8)                    return np.random.RandomState(random_state)
212: (4)                if isinstance(random_state, np.random.Generator):
213: (8)                    return random_state
214: (4)                msg = (
215: (8)                    f"{random_state} cannot be used to create a numpy.random.RandomState
or\n"
216: (8)                    "numpy.random.Generator instance"
217: (4)                )
218: (4)                raise ValueError(msg)
219: (0)            class PythonRandomViaNumpyBits(random.Random):
220: (4)                """Provide the random.random algorithms using a numpy.random bit generator
221: (4)                The intent is to allow people to contribute code that uses Python's random
222: (4)                library, but still allow users to provide a single easily controlled
random
223: (4)                bit-stream for all work with NetworkX. This implementation is based on
helpful
224: (4)                comments and code from Robert Kern on NumPy's GitHub Issue #24458.
225: (4)                This implementation supercedes that of `PythonRandomInterface` which
rewrote
226: (4)                methods to account for subtle differences in API between `random` and
227: (4)                `numpy.random`. Instead this subclasses `random.Random` and overwrites
228: (4)                the methods `random`, `getrandbits`, `getstate`, `setstate` and `seed`.
229: (4)                It makes them use the rng values from an input numpy `RandomState` or
`Generator`.
230: (4)                Those few methods allow the rest of the `random.Random` methods to provide
231: (4)                the API interface of `random.random` while using randomness generated by
232: (4)                a numpy generator.
233: (4)                """
234: (4)                def __init__(self, rng=None):
235: (8)                    try:
236: (12)                       import numpy as np
237: (8)                    except ImportError:
238: (12)                       msg = "numpy not found, only random.random available."
239: (12)                       warnings.warn(msg, ImportWarning)
240: (8)                    if rng is None:
241: (12)                       self._rng = np.random.mtrand._rand
242: (8)                    else:
243: (12)                       self._rng = rng
244: (8)                    self.gauss_next = None
245: (4)                def random(self):
246: (8)                    """Get the next random number in the range 0.0 <= X < 1.0."""
247: (8)                    return self._rng.random()
248: (4)                def getrandbits(self, k):
249: (8)                    """getrandbits(k) -> x.  Generates an int with k random bits."""
250: (8)                    if k < 0:
251: (12)                       raise ValueError("number of bits must be non-negative")
252: (8)                    numbytes = (k + 7) // 8  # bits / 8 and rounded up
253: (8)                    x = int.from_bytes(self._rng.bytes(numbytes), "big")
254: (8)                    return x >> (numbytes * 8 - k)  # trim excess bits
255: (4)                def getstate(self):
256: (8)                    return self._rng.__getstate__()
257: (4)                def setstate(self, state):
258: (8)                    self._rng.__setstate__(state)
259: (4)                def seed(self, *args, **kwds):
260: (8)                    "Do nothing override method."
261: (8)                    raise NotImplementedError("seed() not implemented in
PythonRandomViaNumpyBits")
262: (0)            class PythonRandomInterface:
```

```
263: (4)                """PythonRandomInterface is included for backward compatibility
264: (4)                New code should use PythonRandomViaNumpyBits instead.
265: (4)                """
266: (4)                def __init__(self, rng=None):
267: (8)                    try:
268: (12)                       import numpy as np
269: (8)                    except ImportError:
270: (12)                       msg = "numpy not found, only random.random available."
271: (12)                       warnings.warn(msg, ImportWarning)
272: (8)                    if rng is None:
273: (12)                       self._rng = np.random.mtrand._rand
274: (8)                    else:
275: (12)                       self._rng = rng
276: (4)                def random(self):
277: (8)                    return self._rng.random()
278: (4)                def uniform(self, a, b):
279: (8)                    return a + (b - a) * self._rng.random()
280: (4)                def randrange(self, a, b=None):
281: (8)                    import numpy as np
282: (8)                    if b is None:
283: (12)                       a, b = 0, a
284: (8)                    if b > 9223372036854775807:  # from np.iinfo(np.int64).max
285: (12)                       tmp_rng = PythonRandomViaNumpyBits(self._rng)
286: (12)                       return tmp_rng.randrange(a, b)
287: (8)                    if isinstance(self._rng, np.random.Generator):
288: (12)                       return self._rng.integers(a, b)
289: (8)                    return self._rng.randint(a, b)
290: (4)                def choice(self, seq):
291: (8)                    import numpy as np
292: (8)                    if isinstance(self._rng, np.random.Generator):
293: (12)                       idx = self._rng.integers(0, len(seq))
294: (8)                    else:
295: (12)                       idx = self._rng.randint(0, len(seq))
296: (8)                    return seq[idx]
297: (4)                def gauss(self, mu, sigma):
298: (8)                    return self._rng.normal(mu, sigma)
299: (4)                def shuffle(self, seq):
300: (8)                    return self._rng.shuffle(seq)
301: (4)                def sample(self, seq, k):
302: (8)                    return self._rng.choice(list(seq), size=(k,), replace=False)
303: (4)                def randint(self, a, b):
304: (8)                    import numpy as np
305: (8)                    if b > 9223372036854775807:  # from np.iinfo(np.int64).max
306: (12)                       tmp_rng = PythonRandomViaNumpyBits(self._rng)
307: (12)                       return tmp_rng.randint(a, b)
308: (8)                    if isinstance(self._rng, np.random.Generator):
309: (12)                       return self._rng.integers(a, b + 1)
310: (8)                    return self._rng.randint(a, b + 1)
311: (4)                def expovariate(self, scale):
312: (8)                    return self._rng.exponential(1 / scale)
313: (4)                def paretovariate(self, shape):
314: (8)                    return self._rng.pareto(shape)
315: (0)            def create_py_random_state(random_state=None):
316: (4)                """Returns a random.Random instance depending on input.
317: (4)                Parameters
318: (4)                ----------
319: (4)                random_state : int or random number generator or None (default=None)
320: (8)                    - If int, return a `random.Random` instance set with seed=int.
321: (8)                    - If `random.Random` instance, return it.
322: (8)                    - If None or the `np.random` package, return the global random number
323: (10)                     generator used by `np.random`.
324: (8)                    - If an `np.random.Generator` instance, or the `np.random` package, or
325: (10)                     the global numpy random number generator, then return it.
326: (10)                     wrapped in a `PythonRandomViaNumpyBits` class.
327: (8)                    - If a `PythonRandomViaNumpyBits` instance, return it.
328: (8)                    - If a `PythonRandomInterface` instance, return it.
329: (8)                    - If a `np.random.RandomState` instance and not the global numpy
default,
330: (10)                     return it wrapped in `PythonRandomInterface` for backward bit-stream
```

```
(10)                    matching with legacy code.
(4)                Notes
(4)                -----
(4)                - A diagram intending to illustrate the relationships behind our support
(6)                  for numpy random numbers is called
(6)                  `NetworkX Numpy Random Numbers
<https://excalidraw.com/#room=b5303f2b03d3af7ccc6a,e5ZDIWdWWCTTsg8OqoRvPA>`_.
(4)                - More discussion about this support also appears in
(6)                  `gh-6869#comment
<https://github.com/networkx/networkx/pull/6869#issuecomment-1944799534>`_.
(4)                - Wrappers of numpy.random number generators allow them to mimic the
Python random
(6)                  number generation algorithms. For example, Python can create arbitrarily
large
(6)                  random ints, and the wrappers use Numpy bit-streams with CPython's
random module
(6)                  to choose arbitrarily large random integers too.
(4)                - We provide two wrapper classes:
(6)                  `PythonRandomViaNumpyBits` is usually what you want and is always used
for
(6)                  `np.Generator` instances. But for users who need to recreate random
numbers
(6)                  produced in NetworkX 3.2 or earlier, we maintain the
`PythonRandomInterface`
(6)                  wrapper as well. We use it only used if passed a (non-default)
`np.RandomState`
(6)                  instance pre-initialized from a seed. Otherwise the newer wrapper is
used.
(4)                """
(4)                if random_state is None or random_state is random:
(8)                    return random._inst
(4)                if isinstance(random_state, random.Random):
(8)                    return random_state
(4)                if isinstance(random_state, int):
(8)                    return random.Random(random_state)
(4)                try:
(8)                    import numpy as np
(4)                except ImportError:
(8)                    pass
(4)                else:
(8)                    if isinstance(random_state, PythonRandomInterface |
PythonRandomViaNumpyBits):
(12)                       return random_state
(8)                    if isinstance(random_state, np.random.Generator):
(12)                       return PythonRandomViaNumpyBits(random_state)
(8)                    if random_state is np.random:
(12)                       return PythonRandomViaNumpyBits(np.random.mtrand._rand)
(8)                    if isinstance(random_state, np.random.RandomState):
(12)                       if random_state is np.random.mtrand._rand:
(16)                           return PythonRandomViaNumpyBits(random_state)
(12)                       return PythonRandomInterface(random_state)
(4)                msg = f"{random_state} cannot be used to generate a random.Random
instance"
(4)                raise ValueError(msg)
(0)            def nodes_equal(nodes1, nodes2):
(4)                """Check if nodes are equal.
(4)                Equality here means equal as Python objects.
(4)                Node data must match if included.
(4)                The order of nodes is not relevant.
(4)                Parameters
(4)                ----------
(4)                nodes1, nodes2 : iterables of nodes, or (node, datadict) tuples
(4)                Returns
(4)                -------
(4)                bool
(8)                    True if nodes are equal, False otherwise.
(4)                """
(4)                nlist1 = list(nodes1)
(4)                nlist2 = list(nodes2)
```

```
(4)                 try:
(8)                     d1 = dict(nlist1)
(8)                     d2 = dict(nlist2)
(4)                 except (ValueError, TypeError):
(8)                     d1 = dict.fromkeys(nlist1)
(8)                     d2 = dict.fromkeys(nlist2)
(4)                 return d1 == d2
(0)             def edges_equal(edges1, edges2):
(4)                 """Check if edges are equal.
(4)                 Equality here means equal as Python objects.
(4)                 Edge data must match if included.
(4)                 The order of the edges is not relevant.
(4)                 Parameters
(4)                 ----------
(4)                 edges1, edges2 : iterables of with u, v nodes as
(8)                     edge tuples (u, v), or
(8)                     edge tuples with data dicts (u, v, d), or
(8)                     edge tuples with keys and data dicts (u, v, k, d)
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     True if edges are equal, False otherwise.
(4)                 """
(4)                 from collections import defaultdict
(4)                 d1 = defaultdict(dict)
(4)                 d2 = defaultdict(dict)
(4)                 c1 = 0
(4)                 for c1, e in enumerate(edges1):
(8)                     u, v = e[0], e[1]
(8)                     data = [e[2:]]
(8)                     if v in d1[u]:
(12)                        data = d1[u][v] + data
(8)                     d1[u][v] = data
(8)                     d1[v][u] = data
(4)                 c2 = 0
(4)                 for c2, e in enumerate(edges2):
(8)                     u, v = e[0], e[1]
(8)                     data = [e[2:]]
(8)                     if v in d2[u]:
(12)                        data = d2[u][v] + data
(8)                     d2[u][v] = data
(8)                     d2[v][u] = data
(4)                 if c1 != c2:
(8)                     return False
(4)                 for n, nbrdict in d1.items():
(8)                     for nbr, datalist in nbrdict.items():
(12)                        if n not in d2:
(16)                            return False
(12)                        if nbr not in d2[n]:
(16)                            return False
(12)                        d2datalist = d2[n][nbr]
(12)                        for data in datalist:
(16)                            if datalist.count(data) != d2datalist.count(data):
(20)                                return False
(4)                 return True
(0)             def graphs_equal(graph1, graph2):
(4)                 """Check if graphs are equal.
(4)                 Equality here means equal as Python objects (not isomorphism).
(4)                 Node, edge and graph data must match.
(4)                 Parameters
(4)                 ----------
(4)                 graph1, graph2 : graph
(4)                 Returns
(4)                 -------
(4)                 bool
(8)                     True if graphs are equal, False otherwise.
(4)                 """
(4)                 return (
(8)                     graph1.adj == graph2.adj
```

```
457: (8)                    and graph1.nodes == graph2.nodes
458: (8)                    and graph1.graph == graph2.graph
459: (4)                )
460: (0)            def _clear_cache(G):
461: (4)                """Clear the cache of a graph (currently stores converted graphs).
462: (4)                Caching is controlled via ``nx.config.cache_converted_graphs``
configuration.
463: (4)                """
464: (4)                if cache := getattr(G, "__networkx_cache__", None):
465: (8)                    cache.clear()
```

File 553 - rcm.py:

```
1: (0)              """
2: (0)              Cuthill-McKee ordering of graph nodes to produce sparse matrices
3: (0)              """
4: (0)              from collections import deque
5: (0)              from operator import itemgetter
6: (0)              import networkx as nx
7: (0)              from ..utils import arbitrary_element
8: (0)              __all__ = ["cuthill_mckee_ordering", "reverse_cuthill_mckee_ordering"]
9: (0)              def cuthill_mckee_ordering(G, heuristic=None):
10: (4)                 """Generate an ordering (permutation) of the graph nodes to make
11: (4)                 a sparse matrix.
12: (4)                 Uses the Cuthill-McKee heuristic (based on breadth-first search) [1]_.
13: (4)                 Parameters
14: (4)                 ----------
15: (4)                 G : graph
16: (6)                   A NetworkX graph
17: (4)                 heuristic : function, optional
18: (6)                   Function to choose starting node for RCM algorithm.  If None
19: (6)                   a node from a pseudo-peripheral pair is used.  A user-defined function
20: (6)                   can be supplied that takes a graph object and returns a single node.
21: (4)                 Returns
22: (4)                 -------
23: (4)                 nodes : generator
24: (7)                    Generator of nodes in Cuthill-McKee ordering.
25: (4)                 Examples
26: (4)                 --------
27: (4)                 >>> from networkx.utils import cuthill_mckee_ordering
28: (4)                 >>> G = nx.path_graph(4)
29: (4)                 >>> rcm = list(cuthill_mckee_ordering(G))
30: (4)                 >>> A = nx.adjacency_matrix(G, nodelist=rcm)
31: (4)                 Smallest degree node as heuristic function:
32: (4)                 >>> def smallest_degree(G):
33: (4)                 ...     return min(G, key=G.degree)
34: (4)                 >>> rcm = list(cuthill_mckee_ordering(G, heuristic=smallest_degree))
35: (4)                 See Also
36: (4)                 --------
37: (4)                 reverse_cuthill_mckee_ordering
38: (4)                 Notes
39: (4)                 -----
40: (4)                 The optimal solution the bandwidth reduction is NP-complete [2]_.
41: (4)                 References
42: (4)                 ----------
43: (4)                 .. [1] E. Cuthill and J. McKee.
44: (7)                    Reducing the bandwidth of sparse symmetric matrices,
45: (7)                    In Proc. 24th Nat. Conf. ACM, pages 157-172, 1969.
46: (7)                    http://doi.acm.org/10.1145/800195.805928
47: (4)                 .. [2]  Steven S. Skiena. 1997. The Algorithm Design Manual.
48: (7)                    Springer-Verlag New York, Inc., New York, NY, USA.
49: (4)                 """
50: (4)                 for c in nx.connected_components(G):
51: (8)                     yield from connected_cuthill_mckee_ordering(G.subgraph(c), heuristic)
52: (0)             def reverse_cuthill_mckee_ordering(G, heuristic=None):
53: (4)                 """Generate an ordering (permutation) of the graph nodes to make
54: (4)                 a sparse matrix.
```

```
 55: (4)                Uses the reverse Cuthill-McKee heuristic (based on breadth-first search)
 56: (4)                [1]_.
 57: (4)                Parameters
 58: (4)                ----------
 59: (4)                G : graph
 60: (6)                  A NetworkX graph
 61: (4)                heuristic : function, optional
 62: (6)                  Function to choose starting node for RCM algorithm.  If None
 63: (6)                  a node from a pseudo-peripheral pair is used.  A user-defined function
 64: (6)                  can be supplied that takes a graph object and returns a single node.
 65: (4)                Returns
 66: (4)                -------
 67: (4)                nodes : generator
 68: (7)                   Generator of nodes in reverse Cuthill-McKee ordering.
 69: (4)                Examples
 70: (4)                --------
 71: (4)                >>> from networkx.utils import reverse_cuthill_mckee_ordering
 72: (4)                >>> G = nx.path_graph(4)
 73: (4)                >>> rcm = list(reverse_cuthill_mckee_ordering(G))
 74: (4)                >>> A = nx.adjacency_matrix(G, nodelist=rcm)
 75: (4)                Smallest degree node as heuristic function:
 76: (4)                >>> def smallest_degree(G):
 77: (4)                ...     return min(G, key=G.degree)
 78: (4)                >>> rcm = list(reverse_cuthill_mckee_ordering(G,
heuristic=smallest_degree))
 79: (4)                See Also
 80: (4)                --------
 81: (4)                cuthill_mckee_ordering
 82: (4)                Notes
 83: (4)                -----
 84: (4)                The optimal solution the bandwidth reduction is NP-complete [2]_.
 85: (4)                References
 86: (4)                ----------
 87: (4)                .. [1] E. Cuthill and J. McKee.
 88: (7)                   Reducing the bandwidth of sparse symmetric matrices,
 89: (7)                   In Proc. 24th Nat. Conf. ACM, pages 157-72, 1969.
 90: (7)                   http://doi.acm.org/10.1145/800195.805928
 91: (4)                .. [2]  Steven S. Skiena. 1997. The Algorithm Design Manual.
 92: (7)                   Springer-Verlag New York, Inc., New York, NY, USA.
 93: (4)                """
 94: (4)                return reversed(list(cuthill_mckee_ordering(G, heuristic=heuristic)))
 95: (0)            def connected_cuthill_mckee_ordering(G, heuristic=None):
 96: (4)                if heuristic is None:
 97: (8)                    start = pseudo_peripheral_node(G)
 98: (4)                else:
 99: (8)                    start = heuristic(G)
100: (4)                visited = {start}
101: (4)                queue = deque([start])
102: (4)                while queue:
103: (8)                    parent = queue.popleft()
104: (8)                    yield parent
105: (8)                    nd = sorted(G.degree(set(G[parent]) - visited), key=itemgetter(1))
106: (8)                    children = [n for n, d in nd]
107: (8)                    visited.update(children)
108: (8)                    queue.extend(children)
109: (0)            def pseudo_peripheral_node(G):
110: (4)                u = arbitrary_element(G)
111: (4)                lp = 0
112: (4)                v = u
113: (4)                while True:
114: (8)                    spl = dict(nx.shortest_path_length(G, v))
115: (8)                    l = max(spl.values())
116: (8)                    if l <= lp:
117: (12)                       break
118: (8)                    lp = l
119: (8)                    farthest = (n for n, dist in spl.items() if dist == l)
120: (8)                    v, deg = min(G.degree(farthest), key=itemgetter(1))
121: (4)                return v
```

```
----------------------------------------

File 554 - random_sequence.py:

1: (0)              """
2: (0)              Utilities for generating random numbers, random sequences, and
3: (0)              random selections.
4: (0)              """
5: (0)              import networkx as nx
6: (0)              from networkx.utils import py_random_state
7: (0)              __all__ = [
8: (4)                  "powerlaw_sequence",
9: (4)                  "zipf_rv",
10: (4)                 "cumulative_distribution",
11: (4)                 "discrete_sequence",
12: (4)                 "random_weighted_sample",
13: (4)                 "weighted_choice",
14: (0)             ]
15: (0)             @py_random_state(2)
16: (0)             def powerlaw_sequence(n, exponent=2.0, seed=None):
17: (4)                 """
18: (4)                 Return sample sequence of length n from a power law distribution.
19: (4)                 """
20: (4)                 return [seed.paretovariate(exponent - 1) for i in range(n)]
21: (0)             @py_random_state(2)
22: (0)             def zipf_rv(alpha, xmin=1, seed=None):
23: (4)                 r"""Returns a random value chosen from the Zipf distribution.
24: (4)                 The return value is an integer drawn from the probability distribution
25: (4)                 .. math::
26: (8)                     p(x)=\frac{x^{-\alpha}}{\zeta(\alpha, x_{\min})},
27: (4)                 where $\zeta(\alpha, x_{\min})$ is the Hurwitz zeta function.
28: (4)                 Parameters
29: (4)                 ----------
30: (4)                 alpha : float
31: (6)                   Exponent value of the distribution
32: (4)                 xmin : int
33: (6)                   Minimum value
34: (4)                 seed : integer, random_state, or None (default)
35: (8)                     Indicator of random number generation state.
36: (8)                     See :ref:`Randomness<randomness>`.
37: (4)                 Returns
38: (4)                 -------
39: (4)                 x : int
40: (6)                   Random value from Zipf distribution
41: (4)                 Raises
42: (4)                 ------
43: (4)                 ValueError:
44: (6)                   If xmin < 1 or
45: (6)                   If alpha <= 1
46: (4)                 Notes
47: (4)                 -----
48: (4)                 The rejection algorithm generates random values for a the power-law
49: (4)                 distribution in uniformly bounded expected time dependent on
50: (4)                 parameters.  See [1]_ for details on its operation.
51: (4)                 Examples
52: (4)                 --------
53: (4)                 >>> nx.utils.zipf_rv(alpha=2, xmin=3, seed=42)
54: (4)                 8
55: (4)                 References
56: (4)                 ----------
57: (4)                 .. [1] Luc Devroye, Non-Uniform Random Variate Generation,
58: (7)                    Springer-Verlag, New York, 1986.
59: (4)                 """
60: (4)                 if xmin < 1:
61: (8)                     raise ValueError("xmin < 1")
62: (4)                 if alpha <= 1:
63: (8)                     raise ValueError("a <= 1.0")
64: (4)                 a1 = alpha - 1.0
65: (4)                 b = 2**a1
```

```
66: (4)                 while True:
67: (8)                     u = 1.0 - seed.random()  # u in (0,1]
68: (8)                     v = seed.random()  # v in [0,1)
69: (8)                     x = int(xmin * u ** -(1.0 / a1))
70: (8)                     t = (1.0 + (1.0 / x)) ** a1
71: (8)                     if v * x * (t - 1.0) / (b - 1.0) <= t / b:
72: (12)                        break
73: (4)                 return x
74: (0)             def cumulative_distribution(distribution):
75: (4)                 """Returns normalized cumulative distribution from discrete
distribution."""
76: (4)                 cdf = [0.0]
77: (4)                 psum = sum(distribution)
78: (4)                 for i in range(len(distribution)):
79: (8)                     cdf.append(cdf[i] + distribution[i] / psum)
80: (4)                 return cdf
81: (0)             @py_random_state(3)
82: (0)             def discrete_sequence(n, distribution=None, cdistribution=None, seed=None):
83: (4)                 """
84: (4)                 Return sample sequence of length n from a given discrete distribution
85: (4)                 or discrete cumulative distribution.
86: (4)                 One of the following must be specified.
87: (4)                 distribution = histogram of values, will be normalized
88: (4)                 cdistribution = normalized discrete cumulative distribution
89: (4)                 """
90: (4)                 import bisect
91: (4)                 if cdistribution is not None:
92: (8)                     cdf = cdistribution
93: (4)                 elif distribution is not None:
94: (8)                     cdf = cumulative_distribution(distribution)
95: (4)                 else:
96: (8)                     raise nx.NetworkXError(
97: (12)                        "discrete_sequence: distribution or cdistribution missing"
98: (8)                     )
99: (4)                 inputseq = [seed.random() for i in range(n)]
100: (4)                seq = [bisect.bisect_left(cdf, s) - 1 for s in inputseq]
101: (4)                return seq
102: (0)            @py_random_state(2)
103: (0)            def random_weighted_sample(mapping, k, seed=None):
104: (4)                """Returns k items without replacement from a weighted sample.
105: (4)                The input is a dictionary of items with weights as values.
106: (4)                """
107: (4)                if k > len(mapping):
108: (8)                    raise ValueError("sample larger than population")
109: (4)                sample = set()
110: (4)                while len(sample) < k:
111: (8)                    sample.add(weighted_choice(mapping, seed))
112: (4)                return list(sample)
113: (0)            @py_random_state(1)
114: (0)            def weighted_choice(mapping, seed=None):
115: (4)                """Returns a single element from a weighted sample.
116: (4)                The input is a dictionary of items with weights as values.
117: (4)                """
118: (4)                rnd = seed.random() * sum(mapping.values())
119: (4)                for k, w in mapping.items():
120: (8)                    rnd -= w
121: (8)                    if rnd < 0:
122: (12)                       return k

----------------------------------------

File 555 - __init__.py:

1: (0)

----------------------------------------

File 556 - union_find.py:
```

```
1: (0)              """
2: (0)              Union-find data structure.
3: (0)              """
4: (0)              from networkx.utils import groups
5: (0)              class UnionFind:
6: (4)                  """Union-find data structure.
7: (4)                  Each unionFind instance X maintains a family of disjoint sets of
8: (4)                  hashable objects, supporting the following two methods:
9: (4)                  - X[item] returns a name for the set containing the given item.
10: (6)                   Each set is named by an arbitrarily-chosen one of its members; as
11: (6)                   long as the set remains unchanged it will keep the same name. If
12: (6)                   the item is not yet part of a set in X, a new singleton set is
13: (6)                   created for it.
14: (4)                 - X.union(item1, item2, ...) merges the sets containing each item
15: (6)                   into a single larger set.  If any item is not yet part of a set
16: (6)                   in X, it is added to X as one of the members of the merged set.
17: (6)                   Union-find data structure. Based on Josiah Carlson's code,
18: (6)                   https://code.activestate.com/recipes/215912/
19: (6)                   with significant additional changes by D. Eppstein.
20: (6)                   http://www.ics.uci.edu/~eppstein/PADS/UnionFind.py
21: (4)                 """
22: (4)                 def __init__(self, elements=None):
23: (8)                     """Create a new empty union-find structure.
24: (8)                     If *elements* is an iterable, this structure will be initialized
25: (8)                     with the discrete partition on the given set of elements.
26: (8)                     """
27: (8)                     if elements is None:
28: (12)                        elements = ()
29: (8)                     self.parents = {}
30: (8)                     self.weights = {}
31: (8)                     for x in elements:
32: (12)                        self.weights[x] = 1
33: (12)                        self.parents[x] = x
34: (4)                 def __getitem__(self, object):
35: (8)                     """Find and return the name of the set containing the object."""
36: (8)                     if object not in self.parents:
37: (12)                        self.parents[object] = object
38: (12)                        self.weights[object] = 1
39: (12)                        return object
40: (8)                     path = []
41: (8)                     root = self.parents[object]
42: (8)                     while root != object:
43: (12)                        path.append(object)
44: (12)                        object = root
45: (12)                        root = self.parents[object]
46: (8)                     for ancestor in path:
47: (12)                        self.parents[ancestor] = root
48: (8)                     return root
49: (4)                 def __iter__(self):
50: (8)                     """Iterate through all items ever found or unioned by this
structure."""
51: (8)                     return iter(self.parents)
52: (4)                 def to_sets(self):
53: (8)                     """Iterates over the sets stored in this structure.
54: (8)                     For example::
55: (12)                        >>> partition = UnionFind("xyz")
56: (12)                        >>> sorted(map(sorted, partition.to_sets()))
57: (12)                        [['x'], ['y'], ['z']]
58: (12)                        >>> partition.union("x", "y")
59: (12)                        >>> sorted(map(sorted, partition.to_sets()))
60: (12)                        [['x', 'y'], ['z']]
61: (8)                     """
62: (8)                     for x in self.parents:
63: (12)                        _ = self[x]  # Evaluated for side-effect only
64: (8)                     yield from groups(self.parents).values()
65: (4)                 def union(self, *objects):
66: (8)                     """Find the sets containing the objects and merge them all."""
67: (8)                     roots = iter(
68: (12)                        sorted(
```

```
69: (16)                            {self[x] for x in objects}, key=lambda r: self.weights[r],
reverse=True
70: (12)                        )
71: (8)                     )
72: (8)                     try:
73: (12)                        root = next(roots)
74: (8)                     except StopIteration:
75: (12)                        return
76: (8)                     for r in roots:
77: (12)                        self.weights[root] += self.weights[r]
78: (12)                        self.parents[r] = root
```

File 557 - test__init.py:

```
1: (0)              import pytest
2: (0)              def test_utils_namespace():
3: (4)                  """Ensure objects are not unintentionally exposed in utils namespace."""
4: (4)                  with pytest.raises(ImportError):
5: (8)                      from networkx.utils import nx
6: (4)                  with pytest.raises(ImportError):
7: (8)                      from networkx.utils import sys
8: (4)                  with pytest.raises(ImportError):
9: (8)                      from networkx.utils import defaultdict, deque
```

File 558 - test_backends.py:

```
1: (0)              import pickle
2: (0)              import pytest
3: (0)              import networkx as nx
4: (0)              sp = pytest.importorskip("scipy")
5: (0)              pytest.importorskip("numpy")
6: (0)              def test_dispatch_kwds_vs_args():
7: (4)                  G = nx.path_graph(4)
8: (4)                  nx.pagerank(G)
9: (4)                  nx.pagerank(G=G)
10: (4)                 with pytest.raises(TypeError):
11: (8)                     nx.pagerank()
12: (0)             def test_pickle():
13: (4)                 count = 0
14: (4)                 for name, func in nx.utils.backends._registered_algorithms.items():
15: (8)                     try:
16: (12)                        pickled = pickle.dumps(func)
17: (8)                     except pickle.PicklingError:
18: (12)                        continue
19: (8)                     assert pickle.loads(pickled) is func
20: (8)                     count += 1
21: (4)                 assert count > 0
22: (4)                 assert pickle.loads(pickle.dumps(nx.inverse_line_graph)) is
nx.inverse_line_graph
23: (0)             @pytest.mark.skipif(
24: (4)                 "not nx.config['backend_priority'] "
25: (4)                 "or nx.config['backend_priority'][0] != 'nx-loopback'"
26: (0)             )
27: (0)             def test_graph_converter_needs_backend():
28: (4)                 from networkx.classes.tests.dispatch_interface import (
29: (8)                     LoopbackDispatcher,
30: (8)                     LoopbackGraph,
31: (4)                 )
32: (4)                 A = sp.sparse.coo_array([[0, 3, 2], [3, 0, 1], [2, 1, 0]])
33: (4)                 side_effects = []
34: (4)                 def from_scipy_sparse_array(self, *args, **kwargs):
35: (8)                     side_effects.append(1)  # Just to prove this was called
36: (8)                     return self.convert_from_nx(
37: (12)                        self.__getattr__("from_scipy_sparse_array")(*args, **kwargs),
38: (12)                        preserve_edge_attrs=True,
```

```
39: (12)                        preserve_node_attrs=True,
40: (12)                        preserve_graph_attrs=True,
41: (8)                     )
42: (4)                 @staticmethod
43: (4)                 def convert_to_nx(obj, *, name=None):
44: (8)                     if type(obj) is nx.Graph:
45: (12)                        return obj
46: (8)                     return nx.Graph(obj)
47: (4)                 orig_convert_to_nx = LoopbackDispatcher.convert_to_nx
48: (4)                 LoopbackDispatcher.convert_to_nx = convert_to_nx
49: (4)                 LoopbackDispatcher.from_scipy_sparse_array = from_scipy_sparse_array
50: (4)                 try:
51: (8)                     assert side_effects == []
52: (8)                     assert type(nx.from_scipy_sparse_array(A)) is nx.Graph
53: (8)                     assert side_effects == [1]
54: (8)                     assert (
55: (12)                        type(nx.from_scipy_sparse_array(A, backend="nx-loopback")) is
LoopbackGraph
56: (8)                     )
57: (8)                     assert side_effects == [1, 1]
58: (4)                 finally:
59: (8)                     LoopbackDispatcher.convert_to_nx = staticmethod(orig_convert_to_nx)
60: (8)                     del LoopbackDispatcher.from_scipy_sparse_array
61: (4)                 with pytest.raises(ImportError, match="Unable to load"):
62: (8)                     nx.from_scipy_sparse_array(A, backend="bad-backend-name")
63: (0)             def test_dispatchable_are_functions():
64: (4)                 assert type(nx.pagerank) is type(nx.pagerank.orig_func)
```

File 559 - test_config.py:

```
1: (0)              import collections
2: (0)              import pickle
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.utils.configs import Config
6: (0)              class ExampleConfig(Config):
7: (4)                  """Example configuration."""
8: (4)                  x: int
9: (4)                  y: str
10: (4)                 def _check_config(self, key, value):
11: (8)                     if key == "x" and value <= 0:
12: (12)                        raise ValueError("x must be positive")
13: (8)                     if key == "y" and not isinstance(value, str):
14: (12)                        raise TypeError("y must be a str")
15: (0)             class EmptyConfig(Config):
16: (4)                 pass
17: (0)             @pytest.mark.parametrize("cfg", [EmptyConfig(), Config()])
18: (0)             def test_config_empty(cfg):
19: (4)                 assert dir(cfg) == []
20: (4)                 with pytest.raises(AttributeError):
21: (8)                     cfg.x = 1
22: (4)                 with pytest.raises(KeyError):
23: (8)                     cfg["x"] = 1
24: (4)                 with pytest.raises(AttributeError):
25: (8)                     cfg.x
26: (4)                 with pytest.raises(KeyError):
27: (8)                     cfg["x"]
28: (4)                 assert len(cfg) == 0
29: (4)                 assert "x" not in cfg
30: (4)                 assert cfg == cfg
31: (4)                 assert cfg.get("x", 2) == 2
32: (4)                 assert set(cfg.keys()) == set()
33: (4)                 assert set(cfg.values()) == set()
34: (4)                 assert set(cfg.items()) == set()
35: (4)                 cfg2 = pickle.loads(pickle.dumps(cfg))
36: (4)                 assert cfg == cfg2
37: (4)                 assert isinstance(cfg, collections.abc.Collection)
```

```
 38: (4)                 assert isinstance(cfg, collections.abc.Mapping)
 39: (0)             def test_config_subclass():
 40: (4)                 with pytest.raises(TypeError, match="missing 2 required keyword-only"):
 41: (8)                     ExampleConfig()
 42: (4)                 with pytest.raises(ValueError, match="x must be positive"):
 43: (8)                     ExampleConfig(x=0, y="foo")
 44: (4)                 with pytest.raises(TypeError, match="unexpected keyword"):
 45: (8)                     ExampleConfig(x=1, y="foo", z="bad config")
 46: (4)                 with pytest.raises(TypeError, match="unexpected keyword"):
 47: (8)                     EmptyConfig(z="bad config")
 48: (4)                 cfg = ExampleConfig(x=1, y="foo")
 49: (4)                 assert cfg.x == 1
 50: (4)                 assert cfg["x"] == 1
 51: (4)                 assert cfg["y"] == "foo"
 52: (4)                 assert cfg.y == "foo"
 53: (4)                 assert "x" in cfg
 54: (4)                 assert "y" in cfg
 55: (4)                 assert "z" not in cfg
 56: (4)                 assert len(cfg) == 2
 57: (4)                 assert set(iter(cfg)) == {"x", "y"}
 58: (4)                 assert set(cfg.keys()) == {"x", "y"}
 59: (4)                 assert set(cfg.values()) == {1, "foo"}
 60: (4)                 assert set(cfg.items()) == {("x", 1), ("y", "foo")}
 61: (4)                 assert dir(cfg) == ["x", "y"]
 62: (4)                 cfg.x = 2
 63: (4)                 cfg["y"] = "bar"
 64: (4)                 assert cfg["x"] == 2
 65: (4)                 assert cfg.y == "bar"
 66: (4)                 with pytest.raises(TypeError, match="can't be deleted"):
 67: (8)                     del cfg.x
 68: (4)                 with pytest.raises(TypeError, match="can't be deleted"):
 69: (8)                     del cfg["y"]
 70: (4)                 assert cfg.x == 2
 71: (4)                 assert cfg == cfg
 72: (4)                 assert cfg == ExampleConfig(x=2, y="bar")
 73: (4)                 assert cfg != ExampleConfig(x=3, y="baz")
 74: (4)                 assert cfg != Config(x=2, y="bar")
 75: (4)                 with pytest.raises(TypeError, match="y must be a str"):
 76: (8)                     cfg["y"] = 5
 77: (4)                 with pytest.raises(ValueError, match="x must be positive"):
 78: (8)                     cfg.x = -5
 79: (4)                 assert cfg.get("x", 10) == 2
 80: (4)                 with pytest.raises(AttributeError):
 81: (8)                     cfg.z = 5
 82: (4)                 with pytest.raises(KeyError):
 83: (8)                     cfg["z"] = 5
 84: (4)                 with pytest.raises(AttributeError):
 85: (8)                     cfg.z
 86: (4)                 with pytest.raises(KeyError):
 87: (8)                     cfg["z"]
 88: (4)                 cfg2 = pickle.loads(pickle.dumps(cfg))
 89: (4)                 assert cfg == cfg2
 90: (4)                 assert cfg.__doc__ == "Example configuration."
 91: (4)                 assert cfg2.__doc__ == "Example configuration."
 92: (0)             def test_config_defaults():
 93: (4)                 class DefaultConfig(Config):
 94: (8)                     x: int = 0
 95: (8)                     y: int
 96: (4)                 cfg = DefaultConfig(y=1)
 97: (4)                 assert cfg.x == 0
 98: (4)                 cfg = DefaultConfig(x=2, y=1)
 99: (4)                 assert cfg.x == 2
100: (0)             def test_nxconfig():
101: (4)                 assert isinstance(nx.config.backend_priority, list)
102: (4)                 assert isinstance(nx.config.backends, Config)
103: (4)                 with pytest.raises(TypeError, match="must be a list of backend names"):
104: (8)                     nx.config.backend_priority = "nx_loopback"
105: (4)                 with pytest.raises(ValueError, match="Unknown backend when setting"):
106: (8)                     nx.config.backend_priority =
```

```
["this_almost_certainly_is_not_a_backend"]
107: (4)                with pytest.raises(TypeError, match="must be a Config of backend
configs"):
108: (8)                    nx.config.backends = {}
109: (4)                with pytest.raises(TypeError, match="must be a Config of backend
configs"):
110: (8)                    nx.config.backends = Config(plausible_backend_name={})
111: (4)                with pytest.raises(ValueError, match="Unknown backend when setting"):
112: (8)                    nx.config.backends =
Config(this_almost_certainly_is_not_a_backend=Config())
113: (4)                with pytest.raises(TypeError, match="must be True or False"):
114: (8)                    nx.config.cache_converted_graphs = "bad value"
115: (0)            def test_not_strict():
116: (4)                class FlexibleConfig(Config, strict=False):
117: (8)                    x: int
118: (4)                cfg = FlexibleConfig(x=1)
119: (4)                assert "_strict" not in cfg
120: (4)                assert len(cfg) == 1
121: (4)                assert list(cfg) == ["x"]
122: (4)                assert list(cfg.keys()) == ["x"]
123: (4)                assert list(cfg.values()) == [1]
124: (4)                assert list(cfg.items()) == [("x", 1)]
125: (4)                assert cfg.x == 1
126: (4)                assert cfg["x"] == 1
127: (4)                assert "x" in cfg
128: (4)                assert hasattr(cfg, "x")
129: (4)                assert "FlexibleConfig(x=1)" in repr(cfg)
130: (4)                assert cfg == FlexibleConfig(x=1)
131: (4)                del cfg.x
132: (4)                assert "FlexibleConfig()" in repr(cfg)
133: (4)                assert len(cfg) == 0
134: (4)                assert not hasattr(cfg, "x")
135: (4)                assert "x" not in cfg
136: (4)                assert not hasattr(cfg, "y")
137: (4)                assert "y" not in cfg
138: (4)                cfg.y = 2
139: (4)                assert len(cfg) == 1
140: (4)                assert list(cfg) == ["y"]
141: (4)                assert list(cfg.keys()) == ["y"]
142: (4)                assert list(cfg.values()) == [2]
143: (4)                assert list(cfg.items()) == [("y", 2)]
144: (4)                assert cfg.y == 2
145: (4)                assert cfg["y"] == 2
146: (4)                assert hasattr(cfg, "y")
147: (4)                assert "y" in cfg
148: (4)                del cfg["y"]
149: (4)                assert len(cfg) == 0
150: (4)                assert list(cfg) == []
151: (4)                with pytest.raises(AttributeError, match="y"):
152: (8)                    del cfg.y
153: (4)                with pytest.raises(KeyError, match="y"):
154: (8)                    del cfg["y"]
155: (4)                with pytest.raises(TypeError, match="missing 1 required keyword-only"):
156: (8)                    FlexibleConfig()
157: (4)                with pytest.raises(TypeError, match="unexpected keyword argument 'y'"):
158: (8)                    FlexibleConfig(x=1, y=2)
159: (4)                class FlexibleConfigWithDefault(Config, strict=False):
160: (8)                    x: int = 0
161: (4)                assert FlexibleConfigWithDefault().x == 0
162: (4)                assert FlexibleConfigWithDefault(x=1)["x"] == 1

----------------------------------------

File 560 - test_decorators.py:

1: (0)              import os
2: (0)              import pathlib
3: (0)              import random
4: (0)              import tempfile
```

```
import pytest
import networkx as nx
from networkx.utils.decorators import (
    argmap,
    not_implemented_for,
    np_random_state,
    open_file,
    py_random_state,
)
from networkx.utils.misc import PythonRandomInterface,
PythonRandomViaNumpyBits
def test_not_implemented_decorator():
    @not_implemented_for("directed")
    def test_d(G):
        pass
    test_d(nx.Graph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_d(nx.DiGraph())
    @not_implemented_for("undirected")
    def test_u(G):
        pass
    test_u(nx.DiGraph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_u(nx.Graph())
    @not_implemented_for("multigraph")
    def test_m(G):
        pass
    test_m(nx.Graph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_m(nx.MultiGraph())
    @not_implemented_for("graph")
    def test_g(G):
        pass
    test_g(nx.MultiGraph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_g(nx.Graph())
    @not_implemented_for("directed", "multigraph")
    def test_not_md(G):
        pass
    test_not_md(nx.Graph())
    test_not_md(nx.DiGraph())
    test_not_md(nx.MultiGraph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_not_md(nx.MultiDiGraph())
    @not_implemented_for("directed")
    @not_implemented_for("multigraph")
    def test_graph_only(G):
        pass
    test_graph_only(nx.Graph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_graph_only(nx.DiGraph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_graph_only(nx.MultiGraph())
    with pytest.raises(nx.NetworkXNotImplemented):
        test_graph_only(nx.MultiDiGraph())
    with pytest.raises(ValueError):
        not_implemented_for("directed", "undirected")
    with pytest.raises(ValueError):
        not_implemented_for("multigraph", "graph")
def test_not_implemented_decorator_key():
    with pytest.raises(KeyError):
        @not_implemented_for("foo")
        def test1(G):
            pass
        test1(nx.Graph())
def test_not_implemented_decorator_raise():
    with pytest.raises(nx.NetworkXNotImplemented):
        @not_implemented_for("graph")
        def test1(G):
```

```
 73: (12)                        pass
 74: (8)                     test1(nx.Graph())
 75: (0)             class TestOpenFileDecorator:
 76: (4)                 def setup_method(self):
 77: (8)                     self.text = ["Blah... ", "BLAH ", "BLAH!!!!"]
 78: (8)                     self.fobj = tempfile.NamedTemporaryFile("wb+", delete=False)
 79: (8)                     self.name = self.fobj.name
 80: (4)                 def teardown_method(self):
 81: (8)                     self.fobj.close()
 82: (8)                     os.unlink(self.name)
 83: (4)                 def write(self, path):
 84: (8)                     for text in self.text:
 85: (12)                        path.write(text.encode("ascii"))
 86: (4)                 @open_file(1, "r")
 87: (4)                 def read(self, path):
 88: (8)                     return path.readlines()[0]
 89: (4)                 @staticmethod
 90: (4)                 @open_file(0, "wb")
 91: (4)                 def writer_arg0(path):
 92: (8)                     path.write(b"demo")
 93: (4)                 @open_file(1, "wb+")
 94: (4)                 def writer_arg1(self, path):
 95: (8)                     self.write(path)
 96: (4)                 @open_file(2, "wb")
 97: (4)                 def writer_arg2default(self, x, path=None):
 98: (8)                     if path is None:
 99: (12)                        with tempfile.NamedTemporaryFile("wb+") as fh:
 100: (16)                           self.write(fh)
 101: (8)                    else:
 102: (12)                       self.write(path)
 103: (4)                @open_file(4, "wb")
 104: (4)                def writer_arg4default(self, x, y, other="hello", path=None, **kwargs):
 105: (8)                    if path is None:
 106: (12)                       with tempfile.NamedTemporaryFile("wb+") as fh:
 107: (16)                           self.write(fh)
 108: (8)                    else:
 109: (12)                       self.write(path)
 110: (4)                @open_file("path", "wb")
 111: (4)                def writer_kwarg(self, **kwargs):
 112: (8)                    path = kwargs.get("path", None)
 113: (8)                    if path is None:
 114: (12)                       with tempfile.NamedTemporaryFile("wb+") as fh:
 115: (16)                           self.write(fh)
 116: (8)                    else:
 117: (12)                       self.write(path)
 118: (4)                def test_writer_arg0_str(self):
 119: (8)                    self.writer_arg0(self.name)
 120: (4)                def test_writer_arg0_fobj(self):
 121: (8)                    self.writer_arg0(self.fobj)
 122: (4)                def test_writer_arg0_pathlib(self):
 123: (8)                    self.writer_arg0(pathlib.Path(self.name))
 124: (4)                def test_writer_arg1_str(self):
 125: (8)                    self.writer_arg1(self.name)
 126: (8)                    assert self.read(self.name) == "".join(self.text)
 127: (4)                def test_writer_arg1_fobj(self):
 128: (8)                    self.writer_arg1(self.fobj)
 129: (8)                    assert not self.fobj.closed
 130: (8)                    self.fobj.close()
 131: (8)                    assert self.read(self.name) == "".join(self.text)
 132: (4)                def test_writer_arg2default_str(self):
 133: (8)                    self.writer_arg2default(0, path=None)
 134: (8)                    self.writer_arg2default(0, path=self.name)
 135: (8)                    assert self.read(self.name) == "".join(self.text)
 136: (4)                def test_writer_arg2default_fobj(self):
 137: (8)                    self.writer_arg2default(0, path=self.fobj)
 138: (8)                    assert not self.fobj.closed
 139: (8)                    self.fobj.close()
 140: (8)                    assert self.read(self.name) == "".join(self.text)
 141: (4)                def test_writer_arg2default_fobj_path_none(self):
```

```
142: (8)                    self.writer_arg2default(0, path=None)
143: (4)                def test_writer_arg4default_fobj(self):
144: (8)                    self.writer_arg4default(0, 1, dog="dog", other="other")
145: (8)                    self.writer_arg4default(0, 1, dog="dog", other="other",
path=self.name)
146: (8)                    assert self.read(self.name) == "".join(self.text)
147: (4)                def test_writer_kwarg_str(self):
148: (8)                    self.writer_kwarg(path=self.name)
149: (8)                    assert self.read(self.name) == "".join(self.text)
150: (4)                def test_writer_kwarg_fobj(self):
151: (8)                    self.writer_kwarg(path=self.fobj)
152: (8)                    self.fobj.close()
153: (8)                    assert self.read(self.name) == "".join(self.text)
154: (4)                def test_writer_kwarg_path_none(self):
155: (8)                    self.writer_kwarg(path=None)
156: (0)            class TestRandomState:
157: (4)                @classmethod
158: (4)                def setup_class(cls):
159: (8)                    global np
160: (8)                    np = pytest.importorskip("numpy")
161: (4)                @np_random_state(1)
162: (4)                def instantiate_np_random_state(self, random_state):
163: (8)                    allowed = (np.random.RandomState, np.random.Generator)
164: (8)                    assert isinstance(random_state, allowed)
165: (8)                    return random_state.random()
166: (4)                @py_random_state(1)
167: (4)                def instantiate_py_random_state(self, random_state):
168: (8)                    allowed = (random.Random, PythonRandomInterface,
PythonRandomViaNumpyBits)
169: (8)                    assert isinstance(random_state, allowed)
170: (8)                    return random_state.random()
171: (4)                def test_random_state_None(self):
172: (8)                    np.random.seed(42)
173: (8)                    rv = np.random.random()
174: (8)                    np.random.seed(42)
175: (8)                    assert rv == self.instantiate_np_random_state(None)
176: (8)                    random.seed(42)
177: (8)                    rv = random.random()
178: (8)                    random.seed(42)
179: (8)                    assert rv == self.instantiate_py_random_state(None)
180: (4)                def test_random_state_np_random(self):
181: (8)                    np.random.seed(42)
182: (8)                    rv = np.random.random()
183: (8)                    np.random.seed(42)
184: (8)                    assert rv == self.instantiate_np_random_state(np.random)
185: (8)                    np.random.seed(42)
186: (8)                    assert rv == self.instantiate_py_random_state(np.random)
187: (4)                def test_random_state_int(self):
188: (8)                    np.random.seed(42)
189: (8)                    np_rv = np.random.random()
190: (8)                    random.seed(42)
191: (8)                    py_rv = random.random()
192: (8)                    np.random.seed(42)
193: (8)                    seed = 1
194: (8)                    rval = self.instantiate_np_random_state(seed)
195: (8)                    rval_expected = np.random.RandomState(seed).rand()
196: (8)                    assert rval == rval_expected
197: (8)                    assert np_rv == np.random.random()
198: (8)                    random.seed(42)
199: (8)                    rval = self.instantiate_py_random_state(seed)
200: (8)                    rval_expected = random.Random(seed).random()
201: (8)                    assert rval == rval_expected
202: (8)                    assert py_rv == random.random()
203: (4)                def test_random_state_np_random_Generator(self):
204: (8)                    np.random.seed(42)
205: (8)                    np_rv = np.random.random()
206: (8)                    np.random.seed(42)
207: (8)                    seed = 1
208: (8)                    rng = np.random.default_rng(seed)
```

```
(8)                 rval = self.instantiate_np_random_state(rng)
(8)                 rval_expected = np.random.default_rng(seed).random()
(8)                 assert rval == rval_expected
(8)                 rval = self.instantiate_py_random_state(rng)
(8)                 rval_expected = np.random.default_rng(seed).random(size=2)[1]
(8)                 assert rval == rval_expected
(8)                 assert np_rv == np.random.random()
(4)             def test_random_state_np_random_RandomState(self):
(8)                 np.random.seed(42)
(8)                 np_rv = np.random.random()
(8)                 np.random.seed(42)
(8)                 seed = 1
(8)                 rng = np.random.RandomState(seed)
(8)                 rval = self.instantiate_np_random_state(rng)
(8)                 rval_expected = np.random.RandomState(seed).random()
(8)                 assert rval == rval_expected
(8)                 rval = self.instantiate_py_random_state(rng)
(8)                 rval_expected = np.random.RandomState(seed).random(size=2)[1]
(8)                 assert rval == rval_expected
(8)                 assert np_rv == np.random.random()
(4)             def test_random_state_py_random(self):
(8)                 seed = 1
(8)                 rng = random.Random(seed)
(8)                 rv = self.instantiate_py_random_state(rng)
(8)                 assert rv == random.Random(seed).random()
(8)                 pytest.raises(ValueError, self.instantiate_np_random_state, rng)
(0)         def test_random_state_string_arg_index():
(4)             with pytest.raises(nx.NetworkXError):
(8)                 @np_random_state("a")
(8)                 def make_random_state(rs):
(12)                    pass
(8)                 rstate = make_random_state(1)
(0)         def test_py_random_state_string_arg_index():
(4)             with pytest.raises(nx.NetworkXError):
(8)                 @py_random_state("a")
(8)                 def make_random_state(rs):
(12)                    pass
(8)                 rstate = make_random_state(1)
(0)         def test_random_state_invalid_arg_index():
(4)             with pytest.raises(nx.NetworkXError):
(8)                 @np_random_state(2)
(8)                 def make_random_state(rs):
(12)                    pass
(8)                 rstate = make_random_state(1)
(0)         def test_py_random_state_invalid_arg_index():
(4)             with pytest.raises(nx.NetworkXError):
(8)                 @py_random_state(2)
(8)                 def make_random_state(rs):
(12)                    pass
(8)                 rstate = make_random_state(1)
(0)         class TestArgmap:
(4)             class ArgmapError(RuntimeError):
(8)                 pass
(4)             def test_trivial_function(self):
(8)                 def do_not_call(x):
(12)                    raise ArgmapError("do not call this function")
(8)                 @argmap(do_not_call)
(8)                 def trivial_argmap():
(12)                    return 1
(8)                 assert trivial_argmap() == 1
(4)             def test_trivial_iterator(self):
(8)                 def do_not_call(x):
(12)                    raise ArgmapError("do not call this function")
(8)                 @argmap(do_not_call)
(8)                 def trivial_argmap():
(12)                    yield from (1, 2, 3)
(8)                 assert tuple(trivial_argmap()) == (1, 2, 3)
(4)             def test_contextmanager(self):
(8)                 container = []
```

```
(8)                     def contextmanager(x):
(12)                        nonlocal container
(12)                        return x, lambda: container.append(x)
(8)                     @argmap(contextmanager, 0, 1, 2, try_finally=True)
(8)                     def foo(x, y, z):
(12)                        return x, y, z
(8)                     x, y, z = foo("a", "b", "c")
(8)                     assert container == ["c", "b", "a"]
(4)                 def test_tryfinally_generator(self):
(8)                     container = []
(8)                     def singleton(x):
(12)                        return (x,)
(8)                     with pytest.raises(nx.NetworkXError):
(12)                        @argmap(singleton, 0, 1, 2, try_finally=True)
(12)                        def foo(x, y, z):
(16)                            yield from (x, y, z)
(8)                     @argmap(singleton, 0, 1, 2)
(8)                     def foo(x, y, z):
(12)                        return x + y + z
(8)                     q = foo("a", "b", "c")
(8)                     assert q == ("a", "b", "c")
(4)                 def test_actual_vararg(self):
(8)                     @argmap(lambda x: -x, 4)
(8)                     def foo(x, y, *args):
(12)                        return (x, y) + tuple(args)
(8)                     assert foo(1, 2, 3, 4, 5, 6) == (1, 2, 3, 4, -5, 6)
(4)                 def test_signature_destroying_intermediate_decorator(self):
(8)                     def add_one_to_first_bad_decorator(f):
(12)                        """Bad because it doesn't wrap the f signature (clobbers it)"""
(12)                        def decorated(a, *args, **kwargs):
(16)                            return f(a + 1, *args, **kwargs)
(12)                        return decorated
(8)                     add_two_to_second = argmap(lambda b: b + 2, 1)
(8)                     @add_two_to_second
(8)                     @add_one_to_first_bad_decorator
(8)                     def add_one_and_two(a, b):
(12)                        return a, b
(8)                     assert add_one_and_two(5, 5) == (6, 7)
(4)                 def test_actual_kwarg(self):
(8)                     @argmap(lambda x: -x, "arg")
(8)                     def foo(*, arg):
(12)                        return arg
(8)                     assert foo(arg=3) == -3
(4)                 def test_nested_tuple(self):
(8)                     def xform(x, y):
(12)                        u, v = y
(12)                        return x + u + v, (x + u, x + v)
(8)                     @argmap(xform, (0, ("t", 2)))
(8)                     def foo(a, *args, **kwargs):
(12)                        return a, args, kwargs
(8)                     a, args, kwargs = foo(1, 2, 3, t=4)
(8)                     assert a == 1 + 4 + 3
(8)                     assert args == (2, 1 + 3)
(8)                     assert kwargs == {"t": 1 + 4}
(4)                 def test_flatten(self):
(8)                     assert tuple(argmap._flatten([[[[[], []], [], []], [], [], []]],
set())) == ()
(8)                     rlist = ["a", ["b", "c"], [["d"], "e"], "f"]
(8)                     assert "".join(argmap._flatten(rlist, set())) == "abcdef"
(4)                 def test_indent(self):
(8)                     code = "\n".join(
(12)                        argmap._indent(
(16)                            *[
(20)                                "try:",
(20)                                "try:",
(20)                                "pass#",
(20)                                "finally:",
(20)                                "pass#",
(20)                                "#",
```

```
346: (20)                                "finally:",
347: (20)                                "pass#",
348: (16)                            ]
349: (12)                        )
350: (8)                    )
351: (8)                    assert (
352: (12)                        code
353: (12)                        == """try:
354: (1)             try:
355: (2)              pass#
356: (1)             finally:
357: (2)              pass#
358: (0)            finally:
359: (1)             pass#"""
360: (8)                    )
361: (4)                def test_immediate_raise(self):
362: (8)                    @not_implemented_for("directed")
363: (8)                    def yield_nodes(G):
364: (12)                       yield from G
365: (8)                    G = nx.Graph([(1, 2)])
366: (8)                    D = nx.DiGraph()
367: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
368: (12)                       node_iter = yield_nodes(D)
369: (8)                    with pytest.raises(nx.NetworkXNotImplemented):
370: (12)                       node_iter = yield_nodes(D)
371: (8)                    node_iter = yield_nodes(G)
372: (8)                    next(node_iter)
373: (8)                    next(node_iter)
374: (8)                    with pytest.raises(StopIteration):
375: (12)                       next(node_iter)
```

File 561 - test_heaps.py:

```
1: (0)              import pytest
2: (0)              import networkx as nx
3: (0)              from networkx.utils import BinaryHeap, PairingHeap
4: (0)              class X:
5: (4)                  def __eq__(self, other):
6: (8)                      raise self is other
7: (4)                  def __ne__(self, other):
8: (8)                      raise self is not other
9: (4)                  def __lt__(self, other):
10: (8)                     raise TypeError("cannot compare")
11: (4)                 def __le__(self, other):
12: (8)                     raise TypeError("cannot compare")
13: (4)                 def __ge__(self, other):
14: (8)                     raise TypeError("cannot compare")
15: (4)                 def __gt__(self, other):
16: (8)                     raise TypeError("cannot compare")
17: (4)                 def __hash__(self):
18: (8)                     return hash(id(self))
19: (0)             x = X()
20: (0)             data = [  # min should not invent an element.
21: (4)                 ("min", nx.NetworkXError),
22: (4)                 ("pop", nx.NetworkXError),
23: (4)                 ("get", 0, None),
24: (4)                 ("get", x, None),
25: (4)                 ("get", None, None),
26: (4)                 ("insert", x, 1, True),
27: (4)                 ("get", x, 1),
28: (4)                 ("min", (x, 1)),
29: (4)                 ("min", (x, 1)),
30: (4)                 ("insert", 1, -2.0, True),
31: (4)                 ("min", (1, -2.0)),
32: (4)                 ("insert", 3, -(10**100), True),
33: (4)                 ("insert", 4, 5, True),
34: (4)                 ("pop", (3, -(10**100))),
```

```
35: (4)                 ("pop", (1, -2.0)),
36: (4)                 ("insert", 4, -50, True),
37: (4)                 ("insert", 4, -60, False, True),
38: (4)                 ("pop", (4, -60)),
39: (4)                 ("pop", (x, 1)),
40: (4)                 ("min", nx.NetworkXError),
41: (4)                 ("pop", nx.NetworkXError),
42: (4)                 ("insert", x, 0, True),
43: (4)                 ("insert", x, 0, False, False),
44: (4)                 ("min", (x, 0)),
45: (4)                 ("insert", x, 0, True, False),
46: (4)                 ("min", (x, 0)),
47: (4)                 ("pop", (x, 0)),
48: (4)                 ("pop", nx.NetworkXError),
49: (4)                 ("insert", None, 0, True),
50: (4)                 ("insert", 2, -1, True),
51: (4)                 ("min", (2, -1)),
52: (4)                 ("insert", 2, 1, True, False),
53: (4)                 ("min", (None, 0)),
54: (4)                 ("insert", None, 2, False, False),
55: (4)                 ("min", (None, 0)),
56: (4)                 ("pop", (None, 0)),
57: (4)                 ("pop", (2, 1)),
58: (4)                 ("min", nx.NetworkXError),
59: (4)                 ("pop", nx.NetworkXError),
60: (0)             ]
61: (0)             def _test_heap_class(cls, *args, **kwargs):
62: (4)                 heap = cls(*args, **kwargs)
63: (4)                 for op in data:
64: (8)                     if op[-1] is not nx.NetworkXError:
65: (12)                        assert op[-1] == getattr(heap, op[0])(*op[1:-1])
66: (8)                     else:
67: (12)                        pytest.raises(op[-1], getattr(heap, op[0]), *op[1:-1])
68: (4)                 for i in range(99, -1, -1):
69: (8)                     assert heap.insert(i, i)
70: (4)                 for i in range(50):
71: (8)                     assert heap.pop() == (i, i)
72: (4)                 for i in range(100):
73: (8)                     assert heap.insert(i, i) == (i < 50)
74: (4)                 for i in range(100):
75: (8)                     assert not heap.insert(i, i + 1)
76: (4)                 for i in range(50):
77: (8)                     assert heap.pop() == (i, i)
78: (4)                 for i in range(100):
79: (8)                     assert heap.insert(i, i + 1) == (i < 50)
80: (4)                 for i in range(49):
81: (8)                     assert heap.pop() == (i, i + 1)
82: (4)                 assert sorted([heap.pop(), heap.pop()]) == [(49, 50), (50, 50)]
83: (4)                 for i in range(51, 100):
84: (8)                     assert not heap.insert(i, i + 1, True)
85: (4)                 for i in range(51, 70):
86: (8)                     assert heap.pop() == (i, i + 1)
87: (4)                 for i in range(100):
88: (8)                     assert heap.insert(i, i)
89: (4)                 for i in range(100):
90: (8)                     assert heap.pop() == (i, i)
91: (4)                 pytest.raises(nx.NetworkXError, heap.pop)
92: (0)             def test_PairingHeap():
93: (4)                 _test_heap_class(PairingHeap)
94: (0)             def test_BinaryHeap():
95: (4)                 _test_heap_class(BinaryHeap)
```

File 562 - test_mapped_queue.py:

```
1: (0)              import pytest
2: (0)              from networkx.utils.mapped_queue import MappedQueue, _HeapElement
3: (0)              def test_HeapElement_gtlt():
```

```
4: (4)                  bar = _HeapElement(1.1, "a")
5: (4)                  foo = _HeapElement(1, "b")
6: (4)                  assert foo < bar
7: (4)                  assert bar > foo
8: (4)                  assert foo < 1.1
9: (4)                  assert 1 < bar
10: (0)             def test_HeapElement_gtlt_tied_priority():
11: (4)                 bar = _HeapElement(1, "a")
12: (4)                 foo = _HeapElement(1, "b")
13: (4)                 assert foo > bar
14: (4)                 assert bar < foo
15: (0)             def test_HeapElement_eq():
16: (4)                 bar = _HeapElement(1.1, "a")
17: (4)                 foo = _HeapElement(1, "a")
18: (4)                 assert foo == bar
19: (4)                 assert bar == foo
20: (4)                 assert foo == "a"
21: (0)             def test_HeapElement_iter():
22: (4)                 foo = _HeapElement(1, "a")
23: (4)                 bar = _HeapElement(1.1, (3, 2, 1))
24: (4)                 assert list(foo) == [1, "a"]
25: (4)                 assert list(bar) == [1.1, 3, 2, 1]
26: (0)             def test_HeapElement_getitem():
27: (4)                 foo = _HeapElement(1, "a")
28: (4)                 bar = _HeapElement(1.1, (3, 2, 1))
29: (4)                 assert foo[1] == "a"
30: (4)                 assert foo[0] == 1
31: (4)                 assert bar[0] == 1.1
32: (4)                 assert bar[2] == 2
33: (4)                 assert bar[3] == 1
34: (4)                 pytest.raises(IndexError, bar.__getitem__, 4)
35: (4)                 pytest.raises(IndexError, foo.__getitem__, 2)
36: (0)             class TestMappedQueue:
37: (4)                 def setup_method(self):
38: (8)                     pass
39: (4)                 def _check_map(self, q):
40: (8)                     assert q.position == {elt: pos for pos, elt in enumerate(q.heap)}
41: (4)                 def _make_mapped_queue(self, h):
42: (8)                     q = MappedQueue()
43: (8)                     q.heap = h
44: (8)                     q.position = {elt: pos for pos, elt in enumerate(h)}
45: (8)                     return q
46: (4)                 def test_heapify(self):
47: (8)                     h = [5, 4, 3, 2, 1, 0]
48: (8)                     q = self._make_mapped_queue(h)
49: (8)                     q._heapify()
50: (8)                     self._check_map(q)
51: (4)                 def test_init(self):
52: (8)                     h = [5, 4, 3, 2, 1, 0]
53: (8)                     q = MappedQueue(h)
54: (8)                     self._check_map(q)
55: (4)                 def test_incomparable(self):
56: (8)                     h = [5, 4, "a", 2, 1, 0]
57: (8)                     pytest.raises(TypeError, MappedQueue, h)
58: (4)                 def test_len(self):
59: (8)                     h = [5, 4, 3, 2, 1, 0]
60: (8)                     q = MappedQueue(h)
61: (8)                     self._check_map(q)
62: (8)                     assert len(q) == 6
63: (4)                 def test_siftup_leaf(self):
64: (8)                     h = [2]
65: (8)                     h_sifted = [2]
66: (8)                     q = self._make_mapped_queue(h)
67: (8)                     q._siftup(0)
68: (8)                     assert q.heap == h_sifted
69: (8)                     self._check_map(q)
70: (4)                 def test_siftup_one_child(self):
71: (8)                     h = [2, 0]
72: (8)                     h_sifted = [0, 2]
```

```
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftup(0)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_siftup_left_child(self):
(8)                     h = [2, 0, 1]
(8)                     h_sifted = [0, 2, 1]
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftup(0)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_siftup_right_child(self):
(8)                     h = [2, 1, 0]
(8)                     h_sifted = [0, 1, 2]
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftup(0)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_siftup_multiple(self):
(8)                     h = [0, 1, 2, 4, 3, 5, 6]
(8)                     h_sifted = [0, 1, 2, 4, 3, 5, 6]
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftup(0)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_siftdown_leaf(self):
(8)                     h = [2]
(8)                     h_sifted = [2]
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftdown(0, 0)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_siftdown_single(self):
(8)                     h = [1, 0]
(8)                     h_sifted = [0, 1]
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftdown(0, len(h) - 1)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_siftdown_multiple(self):
(8)                     h = [1, 2, 3, 4, 5, 6, 7, 0]
(8)                     h_sifted = [0, 1, 3, 2, 5, 6, 7, 4]
(8)                     q = self._make_mapped_queue(h)
(8)                     q._siftdown(0, len(h) - 1)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_push(self):
(8)                     to_push = [6, 1, 4, 3, 2, 5, 0]
(8)                     h_sifted = [0, 2, 1, 6, 3, 5, 4]
(8)                     q = MappedQueue()
(8)                     for elt in to_push:
(12)                        q.push(elt)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_push_duplicate(self):
(8)                     to_push = [2, 1, 0]
(8)                     h_sifted = [0, 2, 1]
(8)                     q = MappedQueue()
(8)                     for elt in to_push:
(12)                        inserted = q.push(elt)
(12)                        assert inserted
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(8)                     inserted = q.push(1)
(8)                     assert not inserted
(4)                 def test_pop(self):
(8)                     h = [3, 4, 6, 0, 1, 2, 5]
(8)                     h_sorted = sorted(h)
(8)                     q = self._make_mapped_queue(h)
```

```
(8)                     q._heapify()
(8)                     popped = [q.pop() for _ in range(len(h))]
(8)                     assert popped == h_sorted
(8)                     self._check_map(q)
(4)                 def test_remove_leaf(self):
(8)                     h = [0, 2, 1, 6, 3, 5, 4]
(8)                     h_removed = [0, 2, 1, 6, 4, 5]
(8)                     q = self._make_mapped_queue(h)
(8)                     removed = q.remove(3)
(8)                     assert q.heap == h_removed
(4)                 def test_remove_root(self):
(8)                     h = [0, 2, 1, 6, 3, 5, 4]
(8)                     h_removed = [1, 2, 4, 6, 3, 5]
(8)                     q = self._make_mapped_queue(h)
(8)                     removed = q.remove(0)
(8)                     assert q.heap == h_removed
(4)                 def test_update_leaf(self):
(8)                     h = [0, 20, 10, 60, 30, 50, 40]
(8)                     h_updated = [0, 15, 10, 60, 20, 50, 40]
(8)                     q = self._make_mapped_queue(h)
(8)                     removed = q.update(30, 15)
(8)                     assert q.heap == h_updated
(4)                 def test_update_root(self):
(8)                     h = [0, 20, 10, 60, 30, 50, 40]
(8)                     h_updated = [10, 20, 35, 60, 30, 50, 40]
(8)                     q = self._make_mapped_queue(h)
(8)                     removed = q.update(0, 35)
(8)                     assert q.heap == h_updated
(0)             class TestMappedDict(TestMappedQueue):
(4)                 def _make_mapped_queue(self, h):
(8)                     priority_dict = {elt: elt for elt in h}
(8)                     return MappedQueue(priority_dict)
(4)                 def test_init(self):
(8)                     d = {5: 0, 4: 1, "a": 2, 2: 3, 1: 4}
(8)                     q = MappedQueue(d)
(8)                     assert q.position == d
(4)                 def test_ties(self):
(8)                     d = {5: 0, 4: 1, 3: 2, 2: 3, 1: 4}
(8)                     q = MappedQueue(d)
(8)                     assert q.position == {elt: pos for pos, elt in enumerate(q.heap)}
(4)                 def test_pop(self):
(8)                     d = {5: 0, 4: 1, 3: 2, 2: 3, 1: 4}
(8)                     q = MappedQueue(d)
(8)                     assert q.pop() == _HeapElement(0, 5)
(8)                     assert q.position == {elt: pos for pos, elt in enumerate(q.heap)}
(4)                 def test_empty_pop(self):
(8)                     q = MappedQueue()
(8)                     pytest.raises(IndexError, q.pop)
(4)                 def test_incomparable_ties(self):
(8)                     d = {5: 0, 4: 0, "a": 0, 2: 0, 1: 0}
(8)                     pytest.raises(TypeError, MappedQueue, d)
(4)                 def test_push(self):
(8)                     to_push = [6, 1, 4, 3, 2, 5, 0]
(8)                     h_sifted = [0, 2, 1, 6, 3, 5, 4]
(8)                     q = MappedQueue()
(8)                     for elt in to_push:
(12)                        q.push(elt, priority=elt)
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(4)                 def test_push_duplicate(self):
(8)                     to_push = [2, 1, 0]
(8)                     h_sifted = [0, 2, 1]
(8)                     q = MappedQueue()
(8)                     for elt in to_push:
(12)                        inserted = q.push(elt, priority=elt)
(12)                        assert inserted
(8)                     assert q.heap == h_sifted
(8)                     self._check_map(q)
(8)                     inserted = q.push(1, priority=1)
```

```
211: (8)                    assert not inserted
212: (4)                def test_update_leaf(self):
213: (8)                    h = [0, 20, 10, 60, 30, 50, 40]
214: (8)                    h_updated = [0, 15, 10, 60, 20, 50, 40]
215: (8)                    q = self._make_mapped_queue(h)
216: (8)                    removed = q.update(30, 15, priority=15)
217: (8)                    assert q.heap == h_updated
218: (4)                def test_update_root(self):
219: (8)                    h = [0, 20, 10, 60, 30, 50, 40]
220: (8)                    h_updated = [10, 20, 35, 60, 30, 50, 40]
221: (8)                    q = self._make_mapped_queue(h)
222: (8)                    removed = q.update(0, 35, priority=35)
223: (8)                    assert q.heap == h_updated

----------------------------------------

File 563 - test_misc.py:

1: (0)              import random
2: (0)              from copy import copy
3: (0)              import pytest
4: (0)              import networkx as nx
5: (0)              from networkx.utils import (
6: (4)                  PythonRandomInterface,
7: (4)                  PythonRandomViaNumpyBits,
8: (4)                  arbitrary_element,
9: (4)                  create_py_random_state,
10: (4)                 create_random_state,
11: (4)                 dict_to_numpy_array,
12: (4)                 discrete_sequence,
13: (4)                 flatten,
14: (4)                 groups,
15: (4)                 make_list_of_ints,
16: (4)                 pairwise,
17: (4)                 powerlaw_sequence,
18: (0)             )
19: (0)             from networkx.utils.misc import _dict_to_numpy_array1, _dict_to_numpy_array2
20: (0)             nested_depth = (
21: (4)                 1,
22: (4)                 2,
23: (4)                 (3, 4, ((5, 6, (7,), (8, (9, 10), 11), (12, 13, (14, 15)), 16), 17), 18,
19),
24: (4)                 20,
25: (0)             )
26: (0)             nested_set = {
27: (4)                 (1, 2, 3, 4),
28: (4)                 (5, 6, 7, 8, 9),
29: (4)                 (10, 11, (12, 13, 14), (15, 16, 17, 18)),
30: (4)                 19,
31: (4)                 20,
32: (0)             }
33: (0)             nested_mixed = [
34: (4)                 1,
35: (4)                 (2, 3, {4, (5, 6), 7}, [8, 9]),
36: (4)                 {10: "foo", 11: "bar", (12, 13): "baz"},
37: (4)                 {(14, 15): "qwe", 16: "asd"},
38: (4)                 (17, (18, "19"), 20),
39: (0)             ]
40: (0)             @pytest.mark.parametrize("result", [None, [], ["existing"], ["existing1",
"existing2"]])
41: (0)             @pytest.mark.parametrize("nested", [nested_depth, nested_mixed, nested_set])
42: (0)             def test_flatten(nested, result):
43: (4)                 if result is None:
44: (8)                     val = flatten(nested, result)
45: (8)                     assert len(val) == 20
46: (4)                 else:
47: (8)                     _result = copy(result)  # because pytest passes parameters as is
48: (8)                     nexisting = len(_result)
49: (8)                     val = flatten(nested, _result)
```

```
 50: (8)                    assert len(val) == len(_result) == 20 + nexisting
 51: (4)                assert issubclass(type(val), tuple)
 52: (0)            def test_make_list_of_ints():
 53: (4)                mylist = [1, 2, 3.0, 42, -2]
 54: (4)                assert make_list_of_ints(mylist) is mylist
 55: (4)                assert make_list_of_ints(mylist) == mylist
 56: (4)                assert type(make_list_of_ints(mylist)[2]) is int
 57: (4)                pytest.raises(nx.NetworkXError, make_list_of_ints, [1, 2, 3, "kermit"])
 58: (4)                pytest.raises(nx.NetworkXError, make_list_of_ints, [1, 2, 3.1])
 59: (0)            def test_random_number_distribution():
 60: (4)                z = powerlaw_sequence(20, exponent=2.5)
 61: (4)                z = discrete_sequence(20, distribution=[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 3])
 62: (0)            class TestNumpyArray:
 63: (4)                @classmethod
 64: (4)                def setup_class(cls):
 65: (8)                    global np
 66: (8)                    np = pytest.importorskip("numpy")
 67: (4)                def test_numpy_to_list_of_ints(self):
 68: (8)                    a = np.array([1, 2, 3], dtype=np.int64)
 69: (8)                    b = np.array([1.0, 2, 3])
 70: (8)                    c = np.array([1.1, 2, 3])
 71: (8)                    assert type(make_list_of_ints(a)) == list
 72: (8)                    assert make_list_of_ints(b) == list(b)
 73: (8)                    B = make_list_of_ints(b)
 74: (8)                    assert type(B[0]) == int
 75: (8)                    pytest.raises(nx.NetworkXError, make_list_of_ints, c)
 76: (4)                def test__dict_to_numpy_array1(self):
 77: (8)                    d = {"a": 1, "b": 2}
 78: (8)                    a = _dict_to_numpy_array1(d, mapping={"a": 0, "b": 1})
 79: (8)                    np.testing.assert_allclose(a, np.array([1, 2]))
 80: (8)                    a = _dict_to_numpy_array1(d, mapping={"b": 0, "a": 1})
 81: (8)                    np.testing.assert_allclose(a, np.array([2, 1]))
 82: (8)                    a = _dict_to_numpy_array1(d)
 83: (8)                    np.testing.assert_allclose(a.sum(), 3)
 84: (4)                def test__dict_to_numpy_array2(self):
 85: (8)                    d = {"a": {"a": 1, "b": 2}, "b": {"a": 10, "b": 20}}
 86: (8)                    mapping = {"a": 1, "b": 0}
 87: (8)                    a = _dict_to_numpy_array2(d, mapping=mapping)
 88: (8)                    np.testing.assert_allclose(a, np.array([[20, 10], [2, 1]]))
 89: (8)                    a = _dict_to_numpy_array2(d)
 90: (8)                    np.testing.assert_allclose(a.sum(), 33)
 91: (4)                def test_dict_to_numpy_array_a(self):
 92: (8)                    d = {"a": {"a": 1, "b": 2}, "b": {"a": 10, "b": 20}}
 93: (8)                    mapping = {"a": 0, "b": 1}
 94: (8)                    a = dict_to_numpy_array(d, mapping=mapping)
 95: (8)                    np.testing.assert_allclose(a, np.array([[1, 2], [10, 20]]))
 96: (8)                    mapping = {"a": 1, "b": 0}
 97: (8)                    a = dict_to_numpy_array(d, mapping=mapping)
 98: (8)                    np.testing.assert_allclose(a, np.array([[20, 10], [2, 1]]))
 99: (8)                    a = _dict_to_numpy_array2(d)
100: (8)                    np.testing.assert_allclose(a.sum(), 33)
101: (4)                def test_dict_to_numpy_array_b(self):
102: (8)                    d = {"a": 1, "b": 2}
103: (8)                    mapping = {"a": 0, "b": 1}
104: (8)                    a = dict_to_numpy_array(d, mapping=mapping)
105: (8)                    np.testing.assert_allclose(a, np.array([1, 2]))
106: (8)                    a = _dict_to_numpy_array1(d)
107: (8)                    np.testing.assert_allclose(a.sum(), 3)
108: (0)            def test_pairwise():
109: (4)                nodes = range(4)
110: (4)                node_pairs = [(0, 1), (1, 2), (2, 3)]
111: (4)                node_pairs_cycle = node_pairs + [(3, 0)]
112: (4)                assert list(pairwise(nodes)) == node_pairs
113: (4)                assert list(pairwise(iter(nodes))) == node_pairs
114: (4)                assert list(pairwise(nodes, cyclic=True)) == node_pairs_cycle
115: (4)                empty_iter = iter(())
116: (4)                assert list(pairwise(empty_iter)) == []
117: (4)                empty_iter = iter(())
118: (4)                assert list(pairwise(empty_iter, cyclic=True)) == []
```

```
(0)             def test_groups():
(4)                 many_to_one = dict(zip("abcde", [0, 0, 1, 1, 2]))
(4)                 actual = groups(many_to_one)
(4)                 expected = {0: {"a", "b"}, 1: {"c", "d"}, 2: {"e"}}
(4)                 assert actual == expected
(4)                 assert {} == groups({})
(0)             def test_create_random_state():
(4)                 np = pytest.importorskip("numpy")
(4)                 rs = np.random.RandomState
(4)                 assert isinstance(create_random_state(1), rs)
(4)                 assert isinstance(create_random_state(None), rs)
(4)                 assert isinstance(create_random_state(np.random), rs)
(4)                 assert isinstance(create_random_state(rs(1)), rs)
(4)                 rng = np.random.default_rng()
(4)                 assert isinstance(create_random_state(rng), np.random.Generator)
(4)                 pytest.raises(ValueError, create_random_state, "a")
(4)                 assert np.all(rs(1).rand(10) == create_random_state(1).rand(10))
(0)             def test_create_py_random_state():
(4)                 pyrs = random.Random
(4)                 assert isinstance(create_py_random_state(1), pyrs)
(4)                 assert isinstance(create_py_random_state(None), pyrs)
(4)                 assert isinstance(create_py_random_state(pyrs(1)), pyrs)
(4)                 pytest.raises(ValueError, create_py_random_state, "a")
(4)                 np = pytest.importorskip("numpy")
(4)                 rs = np.random.RandomState
(4)                 rng = np.random.default_rng(1000)
(4)                 rng_explicit = np.random.Generator(np.random.SFC64())
(4)                 old_nprs = PythonRandomInterface
(4)                 nprs = PythonRandomViaNumpyBits
(4)                 assert isinstance(create_py_random_state(np.random), nprs)
(4)                 assert isinstance(create_py_random_state(rs(1)), old_nprs)
(4)                 assert isinstance(create_py_random_state(rng), nprs)
(4)                 assert isinstance(create_py_random_state(rng_explicit), nprs)
(4)                 assert isinstance(PythonRandomInterface(), old_nprs)
(4)                 assert isinstance(PythonRandomViaNumpyBits(), nprs)
(4)                 int64max = 9223372036854775807  # from np.iinfo(np.int64).max
(4)                 for r in (rng, rs(1)):
(8)                     prs = create_py_random_state(r)
(8)                     prs.randrange(3, int64max + 5)
(8)                     prs.randint(3, int64max + 5)
(0)             def test_PythonRandomInterface_RandomState():
(4)                 np = pytest.importorskip("numpy")
(4)                 seed = 42
(4)                 rs = np.random.RandomState
(4)                 rng = PythonRandomInterface(rs(seed))
(4)                 rs42 = rs(seed)
(4)                 assert rng.randrange(3, 5) == rs42.randint(3, 5)
(4)                 assert rng.choice([1, 2, 3]) == rs42.choice([1, 2, 3])
(4)                 assert rng.gauss(0, 1) == rs42.normal(0, 1)
(4)                 assert rng.expovariate(1.5) == rs42.exponential(1 / 1.5)
(4)                 assert np.all(rng.shuffle([1, 2, 3]) == rs42.shuffle([1, 2, 3]))
(4)                 assert np.all(
(8)                     rng.sample([1, 2, 3], 2) == rs42.choice([1, 2, 3], (2,),
replace=False)
(4)                 )
(4)                 assert np.all(
(8)                     [rng.randint(3, 5) for _ in range(100)]
(8)                     == [rs42.randint(3, 6) for _ in range(100)]
(4)                 )
(4)                 assert rng.random() == rs42.random_sample()
(0)             def test_PythonRandomInterface_Generator():
(4)                 np = pytest.importorskip("numpy")
(4)                 seed = 42
(4)                 rng = np.random.default_rng(seed)
(4)                 pri = PythonRandomInterface(np.random.default_rng(seed))
(4)                 assert pri.randrange(3, 5) == rng.integers(3, 5)
(4)                 assert pri.choice([1, 2, 3]) == rng.choice([1, 2, 3])
(4)                 assert pri.gauss(0, 1) == rng.normal(0, 1)
(4)                 assert pri.expovariate(1.5) == rng.exponential(1 / 1.5)
```

```
187: (4)                assert np.all(pri.shuffle([1, 2, 3]) == rng.shuffle([1, 2, 3]))
188: (4)                assert np.all(
189: (8)                    pri.sample([1, 2, 3], 2) == rng.choice([1, 2, 3], (2,), replace=False)
190: (4)                )
191: (4)                assert np.all(
192: (8)                    [pri.randint(3, 5) for _ in range(100)]
193: (8)                    == [rng.integers(3, 6) for _ in range(100)]
194: (4)                )
195: (4)                assert pri.random() == rng.random()
196: (0)            @pytest.mark.parametrize(
197: (4)                ("iterable_type", "expected"), ((list, 1), (tuple, 1), (str, "["), (set,
1))
198: (0)            )
199: (0)            def test_arbitrary_element(iterable_type, expected):
200: (4)                iterable = iterable_type([1, 2, 3])
201: (4)                assert arbitrary_element(iterable) == expected
202: (0)            @pytest.mark.parametrize(
203: (4)                "iterator",
204: (4)                ((i for i in range(3)), iter([1, 2, 3])),  # generator
205: (0)            )
206: (0)            def test_arbitrary_element_raises(iterator):
207: (4)                """Value error is raised when input is an iterator."""
208: (4)                with pytest.raises(ValueError, match="from an iterator"):
209: (8)                    arbitrary_element(iterator)
```

File 564 - test_random_sequence.py:

```
1: (0)              import pytest
2: (0)              from networkx.utils import (
3: (4)                  powerlaw_sequence,
4: (4)                  random_weighted_sample,
5: (4)                  weighted_choice,
6: (4)                  zipf_rv,
7: (0)              )
8: (0)              def test_degree_sequences():
9: (4)                  seq = powerlaw_sequence(10, seed=1)
10: (4)                 seq = powerlaw_sequence(10)
11: (4)                 assert len(seq) == 10
12: (0)             def test_zipf_rv():
13: (4)                 r = zipf_rv(2.3, xmin=2, seed=1)
14: (4)                 r = zipf_rv(2.3, 2, 1)
15: (4)                 r = zipf_rv(2.3)
16: (4)                 assert type(r), int
17: (4)                 pytest.raises(ValueError, zipf_rv, 0.5)
18: (4)                 pytest.raises(ValueError, zipf_rv, 2, xmin=0)
19: (0)             def test_random_weighted_sample():
20: (4)                 mapping = {"a": 10, "b": 20}
21: (4)                 s = random_weighted_sample(mapping, 2, seed=1)
22: (4)                 s = random_weighted_sample(mapping, 2)
23: (4)                 assert sorted(s) == sorted(mapping.keys())
24: (4)                 pytest.raises(ValueError, random_weighted_sample, mapping, 3)
25: (0)             def test_random_weighted_choice():
26: (4)                 mapping = {"a": 10, "b": 0}
27: (4)                 c = weighted_choice(mapping, seed=1)
28: (4)                 c = weighted_choice(mapping)
29: (4)                 assert c == "a"
```

File 565 - test_rcm.py:

```
1: (0)              import networkx as nx
2: (0)              from networkx.utils import reverse_cuthill_mckee_ordering
3: (0)              def test_reverse_cuthill_mckee():
4: (4)                  G = nx.Graph(
5: (8)                      [
6: (12)                         (0, 3),
```

```
7: (12)                         (0, 5),
8: (12)                         (1, 2),
9: (12)                         (1, 4),
10: (12)                        (1, 6),
11: (12)                        (1, 9),
12: (12)                        (2, 3),
13: (12)                        (2, 4),
14: (12)                        (3, 5),
15: (12)                        (3, 8),
16: (12)                        (4, 6),
17: (12)                        (5, 6),
18: (12)                        (5, 7),
19: (12)                        (6, 7),
20: (8)                     ]
21: (4)                 )
22: (4)                 rcm = list(reverse_cuthill_mckee_ordering(G))
23: (4)                 assert rcm in [[0, 8, 5, 7, 3, 6, 2, 4, 1, 9], [0, 8, 5, 7, 3, 6, 4, 2, 1,
9]]
24: (0)             def test_rcm_alternate_heuristic():
25: (4)                 G = nx.Graph(
26: (8)                     [
27: (12)                        (0, 0),
28: (12)                        (0, 4),
29: (12)                        (1, 1),
30: (12)                        (1, 2),
31: (12)                        (1, 5),
32: (12)                        (1, 7),
33: (12)                        (2, 2),
34: (12)                        (2, 4),
35: (12)                        (3, 3),
36: (12)                        (3, 6),
37: (12)                        (4, 4),
38: (12)                        (5, 5),
39: (12)                        (5, 7),
40: (12)                        (6, 6),
41: (12)                        (7, 7),
42: (8)                     ]
43: (4)                 )
44: (4)                 answers = [
45: (8)                     [6, 3, 5, 7, 1, 2, 4, 0],
46: (8)                     [6, 3, 7, 5, 1, 2, 4, 0],
47: (8)                     [7, 5, 1, 2, 4, 0, 6, 3],
48: (4)                 ]
49: (4)                 def smallest_degree(G):
50: (8)                     deg, node = min((d, n) for n, d in G.degree())
51: (8)                     return node
52: (4)                 rcm = list(reverse_cuthill_mckee_ordering(G, heuristic=smallest_degree))
53: (4)                 assert rcm in answers

----------------------------------------

File 566 - test_unionfind.py:

1: (0)              import networkx as nx
2: (0)              def test_unionfind():
3: (4)                  x = nx.utils.UnionFind()
4: (4)                  x.union(0, "a")
5: (0)              def test_subtree_union():
6: (4)                  uf = nx.utils.UnionFind()
7: (4)                  uf.union(1, 2)
8: (4)                  uf.union(3, 4)
9: (4)                  uf.union(4, 5)
10: (4)                 uf.union(1, 5)
11: (4)                 assert list(uf.to_sets()) == [{1, 2, 3, 4, 5}]
12: (0)             def test_unionfind_weights():
13: (4)                 uf = nx.utils.UnionFind()
14: (4)                 uf.union(1, 4, 7)
15: (4)                 uf.union(2, 5, 8)
16: (4)                 uf.union(3, 6, 9)
```

```
17: (4)                 uf.union(1, 2, 3, 4, 5, 6, 7, 8, 9)
18: (4)                 assert uf.weights[uf[1]] == 9
19: (0)             def test_unbalanced_merge_weights():
20: (4)                 uf = nx.utils.UnionFind()
21: (4)                 uf.union(1, 2, 3)
22: (4)                 uf.union(4, 5, 6, 7, 8, 9)
23: (4)                 assert uf.weights[uf[1]] == 3
24: (4)                 assert uf.weights[uf[4]] == 6
25: (4)                 largest_root = uf[4]
26: (4)                 uf.union(1, 4)
27: (4)                 assert uf[1] == largest_root
28: (4)                 assert uf.weights[largest_root] == 9
29: (0)             def test_empty_union():
30: (4)                 uf = nx.utils.UnionFind((0, 1))
31: (4)                 uf.union()
32: (4)                 assert uf[0] == 0
33: (4)                 assert uf[1] == 1
```

File 567 - SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRYCOMBINER_aligner_20_characters_for_pythons_codes.py:

```
1: (0)              import os
2: (0)              from datetime import datetime
3: (0)              def get_file_info(root_folder):
4: (4)                  file_info_list = []
5: (4)                  for root, dirs, files in os.walk(root_folder):
6: (8)                      for file in files:
7: (12)                         try:
8: (16)                             if file.endswith('.py'):
9: (20)                                 file_path = os.path.join(root, file)
10: (20)                                creation_time =
datetime.fromtimestamp(os.path.getctime(file_path))
11: (20)                                modified_time =
datetime.fromtimestamp(os.path.getmtime(file_path))
12: (20)                                file_extension = os.path.splitext(file)[1].lower()
13: (20)                                file_info_list.append([file, file_path, creation_time,
modified_time, file_extension, root])
14: (12)                        except Exception as e:
15: (16)                            print(f"Error processing file {file}: {e}")
16: (4)                 file_info_list.sort(key=lambda x: (x[2], x[3], len(x[0]), x[4]))  # Sort
by creation, modification time, name length, extension
17: (4)                 return file_info_list
18: (0)             def process_file(file_info_list):
19: (4)                 combined_output = []
20: (4)                 for idx, (file_name, file_path, creation_time, modified_time,
file_extension, root) in enumerate(file_info_list):
21: (8)                     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
22: (12)                        content = f.read()
23: (12)                        content = "\n".join([line for line in content.split('\n') if
line.strip() and not line.strip().startswith("#")])
24: (12)                        content = content.replace('\t', '    ')
25: (12)                        processed_lines = []
26: (12)                        for i, line in enumerate(content.split('\n')):
27: (16)                            leading_spaces = len(line) - len(line.lstrip(' '))
28: (16)                            line_number_str = f"{i+1}: ({leading_spaces})"
29: (16)                            padding = ' ' * (20 - len(line_number_str))
30: (16)                            processed_line = f"{line_number_str}{padding}{line}"
31: (16)                            processed_lines.append(processed_line)
32: (12)                        content_with_line_numbers = "\n".join(processed_lines)
33: (12)                        combined_output.append(f"File {idx + 1} - {file_name}:\n")
34: (12)                        combined_output.append(content_with_line_numbers)
35: (12)                        combined_output.append("\n" + "-"*40 + "\n")
36: (4)                 return combined_output
37: (0)             root_folder_path = '.'  # Set this to the desired folder
38: (0)             file_info_list = get_file_info(root_folder_path)
39: (0)             combined_output = process_file(file_info_list)
```

```
40: (0)             output_file =
'SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt'
41: (0)             with open(output_file, 'w', encoding='utf-8') as logfile:
42: (4)                 logfile.write("\n".join(combined_output))
43: (0)             print(f"Processed file info logged to {output_file}")

----------------------------------------
```